BASIC MIPS ARCHITECTURE
VERSION 1.4

TECHNICAL REPORT

Nicolas Hamilton, Major, USAF
AFIT/EN/TR-19-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Disclaimer: The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States. This document has been approved for public release; distribution unlimited, case number 88ABW-2019-5340

AFIT/EN/TR-19-02

Abstract

This report describes in detail the architecture of a MIPS-like processor used for Adaptive-Hybrid Redundancy research. There may be many other applications for this MIPS-like processor beyond this specific research area.

Table of Contents

Page

Abstract ii

List of Figures vii

List of Tables ix

I. Introduction 1

II. Basic MIPS 2

2.1 NAND 4
2.2 Inverter 6
2.3 Multiplexers 7
2.4 Left Shift 17
2.5 Left Shift 2 20
2.6 Right Shift 20
2.7 Bitwise Operations 23
2.8 Adder 26
2.9 Load Upper Immediate 36
2.10 Overflow Detection 37
2.11 Comparator 40
2.12 Signed/Unsigned Extensions 48
2.12.1 Signed 16-Bit to 32-Bit Extension 48
2.12.2 Unsigned 16-Bit to 32-Bit Extension 49
2.12.3 Unsigned 5-Bit to 32-Bit Extension 49
2.13 1-Bit Register 49
2.14 Arithmetic Logic Unit 50
2.15 5-Bit to 32-Bit Decoder 62
2.16 Registers 71
2.16.1 N-Bit Register 71
2.16.2 Enabled Register 71
2.16.3 General Purpose Register (GPR) Bank 72
2.17 Datapath 74
2.18 Instruction Decoder 87
2.19 Encoders 138
2.19.1 Encoder6_3 138
2.19.2 Encoder9_4 140
2.20 Instruction Decoder/Encoder 143
2.21 Finite State Machine Control Signal Generator 150
2.22 Controller Finite State Machine (FSM) 160
2.23 Top Level of Basic MIPS Architecture 169

Page

III. Simulation Testing 172

Appendix A. Basic MIPS VHDL Code 173

1.1 NAND 173
1.2 Inverter 174
1.2.1 1-Bit Inverter 174
1.2.2 N-Bit Inverter 175
1.3 Multiplexers 176
1.3.1 2 Input, 1-Bit Multiplexer 176
1.3.2 2 Input, N-Bit Multiplexer 178
1.3.3 4 Input, 1-Bit Multiplexer 179
1.3.4 4 Input, N-Bit Multiplexer 181
1.3.5 8 Input, 1-Bit Multiplexer 183
1.3.6 8 Input, N-Bit Multiplexer 185
1.3.7 16 Input, 1-Bit Multiplexer 187
1.3.8 16 Input, N-Bit Multiplexer 190
1.3.9 32 Input, 1-Bit Multiplexer 193
1.3.10 32 Input, N-Bit Multiplexer 197
1.4 Left Shift 201
1.5 Left Shift 2 230
1.6 Right Shift 231
1.7 Bitwise Operations 260
1.7.1 2-bit Bitwise Operations 260
1.7.2 2-bit Bitwise Operations 263
1.8 Adder 265
1.8.1 1-bit Full Adder 265
1.8.2 4-bit Full Adder 269
1.8.3 4-bit Carry Select Adder 272
1.8.4 32-bit Carry Select Adder 276
1.8.5 Self-Contained 1-bit Full Adder 280
1.8.6 Self-Contained 2-bit Full Adder 285
1.8.7 Self-Contained 4-bit Full Adder 287
1.8.8 Self-Contained 2-bit Carry Select Adder 289
1.8.9 Self-Contained 4-bit Carry Select Adder 292
1.8.10 Self-Contained 30-bit Carry Select Adder 295
1.9 Load Upper Immediate 298
1.10 Overflow Detection 299
1.11 Comparator 304
1.12 Signed/Unsigned Extensions 326
1.12.1 Signed 16-Bit to 32-Bit Extension 326
1.12.2 Unsigned 16-Bit to 32-Bit Extension 327
1.12.3 Unsigned 5-Bit to 32-Bit Extension 328
1.13 1-Bit Register 329

Page

1.14 ALU Core . . . 330
1.15 5-Bit to 32-Bit Decoder . . . 342
1.16 N-Bit Register . . . 355
1.17 Enabled Register . . . 356
1.18 GPR Bank . . . 358
1.19 Datapath . . . 363
1.20 Instruction Decoder . . . 377
1.21 Encoders . . . 416
1.21.1 Encoder6_3 . . . 416
1.21.2 Encoder9_4 . . . 420
1.22 Instruction Decoder/Encoder . . . 425
1.23 Finite State Machine Control Signal Generator . . . 433
1.24 Controller Finite State Machine (FSM) . . . 443
1.25 Basic MIPS Top Level . . . 454

Appendix B. Testbench VHDL Code . . . 459

2.1 NAND . . . 459
2.2 Inverter . . . 461
2.2.1 1-Bit Inverter . . . 461
2.2.2 N-Bit Inverter . . . 462
2.3 Multiplexers . . . 464
2.3.1 2 Input, 1-Bit Multiplexer . . . 464
2.3.2 2 Input, N-Bit Multiplexer . . . 466
2.3.3 4 Input, 1-Bit Multiplexer . . . 468
2.3.4 4 Input, N-Bit Multiplexer . . . 470
2.3.5 8 Input, 1-Bit Multiplexer . . . 472
2.3.6 8 Input, N-Bit Multiplexer . . . 474
2.3.7 16 Input, 1-Bit Multiplexer . . . 476
2.3.8 16 Input, N-Bit Multiplexer . . . 479
2.3.9 32 Input, 1-Bit Multiplexer . . . 483
2.3.10 32 Input, N-Bit Multiplexer . . . 488
2.4 Left Shift . . . 494
2.5 Left Shift 2 . . . 497
2.6 Right Shift . . . 498
2.7 Bitwise Operations . . . 502
2.7.1 2-bit Bitwise Operations . . . 502
2.7.2 2-bit Bitwise Operations . . . 504
2.8 Adder . . . 506
2.8.1 1-bit Full Adder . . . 506
2.8.2 4-bit Full Adder . . . 508
2.8.3 4-bit Carry Select Adder . . . 535
2.8.4 32-bit Carry Select Adder . . . 562
2.8.5 Self-Contained 1-bit Full Adder . . . 564

Page

2.8.6 Self-Contained 2-bit Full Adder . . . 566
2.8.7 Self-Contained 4-bit Full Adder . . . 569
2.8.8 Self-Contained 2-bit Carry Select Adder . . . 595
2.8.9 Self-Contained 4-bit Carry Select Adder . . . 598
2.8.10 Self-Contained 30-bit Carry Select Adder . . . 624
2.9 Load Upper Immediate . . . 626
2.10 Overflow Detection . . . 627
2.11 Comparator . . . 631
2.12 Signed/Unsigned Extensions . . . 643
2.12.1 Signed 16-Bit to 32-Bit Extension . . . 643
2.12.2 Unsigned 5-Bit to 32-Bit Extension . . . 644
2.13 1-Bit Register . . . 645
2.14 5-Bit to 32-Bit Decoder . . . 647
2.15 N-Bit Register . . . 650
2.16 Enabled Register . . . 652
2.17 GPR Bank . . . 655
2.18 Instruction Decoder . . . 662
2.19 Encoders . . . 668
2.19.1 Encoder6_3 . . . 668
2.19.2 Encoder9_4 . . . 670
2.20 Instruction Decoder/Encoder . . . 672
2.21 Finite State Machine Control Signal Generator . . . 678
2.22 Memory Emulator . . . 681
2.23 Controller Finite State Machine (FSM) . . . 686
2.24 Basic MIPS Top Level . . . 690

Appendix C. Version History . . . 693

Bibliography . . . 695

List of Figures

Figure		Page
1	myNAND2	5
2	Schematic Legend	5
3	Inverter Schematic	6
4	Inverter Symbol	6
5	2-Input Multiplexer Schematic	10
6	2-Input Multiplexer Symbol	10
7	4-Input Multiplexer Schematic	11
8	4-Input Multiplexer Symbol	11
9	8-Input Multiplexer Schematic	12
10	8-Input Multiplexer Symbol	12
11	16-Input Multiplexer Schematic	13
12	16-Input Multiplexer Symbol	14
13	32-Input Multiplexer Schematic	15
14	32-Input Multiplexer Symbol	16
15	Left Shift Schematic	19
16	Right Shift Schematic	22
17	Bitwise_Ops2 Schematic	25
18	ALU Core Schematic	57
19	Enabled Register Schematic	72
20	GPR Bank Schematic	74
21	Datapath Schematic	86
22	Instruction Decoder/Encoder Schematic	148

Figure Page

23 Controller FSM Schematic 167

List of Tables

Table		Page
1	Truth Table for NAND Gate	5
2	myNAND2 Inputs and Outputs	5
3	Truth Table for NAND Gate Configured as an Inverter	6
4	myINV Inputs and Outputs	6
5	myINV_N Inputs and Outputs	7
6	Truth Table for 4 Input Multiplexer	8
7	All Multiplexers Designed for Basic MIPS Implementation	9
8	LEFT_SHIFT Inputs and Outputs	18
9	SLL2_32 Inputs and Outputs	20
10	RIGHT_SHIFT Inputs and Outputs	21
11	Bitwise_Ops2 Inputs and Outputs	25
12	Bitwise_Ops2_32 Inputs and Outputs	26
13	Full Adder Truth Table	27
14	FULL_ADDER_1 Inputs and Outputs	29
15	FULL_ADDER_4 Inputs and Outputs	30
16	CARRY_SELECT_ADDER_4 Inputs and Outputs	32
17	CARRY_SELECT_ADDER_4 Connections	32
18	CARRY_SELECT_ADDER_32 Inputs and Outputs	34
19	FULL_ADDER_1_SC (Self-Contained) Inputs and Outputs	35
20	FULL_ADDER_2_SC Inputs and Outputs	35
21	FULL_ADDER_4_SC Inputs and Outputs	35

Table Page

22 CARRY_SELECT_ADDER_2_SC Inputs and Outputs 36

23 CARRY_SELECT_ADDER_4_SC Inputs and Outputs 36

24 CARRY_SELECT_ADDER_30_SC Inputs and Outputs 36

25 LUI Inputs and Outputs 37

26 Over_Detect Inputs and Outputs 39

27 Comparator Inputs and Outputs 47

28 Comparator Connections 48

29 Sign_Extend16_32 Inputs and Outputs 48

30 Extend16_32 Inputs and Outputs 49

31 Extend5_32 Inputs and Outputs 49

32 myReg1 Inputs and Outputs 50

33 ALU Core Inputs and Outputs 51

34 ALU Core Connections 51

35 ALU Control Signals Cycle 1 58

36 ALU Control Signals Cycle 2 60

37 Decoder5_32 Inputs and Outputs 70

38 myRegN Inputs and Outputs 71

39 Enabled_Register Inputs and Outputs 71

40 GPR_Bank Inputs and Outputs 73

41 Datapath Inputs and Outputs 78

42 Datapath Connections 79

43 Instruction Defining Characteristics 88

44 Instruction_Decoder Signal Summary 132

45 Instruction_Decoder Inputs and Outputs 137

Table Page

46 Encoder6_3 Truth Table 139

47 Encoder6_3 Inputs and Outputs 140

48 Encoder9_4 Truth Table 141

49 Encoder9_4 Inputs and Outputs 143

50 Instruction_Dec_Enc Connections 145

51 Instruction_Dec_Enc Inputs and Outputs 149

52 State Dependent Control Signals 151

53 MEM_ADDRESS_SEL Karnaugh Map 153

54 DO_NOT_STORE Karnaugh Map 157

55 FSM_Control_Signal_Generator Signal Summary 159

56 FSM_Control_Signal_Generator Inputs and Outputs 160

57 FSM States 161

58 Controller FSM Connections 162

59 Controller FSM Inputs and Outputs 168

60 Basic MIPS Connections 169

61 Basic MIPS Inputs and Outputs 171

I. Introduction

The goal of this project was to evaluate the power consumption of a Field Programmable Gate Array (FPGA) when implementing no radiation mitigation strategy, Triple Modular Redundancy (TMR) in hardware, Temporal Software Redundancy (TSR), and a new method called Adaptive-Hybrid Redundancy. To accomplish this, a MIPS-like architecture is created in Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) such that it can be programmed onto an FPGA. This method also allows for errors to be injected into these various architectures at will. Chapter II details the design of this MIPS-like architecture called Basic MIPS. Chapter III briefly outlines the test approach used to verify the functionality of Basic MIPS. The Appendices contain all of the VHDL code used to build and test the Basic MIPS architecture.

II. Basic MIPS

The MIPS-like architecture was based upon the MIPS32™ architecture as specified in MIPS Technologies "MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture" and "MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set"[1, 2]. The following MIPS32™ commands were implemented:

1. Shift Word Left Logical (SLL)
2. No Operation (NOP)
3. Shift Word Right Logical (SRL)
4. Shift Word Right Arithmetic (SRA)
5. Shift Word Left Logical Variable (SLLV)
6. Shift Word Right Logical Variable (SRLV)
7. Shift Word Right Arithmetic Variable (SRAV)
8. Add Word (ADD)
9. Add Unsigned Word (ADDU)
10. Subtract Word (SUB)
11. Subtract Unsigned Word (SUBU)
12. Bitwise Logical And (AND)
13. Bitwise Logical Or (OR)
14. Bitwise Logical Exclusive Or (XOR)
15. Bitwise Logical Not Or (NOR)
16. Set on Less Than (SLT)
17. Set on Less Than Unsigned (SLTU)
18. Branch on Greater Than or Equal To Zero (BGEZ)
19. Branch on Less Than Zero (BLTZ)
20. Branch on Equal (BEQ)

21. Branch on Not Equal (BNE)
22. Branch on Less Than or Equal to Zero (BLEZ)
23. Branch on Greater Than Zero (BGTZ)
24. Add Immediate Word (ADDI)
25. Add Immediate Unsigned Word (ADDIU)
26. Set on Less Than Immediate (SLTI)
27. Set on Less Than Immediate Unsigned (SLTIU)
28. Bitwise Logical And Immediate (ANDI)
29. Bitwise Logical Or Immediate (ORI)
30. Bitwise Logical Exclusive Or Immediate (XORI)
31. Load Upper Immediate (LUI)
32. Load Word (LW)
33. Store Word (SW)

These commands are implemented exactly as specified in MIPS Volume II, with two notable exceptions. For signed addition (ADD, ADDI) and subtraction (SUB), when an overflow occurs, no exception signal is created; however, the destination register retains its previous value as specified. For branch instructions (BGEZ, BLTZ, BEQ, BNE, BLEZ, BGTZ), the offset is added to the current address rather than the instruction following the branch. The documentation refers to the instruction following the branch as being in the branch delay slot and specifies that this instruction must be executed before the branch is taken. To simplify implementation, a branch delay slot was not used. As an example, consider that the program counter (PC) is at 1000 when a branch instruction is encountered. Also suppose that the branch instruction will cause a jump of 100 if the branch condition is true. According to the MIPS documentation, if the branch evaluates to true, instruction at PC=1004 should be evaluated before evaluating the instruction at PC=1104. In the Basic MIPS implementation, if the branch evaluates to true, the next instruction evaluated is at

PC=1100. In both implementations, if the branch condition is false, the instruction at PC=1004 is evaluated next.

The Basic MIPS architecture is divided into a Controller and a Datapath. The Controller consists of a finite state machine (FSM), a register to store the instructions, a module that generates state dependent control signals, and an instruction decoding and encoding module that decodes the instructions and sends control signals to the Datapath that account for the current state of the FSM. The Datapath consists of a bank of 32-32-bit registers as well as the program counter, an arithmetic logic unit, and multiplexers that react to control signals from the Controller to adjust the flow of inputs to and outputs from these main blocks. With this basic top level design in mind, implementation started from the bottom level of the Datapath and worked upward toward the top level. A design choice was made to implement all components using a 2-input, single bit NAND gate as the basic fundamental unit upon which to build all combinational logic. The 1-bit D flip-flop was selected as the basic building block for all registers in this design. This methodology was chosen so that creating and tracing transient upsets from source to destination would be a straightforward task. In other words, if a single transient bit flip in a combinational part of the architecture were simulated, it could be traced forward to where that transient would be masked or latched in a register.

2.1 NAND

The NAND gate (myNAND2) is implemented according to the truth table shown below. The inputs to the NAND gate are A and B and the output is Z. A schematics of the NAND gate is shown in Figure 1. Figure 2 shows the legend for interpreting certain schematic symbols. After designing the NAND gate, the next module to implement is the inverter.

Table 1. Truth Table for NAND Gate

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

Table 2. myNAND2 Inputs and Outputs

Port Name	Type	Width
i_A	Input	1
i_B	Input	1
o_Z	Output	1

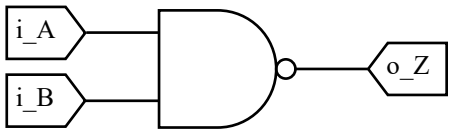

Figure 1. myNAND2

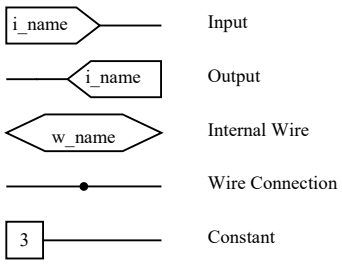

Figure 2. Schematic Legend

2.2 Inverter

The inverter (myINV) is implemented by tying the inputs of a NAND gate together. The truth table for this implementation is shown in Table 3. Note that the B input to the NAND gate is marked in parenthesis as A since the two inputs are tied together. Figure 3 shows how the NAND gate is connected as an inverter and Figure 4 shows schematic symbols for the inverter.

Table 3. Truth Table for NAND Gate Configured as an Inverter

A	B (A)	Z
0	0	1
1	1	0

Table 4. myINV Inputs and Outputs

Port Name	Type	Width
i_A	Input	1
o_Z	Output	1

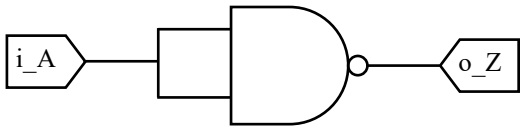

Figure 3. Inverter Schematic

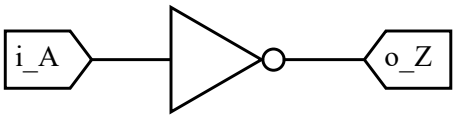

Figure 4. Inverter Symbol

Additionally, an N-bit inverter that inverts all N-bits of an incoming data signal is created by placing N 1-bit inverters in parallel, one per input bit.

Table 5. myINV_N Inputs and Outputs

Port Name	Type	Width
i_A	Input	N
o_Z	Output	N

2.3 Multiplexers

The simplest multiplexer takes two data inputs and a select input and produces a single data output. In the simplest case, the data signals are 1-bit each. If the select input is 0, data input 0 is passed to the output. If the select input is 1, data input 1 is passed to the output. This logic implementing this is:

$$\begin{aligned} Z &= \bar{i_S} i_0 + i_S i_1 \\ Z &= \overline{\overline{\bar{i_S} i_0 + i_S i_1}} \\ Z &= \overline{\overline{\bar{i_S} i_0}\ \overline{i_S i_1}} \\ Z &= NAND(NAND(INV(i_S), i_0), NAND(i_S, i_1)) \end{aligned} \tag{1}$$

Where Z is the output, i_S is the select input, and i_0 and i_1 are data inputs 0 and 1 respectively.

If a four input multiplexer is required, three-two input multiplexers can be used in two stages to produce the desired output. The first stage consists of connecting data input 0 and data input 1 to inputs 0 and 1 of a 2 input multiplexer and also connecting data inputs 2 and 3 to inputs 0 and 1 of another 2 input multiplexer. The select input for both of these first stage multiplexers is the least significant bit of the two bit select signal. The select signal requires two bits because a two bit signal can

take on four values and there are four inputs. Then, the output of the multiplexer with data inputs 0 and 1 are fed to input 0 of the second stage multiplexer. The output of the multiplexer with data inputs 2 and 3 are fed to input 1 of the second stage multiplexer. The select input to the second stage multiplexer is the most significant bit of the two bit select signal. When connected in this configuration, the decimal representation of the select signal corresponds to the input number that is connected to the output. Table 6 shows the truth table for this 4-input multiplexer. In equation form, this is written as shown in Equation 2 where MUX2(i_0, i_1, i_S) indicates a two input multiplexer with data inputs i_0 and i_1 and one bit select signal i_S and MUX4(i_0,i_1,i_2,i_3,i_S) indicates a four input multiplexer with data inputs i_0, i_1, i_2, and i_3 and two bit select signal i_S.

Table 6. Truth Table for 4 Input Multiplexer

i_S(1)	i_S(0)	Z
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

$$\begin{aligned} &Z = MUX4(i_0, i_1, i_2, i_3, i_S) = \cdots \\ &MUX2(MUX2(i_0, i_1, i_S(0)), MUX2(i_2, i_3, i_S(0)), i_S(1)) \end{aligned} \tag{2}$$

An eight input multiplexer is similarly implemented. The eight input multiplexer requires a three bit select signal. The two first stage multiplexers from the four input multiplexer are now replaced by two four input multiplexers whose inputs are data inputs 0 through 3 and data inputs 4 through 7 respectively. The select input to the first stage multiplexers are the two least significant bits of the three bit select signal. The second stage multiplexer is still a two input multiplexer. The second

stage data input 0 comes from the four input multiplexer with data inputs 0 through 3. The second stage data input 1 comes from the four input multiplexer with data inputs 4 through 7. The select signal to the second stage multiplexer is still the most significant bit of the select signal. Continuing to use this pattern as a template results in a 16 input multiplexer having two eight input multiplexers in its first stage and a two input multiplexer in the second stage. A 32 input multiplexer has two 16 input multiplexers for its first stage and still a two input multiplexer in the second stage. Finally, increasing the number of bits in the inputs from one to some arbitrary integer N is a simple as connecting N multiplexers (with the same number of inputs) in parallel. A summary of all the multiplexers is shown in Table 7.

Table 7. All Multiplexers Designed for Basic MIPS Implementation

Name	Number of Data Inputs	Number of Data Bits	Number of Select Bits
MUX2_1	2 (i_0, i_1)	1	1
MUX2_N	2 (i_0, i_1)	N	1
MUX4_1	4 (i_0, i_1, i_2, i_3)	1	2
MUX4_N	4 (i_0, i_1, i_2, i_3)	N	2
MUX8_1	8 (i_0, ..., i_7)	1	3
MUX8_N	8 (i_0, ..., i_7)	N	3
MUX16_1	16 (i_0, ..., i_16)	1	4
MUX16_N	16 (i_0, ..., i_16)	N	4
MUX32_1	32 (i_0, ..., i_31)	1	5
MUX32_N	32 (i_0, ..., i_31)	N	5

Figures 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 show schematics for the multiplexers as well as their schematic symbols. Note that the schematic symbols are used for 1-bit inputs and N-bit inputs interchangeably.

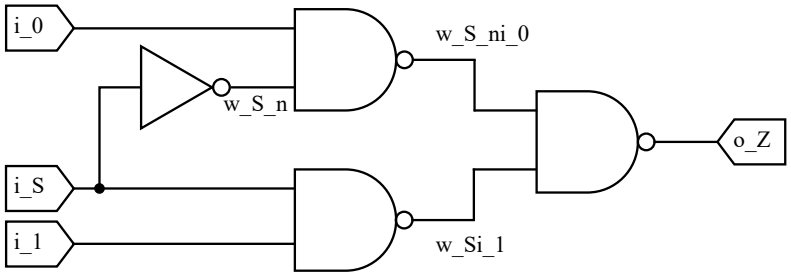

Figure 5. 2-Input Multiplexer Schematic

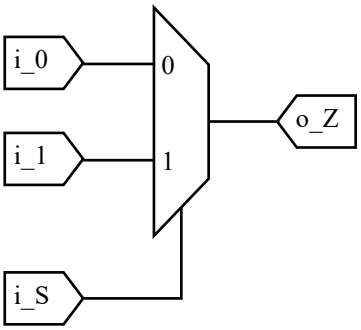

Figure 6. 2-Input Multiplexer Symbol

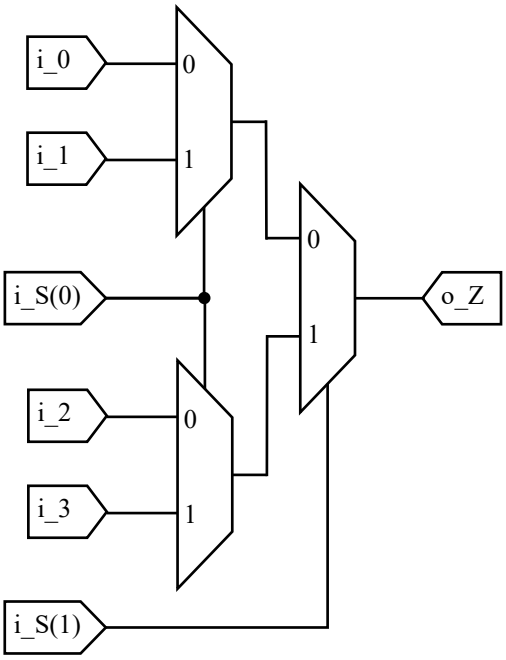

Figure 7. 4-Input Multiplexer Schematic

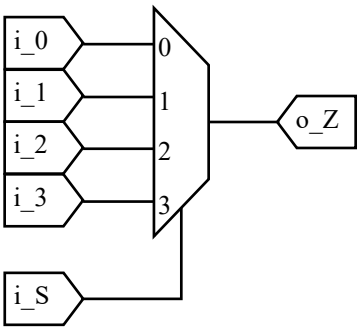

Figure 8. 4-Input Multiplexer Symbol

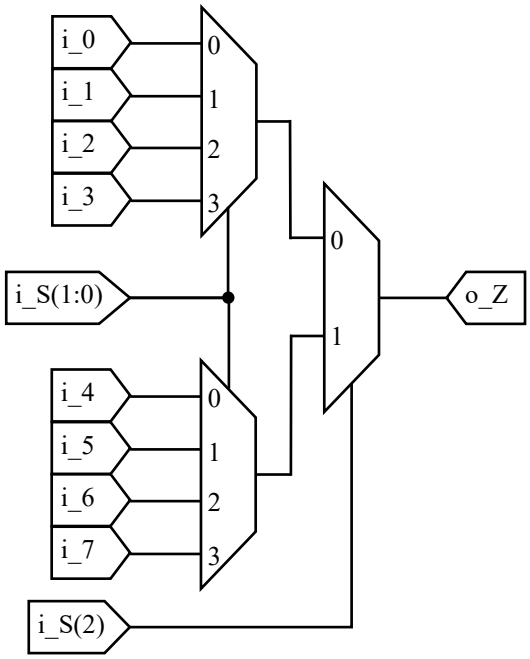

Figure 9. 8-Input Multiplexer Schematic

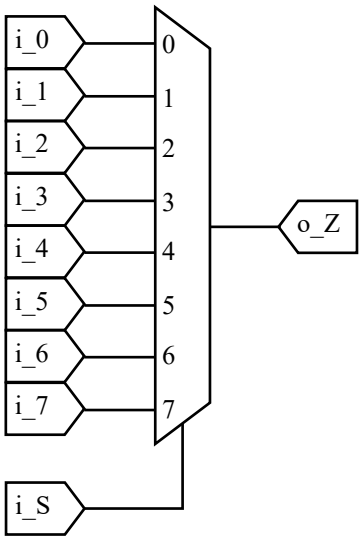

Figure 10. 8-Input Multiplexer Symbol

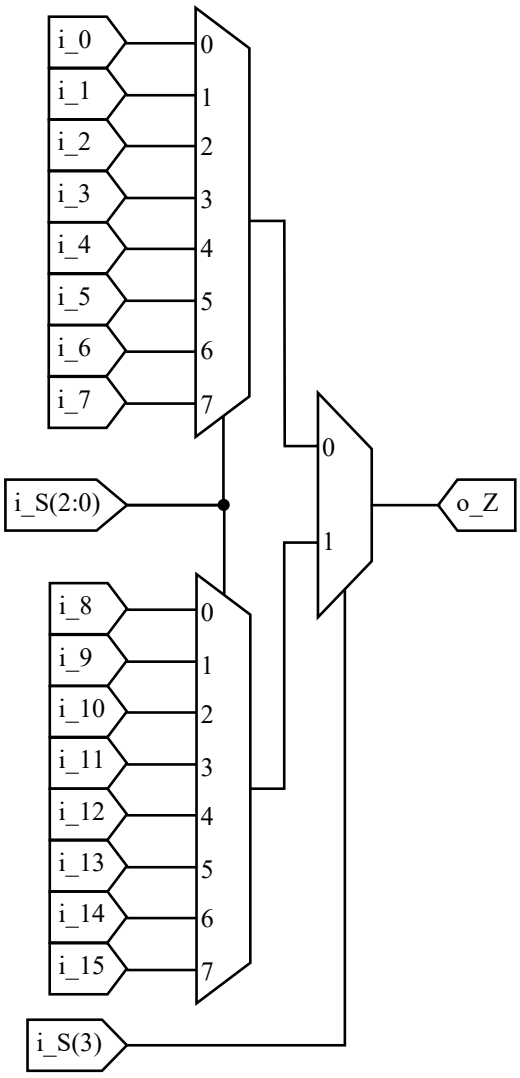

Figure 11. 16-Input Multiplexer Schematic

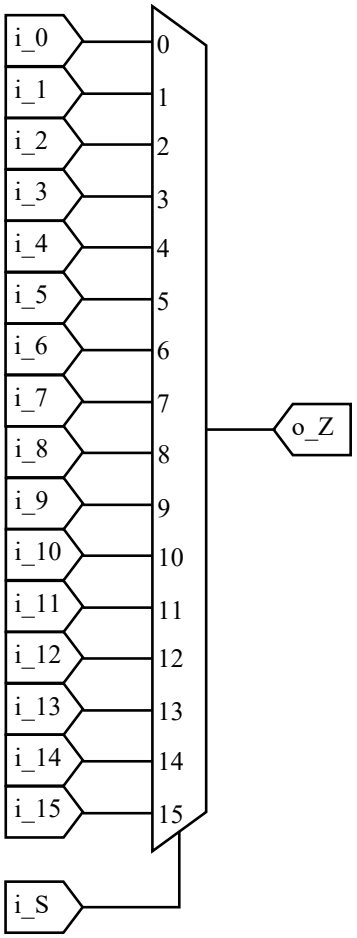

Figure 12. 16-Input Multiplexer Symbol

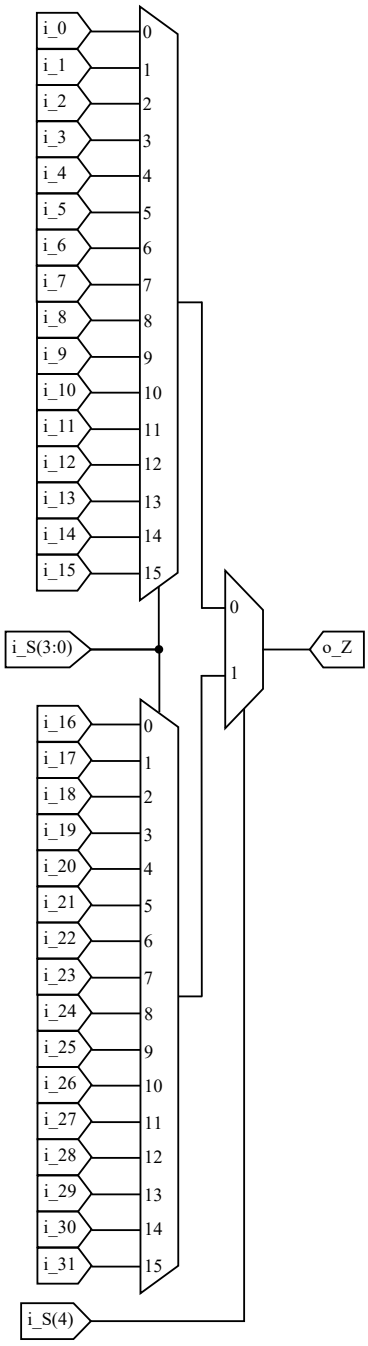

Figure 13. 32-Input Multiplexer Schematic

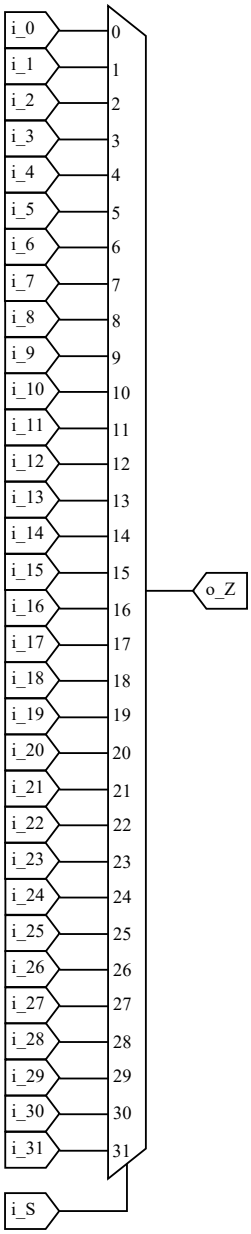

Figure 14. 32-Input Multiplexer Symbol

2.4 Left Shift

The following instructions make use of the Left Shift module:

- SLL
- SLLV

The left shift operation takes a 32 bit input and shifts it left by a specified number of bits between 0 and 31. The most significant bits are thrown away and the new least significant bits are all zeros. The shift amount is a 5 bit input that is interpreted as an unsigned number. For example, if the shift amount is 5, the 5 most significant bits, 31, 30, 29, 28, and 27 are thrown away and the remaining bits are all moved up by 5 bits so that bit 26 is now bit 31 and bit 0 is now bit 5. Bits 4 through 0 are all set equal to 0. The left shift is implemented by having set of 32-32 input multiplexers. Each multiplexer corresponds to a single bit of the output and has the shift amount as its select input. Output bit 31 is generated from a 32 input multiplexer whose data input 0 is bit 31 of the incoming data, data input 1 is bit 30 of the incoming data, and so on until data input 31 is bit 0 of the incoming data. Output bit 30 is generated from another 32 input multiplexer whose data input 0 is bit 30 of the incoming data, data input 1 is bit 29 of the incoming data, and so on until data input 30 is bit 0 of the incoming data and data input 31 is 0. Output bit 29 is generated from another 32 input multiplexer whose data input 0 is bit 29 of the incoming data, data input 1 is bit 28 of the incoming data, and so on until data input 29 is bit 0 of the incoming data and data inputs 30 and 31 are both 0. This pattern continues and output bit 0 is generated from a 32 input multiplexer whose data input 0 is bit 0 of the incoming data and the remaining 31 data inputs are all 0. Figure 15 shows a schematic for the left shift module. Notice that the multiplexer symbols, and their inputs, have been abbreviated.

Table 8. LEFT_SHIFT Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	32	Data
i_SA	Input	5	Shift Amount
o_Z	Output	32	Shifted Input Data

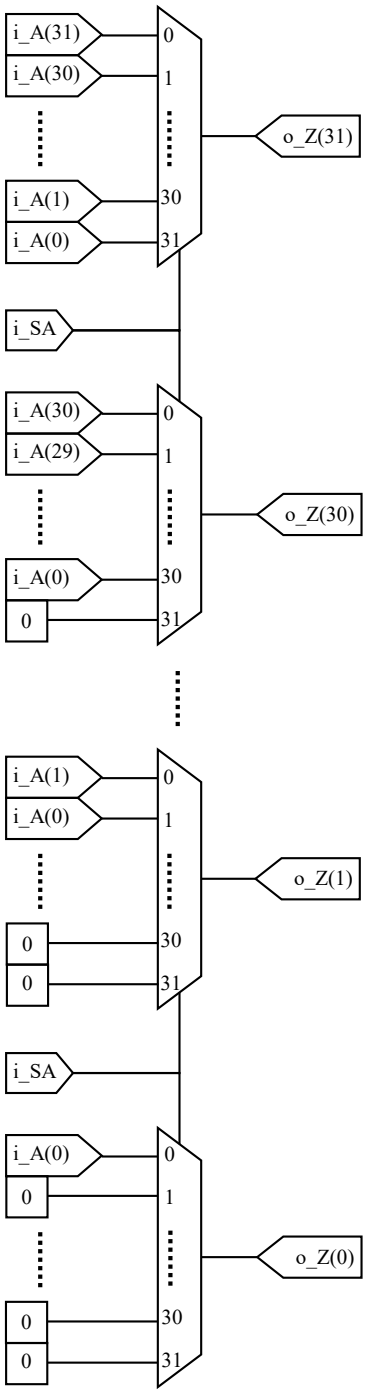

Figure 15. Left Shift Schematic

2.5 Left Shift 2

When there are branch instructions, a 16-bit offset value must be left shifted by 2-bits. Rather than use the main left shift module and incur additional clock cycles, a special purpose 2-bit left shift module is used. This module simply takes in a 32-bit input, which is just the 16-bit offset sign extended to 32-bits, discards bits 30 and 31, and appends to 0's. In other words, input bits 0 through 29 become output bits 2 through 31 and output bits 0 and 1 are set to 0.

Table 9. SLL2_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	32	Data
o_Z	Output	32	Shifted Input Data

2.6 Right Shift

The following instructions make use of the Right Shift module:

- SRL
- SRA
- SRLV
- SRAV

The right shift operation takes a 32 bit input and shifts it right by a specified number of bits between 0 and 31. The least significant bits are thrown away; however, the new most significant bits depend upon another input to the right shift operation. This input determines whether the right shift should be a logical right shift or an arithmetic right shift. The logical right shift makes the new most significant bits all zero. The arithmetic right shift assumes the data input is a signed number and the sign bit, which is the most significant bit of the data input, is then copied to the new most significant bits after the shift. A MUX2_1 takes in a 0 on its data 0 input and

the most significant bit of the right shift data input as its data 1 input. It's select signal is an input that is 0 if a logical right shift is to be performed and a 1 if an arithmetic right shift is to be performed. The output of this multiplexer is the value used for the new most significant bits in the right shifted output.

Similar to the Left Shift operation, a series of 32-32 input multiplexers are used, one for each output bit. Output bit 31 is generated from a 32 input multiplexer whose data input 0 is bit 31 of the incoming data, and the remaining data inputs are all set equal to the new most significant bit value. Output bit 30 is generated from a 32 input multiplexer whose data input 0 is bit 30 of the incoming data, data input 1 is bit 31 of the incoming data, and the remaining data inputs are all set equal to the new most significant bit value. This pattern continues. Output bit 1 is generated from a 32 input multiplexer whose data input 0 is bit 1 of the incoming data, data input 1 is bit 2 of the incoming data, and so on until data input 30 is bit 31 of the incoming data and data input 31 is the new most significant bit. Output bit 0 is generated from a 32 input multiplexer whose data input 0 is bit 0 of the incoming data and so on until data input 31 is bit 31 of the incoming data. Figure 15 shows a schematic for the left shift module. Notice that the multiplexer symbols, and their inputs, have been abbreviated.

Table 10. RIGHT_SHIFT Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	32	Data
i_SA	Input	5	Shift Amount
i_ARITH	Input	1	1 if arithmetic shift, 0 if logical shift
o_Z	Output	32	Shifted Input Data

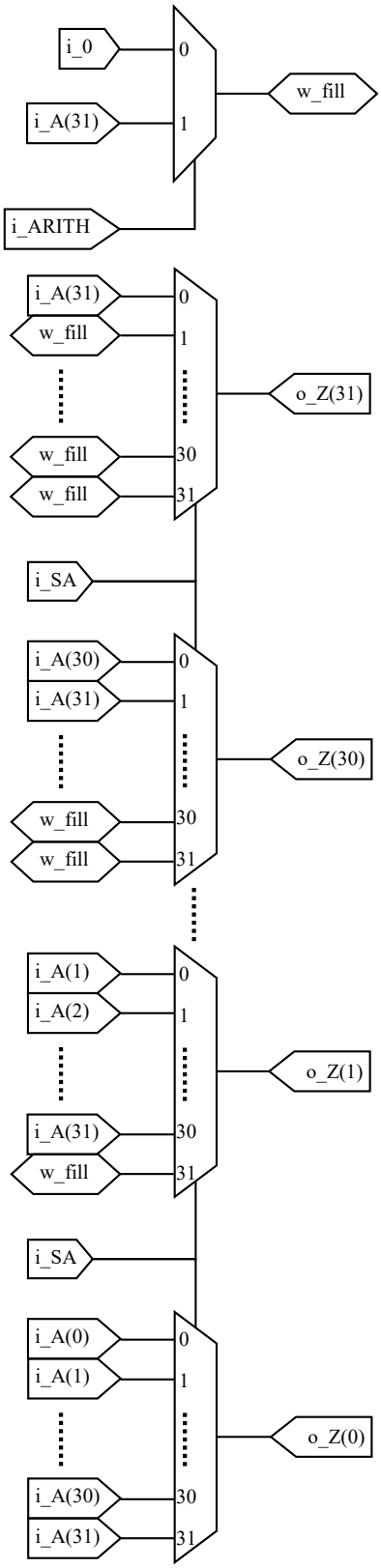

Figure 16. Right Shift Schematic

2.7 Bitwise Operations

The following instructions make use of the Bitwise Operations:

- AND
- OR
- XOR
- NOR
- ANDI
- ORI
- XORI

All of the bitwise operations are implemented as a single block (Bitwise_Ops2) due to synergies in the design. A few bitwise operation outputs not included in the selected set of MIPS instructions were added so that these signals could be reused in other blocks to reduce hardware redundancy. It turns out that some of these bitwise operations can be reused in the adder as well as overflow detection for signed addition and subtraction and also in comparisons between two values. The inputs to Bitwise_Ops2 are A and B and the outputs are $\bar{A}$, $\bar{B}$, NAND(A,B), AND(A,B), OR(A,B), NOR(A,B), XNOR(A,B), and XOR(A,B). The first function implemented in the single bit bitwise operations on two inputs (Bitwise_Ops2) was the AND function. Using the NAND gate and Inverter previously defined, this function is implemented according to:

$$AND = AB = \overline{\overline{AB}} = INV(NAND(A, B)) \tag{3}$$

Notice the use of the line over the text to denote an inverted value, a double line over text to indicate a value that is inverted twice, and the use of INV(X) as a way of inverting signal X and NAND(X,Y) as a way of performing a NAND on signals X and Y. The next function implemented was the OR function. This is implemented according to:

$$OR = A + B = \overline{\overline{A + B}} = \overline{\bar{A}\bar{B}} = NAND(INV(A), INV(B)) \tag{4}$$

Also notice that the intermediate signals $\bar{A}$ and $\bar{B}$ used to compute OR are also output for use elsewhere to minimize redundant hardware in other blocks that need these negated signals. To implement the NOR function, the output of the OR function is inverted:

$$NOR = \overline{A + B} = INV(NAND(INV(A), INV(B))) = INV(OR(A, B)) \tag{5}$$

Finally, the XOR function is defined as:

$$\begin{aligned}
XOR &= A\bar{B} + \bar{A}B \\
XOR &= \overline{\overline{A\bar{B} + \bar{A}B}} \\
XOR &= \overline{(\bar{A} + B)(A + \bar{B})} \\
XOR &= \overline{A\bar{A} + \bar{A}\bar{B} + AB + B\bar{B}} \\
XOR &= \overline{AB + \bar{A}\bar{B}} \\
XOR &= \overline{AB}\ \overline{\bar{A}\bar{B}} \\
XOR &= \overline{AB}(A + B)
\end{aligned} \tag{6}$$

This last step in Equation 6 suggests the outputs of the NAND and OR functions should be combined using another AND function. This is implemented as:

$$\begin{aligned}
XOR &= \overline{AB}(A + B) \\
XOR &= INV(NAND(NAND(A, B), OR(A, B)))
\end{aligned} \tag{7}$$

Buried in this equation is the XNOR function. Notice that the XOR function is an inversion of the XNOR function which is simply:

$$
\begin{aligned}
XNOR &= \overline{\overline{AB}(A+B)} \\
XNOR &= INV(INV(NAND(NAND(A,B),OR(A,B)))) \\
XNOR &= NAND(NAND(A,B),OR(A,B))
\end{aligned} \tag{8}
$$

Table 11. Bitwise_Ops2 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	1	Input A
i_B	Input	1	Input B
o_A_n	Output	1	INV(A)
o_B_n	Output	1	INV(B)
o_AND	Output	1	AND(A,B)
o_NAND	Output	1	NAND(A,B)
o_OR	Output	1	OR(A,B)
o_XOR	Output	1	XOR(A,B)
o_XNOR	Output	1	XNOR(A,B)
o_NOR	Output	1	NOR(A,B)

Figure 17 shows a schematic for how the bitwise operations are determined.

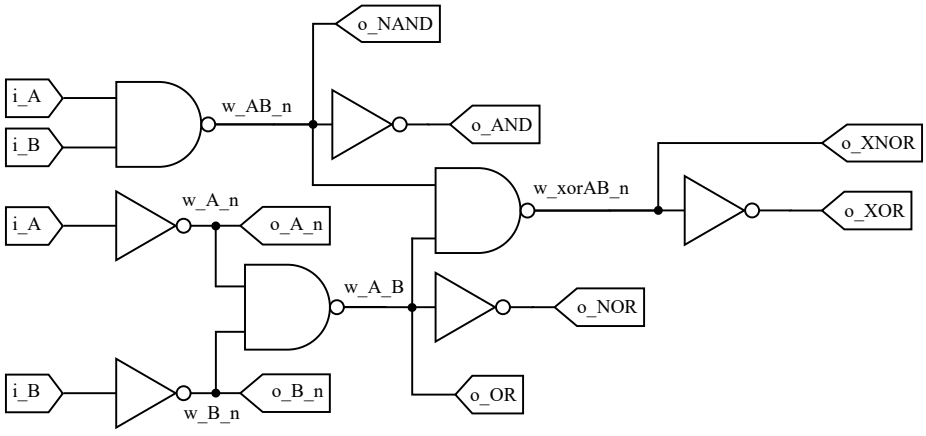

Figure 17. Bitwise_Ops2 Schematic

Then, to expand the Bitwise_Ops2 to 32-bits, 32 of the Bitwise_Ops2 modules are connected in parallel, one for each of 32 bits, so that the inputs and outputs form 32-bit buses.

Table 12. Bitwise_Ops2_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	32	Input A
i_B	Input	32	Input B
o_A_n	Output	32	INV(A)
o_B_n	Output	32	INV(B)
o_AND	Output	32	AND(A,B)
o_NAND	Output	32	NAND(A,B)
o_OR	Output	32	OR(A,B)
o_XOR	Output	32	XOR(A,B)
o_XNOR	Output	32	XNOR(A,B)
o_NOR	Output	32	NOR(A,B)

2.8 Adder

The following instructions make use of the Adder module:

- ADD
- ADDU
- SUB
- SUBU
- SLT
- SLTU
- BGEZ
- BLTZ
- BEQ
- BNE
- BLEZ
- BGTZ

- ADDI
- ADDIU
- SLTI
- SLTIU
- LW
- SW

The basic unit of the adder is the full adder. It performs single bit addition on three inputs and produces two outputs. the inputs are i_A, i_B, and a carry-in i_C. The outputs are the sum o_S and carry out o_C. The truth table for this operation is shown here:

Table 13. Full Adder Truth Table

Inputs			Outputs	
i_A	i_B	i_C	o_S	o_C
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

The logic for the sum output o_S is:

$$
\begin{aligned}
o_S &= \overline{i_A} i_B \overline{i_C} + i_A \overline{i_B}\ \overline{i_C} + \overline{i_A}\ \overline{i_B} i_C + i_A i_B i_C \\
o_S &= \overline{\overline{i_A B \overline{i_C}} + A \overline{i_B}\ \overline{i_C} + \overline{i_A}\ \overline{i_B} i_C + i_A i_B i_C} \\
o_S &= \overline{(i_A + \overline{i_B} + i_C)(\overline{i_A} + i_B + i_C)(i_A + i_B + \overline{i_C})(\overline{i_A} + \overline{i_B} + \overline{i_C})} \\
o_S &= \overline{(\overline{\overline{i_A} i_B} + i_C)(i_A \overline{\overline{i_B}} + i_C)(i_A + i_B + \overline{i_C})(\overline{i_A i_B} + \overline{i_C})}
\end{aligned}
\tag{9}
$$

At this point, note that the third term contains OR(A,B) and the fourth term contains NAND(A,B). Two new signals are also defined before continuing with determining the sum output.

$$w_add_05 = \overline{\overline{i_A} i_B} = NAND(INV(i_A), i_B) \tag{10}$$

$$w_add_06 = \overline{i_A \overline{i_B}} = NAND(i_A, INV(i_B)) \tag{11}$$

$$\begin{aligned}
o_S &= \overline{(\overline{\overline{i_A} i_B} + i_C)(\overline{i_A \overline{i_B}} + i_C)(i_A + i_B + \overline{i_C})(\overline{i_A i_B} + \overline{i_C})} \\
o_S &= \overline{(w_add_05 + i_C)(w_add_06 + i_C)} \\
&\quad \overline{(OR(i_A, i_B) + \overline{i_C})(NAND(i_A, i_B) + \overline{i_C})} \\
w_add_07 &= \overline{\overline{OR(i_A, i_B)} i_C} \\
w_add_07 &= \overline{NOR(i_A, i_B) i_C} = NAND(NOR(i_A, i_B), i_C) \\
w_add_08 &= \overline{\overline{w_add_05}\ \overline{C}} = NAND(INV(w_add_05), INV(C)) \\
w_add_09 &= \overline{\overline{w_add_06}\ \overline{C}} = NAND(INV(w_add_06), INV(C)) \\
w_add_10 &= \overline{\overline{NAND(i_A, i_B)} i_C} \\
w_add_10 &= \overline{AND(i_A, i_B) i_C} = NAND(AND(i_A, i_B), i_C) \\
o_S &= \overline{w_add_07 w_add_08 w_add_09 w_add_10} \\
w_add_11 &= \overline{w_add_7 w_add_8} = NAND(w_add_7, w_add_8) \\
w_add_12 &= \overline{w_add_9 w_add_10} = NAND(w_add_9, w_add_10) \\
o_S &= \overline{\overline{w_add_11}\ \overline{w_add_12}} = NAND(INV(w_add_11), INV(w_add_12))
\end{aligned} \tag{12}$$

Notice that in these equations the AND(A,B) and NOR(A,B) outputs from the bitwise operations are reused rather than computing these again. The logic for the carry output o_C is:

$$
\begin{aligned}
o_C &= i_Ai_B + i_Ai_C + i_Bi_C \\
o_C &= \overline{\overline{i_Ai_B + i_Ai_C + i_Bi_C}} \\
o_C &= \overline{(\overline{i_Ai_B})(\overline{i_Ai_C})(\overline{i_Bi_C})} \\
w_add_01 &= NAND(i_A, i_C) \\
w_add_02 &= NAND(i_B, i_C) \\
o_C &= \overline{NAND(i_A, i_B)w_add_01w_add_02} \\
w_add_03 &= NAND(NAND(i_A, i_B), w_add_01) \\
o_C &= \overline{\overline{w_add_03}w_add_02} \\
o_C &= NAND(\overline{w_add_03}, w_add_02)
\end{aligned}
\tag{13}
$$

Notice that in these equations the NAND(A,B) output from the bitwise operations is reused rather than computing it again. The reuse of the AND(A,B), NOR(A,B), and NAND(A,B) from the output of the bitwise operations is possible because the bitwise operations have the same A and B inputs as the adder in the Arithmetic Logic Unit (ALU). This will become evident later when the ALU Core is discussed.

Table 14. FULL_ADDER_1 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	1	Input A
i_A_n	Input	1	INV(A)
i_B	Input	1	Input B
i_B_n	Input	1	INV(B)
i_AB_AND	Input	1	AND(A,B)
i_AB_NAND	Input	1	NAND(A,B)
i_AB_NOR	Input	1	NOR(A,B)
i_C	Input	1	Carry In
o_S	Output	1	Sum
o_C	Output	1	Carry Out

Rather than chaining 32 1-bit full adders in series to construct a ripple carry

adder, a carry select adder implementation is used instead for the sake of speed. The 32-bit carry select adder is built upon eight 4-bit carry select adders in series. Each 4-bit carry select adder consists of two 4-bit full adders. One of the two 4-bit full adders is provided a carry in of 0 while the other is provided a carry in of 1. When the carry in to the 4-bit carry select adder is available, it selects which of the sums to output based on the value of the carry in. The carry in also triggers logic which determines the carry out of the 4-bit carry select adder. This will be discussed in more detail after first discussing the 4-bit full adder.

The 4-bit full adder is implemented by placing four 1-bit full adders in series to that the carry out of the 1-bit full adder associated with the least significant bit of the 4-bit input (bit 0) is connected to the carry in of the 1-bit full adder associated with bit 1 of the 4-bit input. The carry out from the bit 1 adder is the input to the bit 2 adder. The carry out from the bit 2 adder is the input to the bit 3 adder. The carry out of the bit 3 adder is the carry out for the 4-bit full adder. The carry in to the 4-bit full adder is the carry in to the bit 0 adder.

Table 15. FULL_ADDER_4 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	4	Input A
i_A_n	Input	4	INV(A)
i_B	Input	4	Input B
i_B_n	Input	4	INV(B)
i_AB_AND	Input	4	AND(A,B)
i_AB_NAND	Input	4	NAND(A,B)
i_AB_NOR	Input	4	NOR(A,B)
i_C	Input	1	Carry In
o_S	Output	4	Sum
o_C	Output	1	Carry Out

The 4-bit carry select adder is built directly upon the 4-bit full adder. Two 4-bit

full adders operate in parallel. One has a carry in of 0 and the other has a carry in of 1. In this fashion, the computation of the sum does not depend on the carry in rippling through all four 1-bit full adders. A 2-input 4-bit multiplexer has data input 0 equal to the sum from the 4-bit full adder with carry in 0 and data input 1 equal to the sum from the 4-bit full adder with carry in 1 and the select input connected to the carry in. As soon as the carry in arrives, the correct sum is immediately output. Additionally, the carry out is determined by logic connected to the carry out of both 4-bit adders. Assigning the carry out from the 4-bit full adder with carry in 0 the name w_C0 and the carry out from the 4-bit full adder with carry in 1 the name w_C1, this logic is:

$$
\begin{aligned}
o_C &= i_Cw_C1 + w_C0 \\
o_C &= \overline{\overline{i_Cw_C1 + w_C0}} \\
o_C &= \overline{(\overline{i_Cw_C1})\overline{w_C0}} \\
w_C11 &= NAND(i_C, w_C1) \\
o_C &= \overline{w_C11\overline{w_C0}} \\
o_C &= NAND(w_C11, INV(w_C0))
\end{aligned}
\tag{14}
$$

Table 16 shows the inputs and outputs of the 4-bit carry select adder. Table 17 clarifies how signals are routed between modules withing the 4-bit carry select adder.

Table 16. CARRY_SELECT_ADDER_4 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	4	Input A
i_A_n	Input	4	INV(A)
i_B	Input	4	Input B
i_B_n	Input	4	INV(B)
i_AB_AND	Input	4	AND(A,B)
i_AB_NAND	Input	4	NAND(A,B)
i_AB_NOR	Input	4	NOR(A,B)
i_C	Input	1	Carry In
o_S	Output	4	Sum
o_C	Output	1	Carry Out

Table 17. CARRY_SELECT_ADDER_4 Connections

4-bit Full Adder 0			
I/O	Type	Width	Connection
i_A	Input	4	i_A
i_A_n	Input	4	i_A_n
i_B	Input	4	i_B
i_B_n	Input	4	i_B_n
i_AB_AND	Input	4	i_AB_AND
i_AB_NAND	Input	4	i_AB_NAND
i_AB_NOR	Input	4	i_AB_NOR
i_C	Input	1	0
o_S	Output	4	w_S0
o_C	Output	1	w_C0
4-bit Full Adder 1			
I/O	Type	Width	Connection
i_A	Input	4	i_A
i_A_n	Input	4	i_A_n
i_B	Input	4	i_B
i_B_n	Input	4	i_B_n
i_AB_AND	Input	4	i_AB_AND
i_AB_NAND	Input	4	i_AB_NAND
i_AB_NOR	Input	4	i_AB_NOR
i_C	Input	1	1

Table 17 – *Continued on next page*

Table 17 – *Continued from previous page*

o_S	Output	4	w_S1
o_C	Output	1	w_C1
Sum 4-bit Multiplexer			
i_0	Input	4	w_S0
i_1	Input	4	w_S1
i_S	Input	4	i_C
o_Z	Output	4	o_S
w_C0 Inverter			
i_A	Input	1	w_C0
o_Z	Output	1	w_C0_n
w_C11 NAND			
i_A	Input	1	i_C
i_B	Input	1	w_C1
o_Z	Output	1	w_C11
Carry Out NAND			
i_A	Input	1	w_C11
i_B	Input	1	w_C0_n
o_Z	Output	1	o_C

The final step in implementing the 32-bit adder was to connect eight 4-bit carry select adders in series the same way the four 1-bit full adders were connected in series to create a 4-bit full adder. In this case, the carry in to the 32-bit adder is input to the bit 0 to bit 3, 4-bit carry select adder. The carry out from this 4-bit carry select adder is the input to the bit 4 to bit 7, 4-bit carry select adder. This continues until the output of the 28 to 31, 4-bit carry select adder is made the carry out of the 32-bit adder. Note that the 4-bit and 32-bit carry select adder design comes from "CMOS VLSI Design: A Circuits and Systems Perspective Fourth Editon" by Neil Weste and David Harris and can be found on page 445 in Figure 11.24 [3]. The 32-bit carry select adder is an adaptation of the 16-bit carry select adder presented by Weste and Harris [3].

Table 18. CARRY_SELECT_ADDER_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	32	Input A
i_A_n	Input	32	INV(A)
i_B	Input	32	Input B
i_B_n	Input	32	INV(B)
i_AB_AND	Input	32	AND(A,B)
i_AB_NAND	Input	32	NAND(A,B)
i_AB_NOR	Input	32	NOR(A,B)
i_C	Input	1	Carry In
o_S	Output	32	Sum
o_C	Output	1	Carry Out

This adder implementation that makes use of the AND(A,B), NAND(A,B) and OR(A,B) outputs from Bitwise_Ops32 works great when the inputs to the adder are the same as the inputs to Bitwise_Ops32, however it does not work for the adder that needs to be used for the program counter. With the exception of branch instructions, the program counter must be incremented by four at the completion of every instruction. This could be done using the ALU, however it would add extra clock cycles to the end of every instruction. Instead, a separate, self contained adder is used. However, since the program counter is always incremented by four, only a 30 bit adder is needed to add 1 to bits 2 through 31 of the program counter. The two least significant bits are unaffected by adding four to the program counter. The first difference with this self contained adder is that it must compute the AND(A,B), NAND(A,B) and OR(A,B) operations internally. These are computed in the exact same way they were computed in the Bitwise_Ops2 component. Secondly, instead of using eight 4-bit carry select adders in series to compute a 32-bit sum, seven 4-bit adders are used to compute the sum for bits 2 through 29 and a 2-bit carry select adder is used to add bits 30 and 31. The 2-bit carry select adder is identical to the 4-bit carry select adder. The only difference is that the 2-bit carry select adder uses

two 2-bit full adders rather than two 4-bit full adders. The logic for deciding which adder's sum should be output is the same and so is the logic for computing the carry out. The 2-bit full adder is identical to the 4-bit full adder, except it only has two 1-bit full adders. The pin-outs for the self-contained full adders and self-contained carry select adders are shown in the following tables.

Table 19. FULL_ADDER_1_SC (Self-Contained) Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	1	Input A
i_B	Input	1	Input B
i_C	Input	1	Carry In
o_S	Output	1	Sum
o_C	Output	1	Carry Out

Table 20. FULL_ADDER_2_SC Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	2	Input A
i_B	Input	2	Input B
i_C	Input	1	Carry In
o_S	Output	2	Sum
o_C	Output	1	Carry Out

Table 21. FULL_ADDER_4_SC Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	4	Input A
i_B	Input	4	Input B
i_C	Input	1	Carry In
o_S	Output	4	Sum
o_C	Output	1	Carry Out

Table 22. CARRY_SELECT_ADDER_2_SC Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	2	Input A
i_B	Input	2	Input B
i_C	Input	1	Carry In
o_S	Output	2	Sum
o_C	Output	1	Carry Out

Table 23. CARRY_SELECT_ADDER_4_SC Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	4	Input A
i_B	Input	4	Input B
i_C	Input	1	Carry In
o_S	Output	4	Sum
o_C	Output	1	Carry Out

Table 24. CARRY_SELECT_ADDER_30_SC Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	30	Input A
i_B	Input	30	Input B
i_C	Input	1	Carry In
o_S	Output	30	Sum
o_C	Output	1	Carry Out

2.9 Load Upper Immediate

The following instructions make use of the Load Upper Immediate module:

- LUI

The load upper immediate module is very simple. An immediate value entering the ALU Core is sign extended to 32 bits, but the most significant 16 bits are ignored by the LUI module and only the 16 least significant bits are input to the LUI module. The LUI module places these bits in the most significant 16 bits of the output. The least significant 16 bits of the output are set equal to 0.

Table 25. LUI Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	16	Input A
o_Z	Output	32	Output

2.10 Overflow Detection

The following instructions make use of the Overflow Detection module:

- ADD
- SUB
- ADDI

Overflow detection is used for signed addition and subtraction functions. A subtraction is just an addition where the B input has been inverted prior to arriving at the adder and the carry-in to the adder has been set to 1 instead of 0. If two positive numbers are added and the result is a negative number or two negative numbers are added and the result is a positive number, an overflow has occurred. When an overflow occurs, the data being sent to the destination register should not be the result of the addition, but should be the current value of the destination register. For ADD and SUB instructions, the destination register is the RD register specified in the instruction. For the ADDI instruction, the destination register is the RT register specified in the instruction. A 2-bit control signal named i_OVER_CTRL takes on the value 00 if the instruction is not ADD, SUB, or ADDI. It is 01 for ADD, 10 for

SUB, and 11 for ADDI. This control signal is the input to a 32-bit 4 input multiplexer whose data 0 input is the output of the adder, data 1 and data 2 inputs are the current RD register data, and data 3 input is the current RT register data. However, it has not yet been determined whether an overflow has actually occurred. The output of this multiplexer only depends on a control signal issued by the Controller. A second multiplexer (32-bit 2 inputs) selects between the output of the adder on its data 0 input and the output of the first multiplexer. The select signal for this multiplexer is determined by combinational logic that determines if an overflow has occurred. In this fashion, if the instruction is ADD or SUB and an overflow occurs, the output will be the current RD register data. If no overflow occurs, the output will be the result from the adder. If the instruction is ADDI and an overflow occurs, the output will be the current RT register data. If no overflow occurs, the output will be the result from the adder. If the instruction is not ADD, SUB, or ADDI, the output will be the result from the adder whether an overflow occurs or not. The following equation shows the logic for determining when an overflow has occurred.

$$
\begin{aligned}
w_OV &= \overline{i_A}\ \overline{i_B}\ i_S + i_A\ i_B\ \overline{i_S} \\
w_OV &= \overline{\overline{i_A}\ \overline{i_B}\ i_S + i_A\ i_B\ \overline{i_S}} \\
w_OV &= \overline{(i_A + i_B + \overline{i_S})(\overline{i_A} + \overline{i_B} + i_S)} \\
w_OV &= \overline{i_A\ \overline{i_A} + i_A\ \overline{i_B} + i_A\ i_S + \overline{i_A}\ i_B + i_B\ \overline{i_B} + \cdots} \\
&\quad \overline{i_B\ i_S + \overline{i_A}\ \overline{i_S} + \overline{i_B}\ \overline{i_S} + i_S\ \overline{i_S}} \\
w_OV &= \overline{i_S\ (i_A + i_B) + \overline{i_S}\ (\overline{i_A} + \overline{i_B}) + i_A\ \overline{i_B} + \overline{i_A}\ i_B} \\
w_OV &= \overline{i_S\ OR(i_A, i_B) + \overline{i_S}\ NAND(\overline{i_A}, \overline{i_B}) + XOR(i_A, i_B)} \\
w_OV_0 &= NAND(i_S, OR(i_A, i_B)) \\
w_OV_1 &= NAND(INV(i_S), NAND(i_A, i_B)) \\
w_OV &= \overline{\overline{w_OV_0} + \overline{w_OV_1} + XOR(i_A, i_B)} \\
w_OV_2 &= NAND(w_OV_0, w_OV_1) \\
w_OV &= \overline{w_OV_2 + XOR(i_A, i_B)} \\
w_OV_3 &= NAND(INV(w_OV_2), INV(XOR(i_A, i_B))) \\
w_OV_3 &= NAND(INV(w_OV_2), XNOR(i_A, i_B)) \\
w_OV &= \overline{w_OV_3}
\end{aligned}
\tag{15}
$$

Table 26. Over_Detect Inputs and Outputs

Port Name	Type	Width	Description
i_AB_NAND	Input	1	NAND(A(31),B(31))
i_AB_XNOR	Input	1	XNOR(A(31),B(31))
i_S	Input	32	Output of Adder
i_RD	Input	32	Current Data in RD Register
i_RT	Input	32	Current Data in RT Register
i_OVER_CTRL	Input	2	Overflow Control Signal
o_Z	Output	32	Output

2.11 Comparator

The following instructions make use of the Comparator module:

- SLT
- SLTU
- BGEZ
- BLTZ
- BEQ
- BNE
- BLEZ
- BGTZ
- SLTI
- SLTIU

The comparator's function is to determine how the A input to the ALU compares to the B input to the ALU. For this purpose, B is negated making it negative and B is subtracted from A to create an output that can be compared to zero. A control signal determines whether the output should be the result of the greater than or equal to, less than, equal to, not equal to, less than or equal to, or greater than comparison. This may seem straight forward, but there are a few catches. First, if A and B are signed inputs, an overflow may occur if A is positive and B is negative before it is negated making it positive. An overflow may also occur if A is negative and B is positive before it is negated. In these cases, the adder is either adding two positive numbers or two negative numbers and an overflow may occur. If an overflow is possible, the comparator determines that A is greater than B if the sign bit for A and B are both 0 after B has been negated. Also, the comparator determines that A is less than B if the sign bit for A and B are both 1 after B has been negated. If no overflow is possible, such as when A and B have the same sign before B is negated, the result of the sum is compared to zero.

In the case where two unsigned numbers are being compared, an overflow can still occur when both A and B have the same most significant bit after B is negated. This is actually the same thing as for the signed bits, but the comparison results are different. If A's most significant bit is a 1 and B's most significant bit is a 1 after negation, then A is greater than B. If A's most significant bit is a 0 and B's most significant bit is a 0 after negation, A is less than B. If the most significant bits differ after B is negated, the result of the sum is compared to zero to determine how A compares to B. In this last case, the sign bit of the sum will determine whether A is greater than or less than B.

The final piece complicating matters is that an overflow may only occur for SLT, SLTU, SLTI, and SLTIU instructions, so a control signal named i_overflow is used as an input to toggle the overflow detection on for SLT, SLTU, SLTI, and SLTIU instructions and off for BLTZ instructions.

The following logic shows how the greater than or equal to zero comparison is computed. If the sum sign bit is 0, then A is greater than or equal to B. Otherwise, A is not greater than or equal to B.

$$w_GEZ = INV(i_S(31)) \tag{16}$$

Assuming no overflow can occur, the following logic shows how the less than zero comparison is computed. If the sum sign bit is 1, then A is less than B. Otherwise, A is not less than B.

$$w_LTZ0 = i_S(31) \tag{17}$$

The following logic shows how the equal to zero comparison is computed. If all 32 sum bits are 0, then A equals B. Otherwise, A does not equal B.

$$w_EQ = INV(OR(i_S)) \tag{18}$$

Because a 32-bit OR is not used anywhere else in the Basic MIPS architecture, it is implemented within the comparator module. The logic for this is shown here. Notice that the output is being called w_NE for not equal. In fact, if the result of the 32-bit OR is 1, then A and B are not equal. If it is 0, then A and B are equal.

$$\begin{aligned}
w_NE = {} & i_S(31) + i_S(30) + i_S(29) + i_S(28) + i_S(27) + i_S(26) + \cdots \\
& _S(25) + i_S(24) + i_S(23) + i_S(22) + i_S(21) + \cdots \\
& i_S(20) + i_S(19) + i_S(18) + i_S(27) + i_S(16) + \cdots \\
& i_S(15) + i_S(14) + i_S(13) + i_S(12) + i_S(11) + \cdots \\
& i_S(10) + i_S(9) + i_S(8) + i_S(7) + i_S(6) + \cdots \\
& i_S(5) + i_S(4) + i_S(3) + i_S(2) + i_S(1) + i_S(0) \\
o_NE = {} & \overline{\overline{i_S(31) + i_S(30) + i_S(29) + i_S(28) + i_S(27) + i_S(26) + \cdots}} \\
& \overline{\overline{i_S(25) + i_S(24) + i_S(23) + i_S(22) + i_S(21) + i_S(20) + \cdots}} \\
& \overline{\overline{i_S(19) + i_S(18) + i_S(27) + i_S(16) + i_S(15) + i_S(14) + \cdots}} \\
& \overline{\overline{i_S(13) + i_S(12) + i_S(11) + i_S(10) + i_S(9) + i_S(8) + \cdots}} \\
& \overline{\overline{i_S(7) + i_S(6) + i_S(5) + i_S(4) + i_S(3) + i_S(2) + \cdots}} \\
& \overline{\overline{i_S(1) + i_S(0)}} \\
w_NE = {} & \overline{\overline{i_S(31)}\ \overline{i_S(30)}\ \overline{i_S(29)}\ \overline{i_S(28)}\ \overline{i_S(27)}\ \overline{i_S(26)}\ \overline{i_S(25)}\ \overline{i_S(24)}} \\
& \overline{\overline{i_S(23)}\ \overline{i_S(22)}\ \overline{i_S(21)}\ \overline{i_S(20)}\ \overline{i_S(19)}\ \overline{i_S(18)}\ \overline{i_S(27)}\ \overline{i_S(16)}} \\
& \overline{\overline{i_S(15)}\ \overline{i_S(14)}\ \overline{i_S(13)}\ \overline{i_S(12)}\ \overline{i_S(11)}\ \overline{i_S(10)}\ \overline{i_S(9)}\ \overline{i_S(8)}} \\
& \overline{\overline{i_S(7)}\ \overline{i_S(6)}\ \overline{i_S(5)}\ \overline{i_S(4)}\ \overline{i_S(3)}\ \overline{i_S(2)}\ \overline{i_S(1)}\ \overline{i_S(0)}}
\end{aligned} \tag{19}$$

$$
\begin{aligned}
w_S31_30 &= NAND(INV(i_S(31), INV(i_S(30))) \\
w_S29_28 &= NAND(INV(i_S(29), INV(i_S(28))) \\
w_S27_26 &= NAND(INV(i_S(27), INV(i_S(26))) \\
w_S25_24 &= NAND(INV(i_S(25), INV(i_S(24))) \\
w_S23_22 &= NAND(INV(i_S(22), INV(i_S(22))) \\
w_S21_20 &= NAND(INV(i_S(21), INV(i_S(20))) \\
w_S19_18 &= NAND(INV(i_S(19), INV(i_S(18))) \\
w_S17_16 &= NAND(INV(i_S(17), INV(i_S(16))) \\
w_S15_14 &= NAND(INV(i_S(15), INV(i_S(14))) \\
w_S13_12 &= NAND(INV(i_S(12), INV(i_S(12))) \\
w_S11_10 &= NAND(INV(i_S(11), INV(i_S(10))) \\
w_S09_08 &= NAND(INV(i_S(9), INV(i_S(8))) \\
w_S07_06 &= NAND(INV(i_S(7), INV(i_S(6))) \\
w_S05_04 &= NAND(INV(i_S(5), INV(i_S(4))) \\
w_S03_02 &= NAND(INV(i_S(2), INV(i_S(2))) \\
w_S01_00 &= NAND(INV(i_S(1), INV(i_S(0))) \\
w_NE &= \overline{\overline{w_S31_30}\ \overline{w_S29_28}\ \overline{w_S27_26}\ \overline{w_S25_24}\ \overline{w_S23_22} \cdot \cdot} \\
&\quad\ \overline{\overline{w_S21_20}\ \overline{w_S19_18}\ \overline{w_S17_16}\ \overline{w_S15_14}\ \overline{w_S13_12} \cdot \cdot} \\
&\quad\ \overline{\overline{w_S11_10}\ \overline{w_S09_08}\ \overline{w_S07_06}\ \overline{w_S05_04}\ \overline{w_S03_02} \cdot \cdot} \\
&\quad\ \overline{\overline{w_S01_00}} \\
w_S31_28 &= NAND(INV(w_S31_30), INV(w_S29_28)) \\
w_S27_24 &= NAND(INV(w_S27_26), INV(w_S25_24)) \\
w_S23_20 &= NAND(INV(w_S23_22), INV(w_S21_20)) \\
w_S19_16 &= NAND(INV(w_S19_18), INV(w_S17_16)) \\
w_S15_12 &= NAND(INV(w_S15_14), INV(w_S13_12))
\end{aligned}
\tag{20}
$$

$$
\begin{aligned}
w_S11_08 &= NAND(INV(w_S11_10), INV(w_S09_08)) \\
w_S07_04 &= NAND(INV(w_S07_06), INV(w_S05_04)) \\
w_S03_00 &= NAND(INV(w_S03_02), INV(w_S01_00)) \\
w_NE &= \overline{\overline{w_S31_28}\ \overline{w_S27_24}\ \overline{w_S23_20}\ \overline{w_S19_16} \cdots} \\
&\qquad \overline{\overline{w_S15_12}\ \overline{w_S11_08}\ \overline{w_S07_04}\ \overline{w_S03_00}} \\
w_S31_24 &= NAND(INV(w_S31_28), INV(w_S27_24)) \\
w_S23_16 &= NAND(INV(w_S23_20), INV(w_S19_16)) \\
w_S15_08 &= NAND(INV(w_S15_12), INV(w_S11_08)) \\
w_S07_00 &= NAND(INV(w_S07_04), INV(w_S03_00)) \\
w_NE &= \overline{\overline{w_S31_24}\ \overline{w_S23_16}\ \overline{w_S15_08}\ \overline{w_S07_00}} \\
w_S31_16 &= NAND(INV(w_S31_24), INV(w_S23_16)) \\
w_S15_00 &= NAND(INV(w_S15_08), INV(w_S07_00)) \\
w_NE &= \overline{\overline{w_S31_16}\ \overline{w_S15_00}} \\
w_NE &= NAND(INV(w_S31_16), INV(w_S15_00)) \\
w_EQ &= INV(w_NE0)
\end{aligned}
\tag{21}
$$

The following logic shows how the less than or equal to zero comparison is computed.

$$
\begin{aligned}
w_LEZ &= w_LTZ0 + w_EQ \\
w_LEZ &= \overline{\overline{w_LTZ0 + w_EQ}} \\
w_LEZ &= \overline{\overline{w_LTZ0}\ \overline{w_EQ}} \\
w_LEZ &= \overline{w_GEZ\ w_NE} \\
w_LEZ &= NAND(w_GEZ, w_NE)
\end{aligned}
\tag{22}
$$

The following logic shows how the greater than zero comparison is computed.

$$
\begin{aligned}
w_GTZ &= \overline{w_LEZ} \\
w_GTZ &= INV(w_LEZ)
\end{aligned}
\tag{23}
$$

If overflow can occur, the following logic shows how the less than zero comparison is computed. Once again note that A and B cannot be equal so that if less than zero is 1, so is less than or equal to 0. If less than zero is 0, so is less than or equal to 0.

$$
\begin{aligned}
w_LTZ1 &= \overline{i_unsigned}\ i_A\ i_B + i_unsigned\ \overline{i_A}\ \overline{i_B} \\
w_LTZ1 &= \overline{\overline{i_unsigned}\ i_A\ i_B + i_unsigned\ \overline{i_A}\ \overline{i_B}} \\
w_LTZ1 &= \overline{(i_unsigned + \overline{i_A} + \overline{i_B})(\overline{i_unsigned} + i_A + i_B)} \\
w_LTZ1 &= \overline{i_unsigned\ \overline{i_unsigned} + i_unsigned\ i_A + i_unsigned\ i_B + \cdots} \\
&\quad \overline{\overline{i_unsigned}\ \overline{i_A} + i_A\ \overline{i_A} + \overline{i_A}\ i_B + \cdots} \\
&\quad \overline{\overline{i_unsigned}\ \overline{i_B} + i_A\ \overline{i_B} + i_B\ \overline{i_B}} \\
w_LTZ1 &= \overline{i_unsigned(i_A + i_B) + \overline{i_unsigned}(\overline{i_A} + \overline{i_B}) + \cdots} \\
&\quad \overline{i_A\ \overline{i_B} + \overline{i_A}\ i_B} \\
w_LTZ1 &= \overline{i_unsigned\ OR(i_A, i_B) + \overline{i_unsigned}\ NAND(\overline{i_A}, \overline{i_B}) + \cdots} \\
&\quad \overline{XOR(i_A, i_B)} \\
w_comp4 &= NAND(i_unsigned, OR(i_A, i_B)) \\
w_comp5 &= NAND(\overline{i_unsigned}, NAND(i_A, i_B)) \\
w_LTZ1 &= \overline{\overline{w_comp4} + \overline{w_comp5} + XOR(i_A, i_B)} \\
w_comp6 &= NAND(w_comp4, w_comp5) \\
w_LTZ1 &= \overline{w_comp6 + XOR(i_A, i_B)} \\
w_comp7 &= NAND(\overline{w_comp6}, \overline{XOR(i_A, i_B)}) \\
w_comp7 &= NAND(\overline{w_comp6}, XNOR(i_A, i_B)) \\
w_LTZ1 &= \overline{w_comp7} \\
w_LTZ1 &= INV(w_comp7)
\end{aligned}
\tag{24}
$$

A two input 1-bit multiplexers is used to select between the two possible values for the LTZ comparison; one for no overflow and one for with overflow. The select input to these multiplexers is a signal called w_overcheck which is computed as:

$$\begin{aligned} w_overcheck_n &= NAND(i_overflow, i_XNOR_AB) \\ w_overcheck &= INV(w_overcheck_n) \end{aligned} \tag{25}$$

As previously mentioned, an overflow can only occur when the most significant bits of A and B are the same. Finally, an eight input 1-bit multiplexer selects one of the six comparisons to output. The select signal is a control signal issued by the Controller that decides which comparison should be output. This control signal is dependent upon the instruction. If there is an error in the control signal such that data input 6 or 7 of the multiplexer is selected, these are set to be greater than or equal to zero output. Table 27 shows the input and output connections for the comparator. Table 28 shows some of the internal connections of the comparator for further clarification.

Table 27. Comparator Inputs and Outputs

Port Name	Type	Width	Description
i_S	Input	32	Sum from the Adder
i_NAND_AB	Input	1	NAND(A(31),B(31))
i_OR_AB	Input	1	OR(A(31),B(31))
i_XNOR_AB	Input	1	XNOR(A(31),B(31))
i_unsigned	Input	1	1 for an unsigned comparision, 0 for a signed comparison
i_overflow	Input	1	1 for SLT, SLTU, SLTI, SLTIU instructions, 0 for BLTZinstructions. Does not matter for other instructions
i_COMP_SEL	Input	3	0 for GEZ 1 for LTZ 2 for EQ 3 for NE 4 for LEZ 5 for GTZ 6 for GEZ 7 for GEZ
o_Z	Output	1	Selected Comparison Result

Table 28. Comparator Connections

w_LTZ MUX			
I/O	Type	Width	Connection
i_0	Input	1	w_LTZ0
i_1	Input	1	w_LTZ1
i_S	Input	1	w_overcheck
o_Z	Output	1	w_LTZ

Output MUX			
I/O	Type	Width	Connection
i_0	Input	1	w_GEZ
i_1	Input	1	w_LTZ
i_2	Input	1	w_EQ
i_3	Input	1	w_NE
i_4	Input	1	w_LEZ
i_5	Input	1	w_GTZ
i_6	Input	1	w_GEZ
i_7	Input	1	w_GEZ
i_S	Input	1	i_COMP_SEL
o_Z	Output	1	o_Z

2.12 Signed/Unsigned Extensions

2.12.1 Signed 16-Bit to 32-Bit Extension

This module takes a 16-bit input and produces a 32-bit output. The least significant 16 bits of the output are the input bits. The most significant 16 bits are all equal to the most significant bit of the 16-bit input.

Table 29. Sign_Extend16_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	16	Data Input
o_Z	Output	32	Sign Extended Input

2.12.2 Unsigned 16-Bit to 32-Bit Extension

This module takes a 16-bit input and produces a 32-bit output. The least significant 16 bits of the output are the input bits. The most significant 16 bits are all equal to zero.

Table 30. Extend16_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	16	Data Input
o_Z	Output	32	Zero Extended Input

2.12.3 Unsigned 5-Bit to 32-Bit Extension

This module takes a 5-bit input and produces a 32-bit output. The least significant 5 bits of the output are the input bits. The most significant 27 bits are all equal to zero.

Table 31. Extend5_32 Inputs and Outputs

Port Name	Type	Width	Description
i_A	Input	5	Data Input
o_Z	Output	32	Sign Extended Input

2.13 1-Bit Register

The register is simply a D flip-flop. Whatever data is present on the D input is stored in the register at the rising edge of the clock input. The output Q is the stored value of the register. If D changes after the rising edge of the clock, Q does not change. Q only changes when a new value is stored in the register on the next rising edge. There is also a reset input that forces the stored value to 0 when the

reset input is 1. When the reset input is 0, the register behaves normally.

Table 32. myReg1 Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock Input
i_reset	Input	1	Reset Input
i_D	Input	1	Data Input
o_Q	Output	1	Stored Register Value

2.14 Arithmetic Logic Unit

The Arithmetic Logic Unit Core (ALU Core) combines the left shift, right shift, bitwise operations, adder, load upper immediate, comparator, and overflow detection in order to perform all the necessary arithmetic and logic operations needed by the Basic MIPS architecture. It receives several control signals to select and modify the inputs to the various sub-modules as well as control signals to select which submodules should be output. Table 33 shows the inputs and outputs of the ALU Core.

Table 33. ALU Core Inputs and Outputs

Port Name	Type	Width	Description
i_ALU_SRC_A	Input	2	Select Source A Input
i_ALU_SRC_B	Input	2	Select Source B Input
i_ALU_INV_B	Input	1	Invert the B Input
i_COMP_SEL	Input	3	Select the Comparator Output
i_OVER_CTRL	Input	2	Overflow Control Signal
i_ALU_OUTPUT	Input	4	Select the ALU Output
i_RS_DATA	Input	32	RS Register Data
i_RT_DATA	Input	32	RT Register Data
i_RD_DATA	Input	32	RD Register Data
i_PC	Input	32	PC Register Data
i_imm	Input	16	Immediate Data
i_unsigned	Input	1	Comparator Unsigned Control Signal
i_overflow	Input	1	Comparator Overflow Control Signal
i_imm_extend	Input	1	Determine whether immediate is zero or sign extended
o_ALU_Result	Output	32	ALU Output

Table 34 shows the connections within the ALU Core and Figure 18 shows the connections in a schematic format.

Table 34. ALU Core Connections

Extend SA Extend5_32			
I/O	Type	Width	Connection
i_A	Input	5	i_imm(10 downto 6) This is SA (shift ammount)
o_Z	Output	32	w_SA_EXT
Zero Extend Immediate Extend16_32			
I/O	Type	Width	Connection
i_A	Input	16	i_imm
o_Z	Output	32	w_IMM_EXT0
Sign Extend Immediate Sign_Extend16_32			
I/O	Type	Width	Connection
i_A	Input	16	i_imm
o_Z	Output	32	w_IMM_EXT1

Table 34 – *Continued on next page*

Table 34 – *Continued from previous page*

Shift Sign Extended Immediate Left by Two SLL2_32			
I/O	Type	Width	Connection
i_A	Input	32	w_IMM_EXT1
o_Z	Output	32	w_IMM_SLL2
Determine Data Input 2 to ALU_SRC_B MUX MUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	w_IMM_EXT0
i_1	Input	32	w_IMM_EXT1
i_S	Input	2	i_imm_extend
o_Z	Output	32	w_IMM_EXT
Determine Data Input 3 to ALU_SRC_B MUX MUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	4
i_1	Input	32	w_IMM_SLL2
i_S	Input	2	ff_comp_out
o_Z	Output	32	w_ALU_SRC_B_3
ALU_SRC_A myMUX4_32			
I/O	Type	Width	Connection
i_0	Input	32	i_RS_DATA
i_1	Input	32	w_SA_EXT
i_2	Input	32	i_PC
i_3	Input	32	i_PC
i_S	Input	2	i_ALU_SRC_A
o_Z	Output	32	w_A
ALU_SRC_B myMUX4_32			
I/O	Type	Width	Connection
i_0	Input	32	0
i_1	Input	32	i_RT_DATA
i_2	Input	32	w_IMM_EXT
i_3	Input	32	w_ALU_SRC_B_3
i_S	Input	2	i_ALU_SRC_B
o_Z	Output	32	w_B0
Invert B Input to ALU Core INV_32			
I/O	Type	Width	Connection
i_A	Input	32	w_B0
o_Z	Output	32	w_B0_n
Determine If B Input Is Inverted myMUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	w_B0
i_1	Input	32	w_B0_n

Table 34 – *Continued on next page*

Table 34 – *Continued from previous page*

i_S	Input	2	i_ALU_INV_B
o_Z	Output	32	w_B
LEFT_SHIFT			
I/O	Type	Width	Connection
i_A	Input	32	w_B
i_SA	Input	5	w_A(4 downto 0)
o_Z	Output	32	w_SLX
RIGHT_SHIFT			
I/O	Type	Width	Connection
i_A	Input	32	w_B
i_SA	Input	5	w_A(4 downto 0)
i_ARITH	Input	1	i_imm(0)
o_Z	Output	32	w_SRX
BITWISE2__32			
I/O	Type	Width	Connection
i_A	Input	32	w_A
i_B	Input	32	w_B
o_A_n	Output	32	w_A_n Inverted A
o_B_n	Output	32	w_B_n Inverted B
o_AND	Output	32	w_AND
o_NAND	Output	32	w_AB_NAND
o_OR	Output	32	w_OR
o_XOR	Output	32	w_XOR
o_XNOR	Output	32	w_XNOR
o_NOR	Output	32	w_NOR
CARRY_SELECT_ADDER_32			
I/O	Type	Width	Connection
i_A	Input	32	w_A
i_A_n	Input	32	w_A_n Inverted A
i_B	Input	32	w_B
i_B_n	Input	32	w_B_n Inverted B
i_AB_AND	Input	32	w_AND
i_AB_NAND	Input	32	w_AB_NAND
i_AB_NOR	Input	32	w_NOR
i_C	Input	1	i_ALU_INV_B
o_S	Output	32	w_ADD0
o_C	Output	1	OPEN - unconnected
LUI			
I/O	Type	Width	Connection
i_A	Input	16	w_B(15 downto 0)

Table 34 – *Continued on next page*

Table 34 – *Continued from previous page*

o_Z	Output	32	w_LUI
Comparator			
I/O	Type	Width	Connection
i_S	Input	32	w_ADD0
i_NAND_AB	Input	1	w_AB_NAND(31)
i_OR_AB	Input	1	w_OR(31)
i_XNOR_AB	Input	1	w_XNOR(31)
i_unsigned	Input	1	i_UNSIGNED
i_overflow	Input	1	i_overflow
i_COMP_SEL	Input	3	i_COMP_SEL
o_Z	Output	32	w_comp_out
myReg_delay_1 myReg1			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_D	Input	1	w_comp_out
o_Q	Output	1	f_comp_out
myReg_delay_2 myReg1			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_D	Input	1	f_comp_out
o_Q	Output	1	ff_comp_out
Selected Comparison myMUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	0
i_1	Input	32	1
i_S	Input	1	w_comp_out
o_Z	Output	32	w_COMP
Over_Detect			
I/O	Type	Width	Connection
i_AB_NAND	Input	1	w_AB_NAND(31)
i_AB_OR	Input	1	w_OR(31)
i_AB_XNOR	Input	1	w_XNOR(31)
i_S	Input	32	w_ADD0
i_RD	Input	32	i_RD_DATA
i_RT	Input	32	i_RT_DATA
i_OVER_CTRL	Input	2	i_OVER_CTRL
o_Z	Output	32	w_ADD
ALU Output myMUX16_32			

Table 34 – *Continued on next page*

Table 34 – *Continued from previous page*

I/O	Type	Width	Connection
i_0	Input	32	w_SLX
i_1	Input	32	w_SRX
i_2	Input	32	w_ADD
i_3	Input	32	w_AND
i_4	Input	32	w_OR
i_5	Input	32	w_XOR
i_6	Input	32	w_NOR
i_7	Input	32	w_COMP
i_8	Input	32	w_LUI
i_9	Input	32	w_SRX
i_10	Input	32	w_ADD
i_11	Input	32	w_AND
i_12	Input	32	w_OR
i_13	Input	32	w_XOR
i_14	Input	32	w_NOR
i_15	Input	32	w_COMP
i_S	Input	4	i_ALU_OUTPUT
o_Z	Output	32	o_ALU_RESULT

Each particular instruction uniquely affects the operation of the ALU Core. The control signal values associated with each instruction are shown in Table 35. The table only shows the values of the control signals for the first iteration of calculations for branch instructions. Calculating branches requires two cycles through the Datapath. The first determines if the branch should be taken. The second determines the value of the program counter. If the branch comparison is true, the branch will be taken and program counter will be $PC = PC + (offset << 2)$. Otherwise, $PC = PC + 4$. The controls for the second cycle through the Datapath for the branch instructions are shown in Table 36. The load word instruction also requires a second cycle through the Datapath, but control signals do not change on the second cycle from those on the first cycle so they are not shown in Table 36. Note that X's denote "don't care" meaning that the value of this signal does not matter nor does it affect the output.

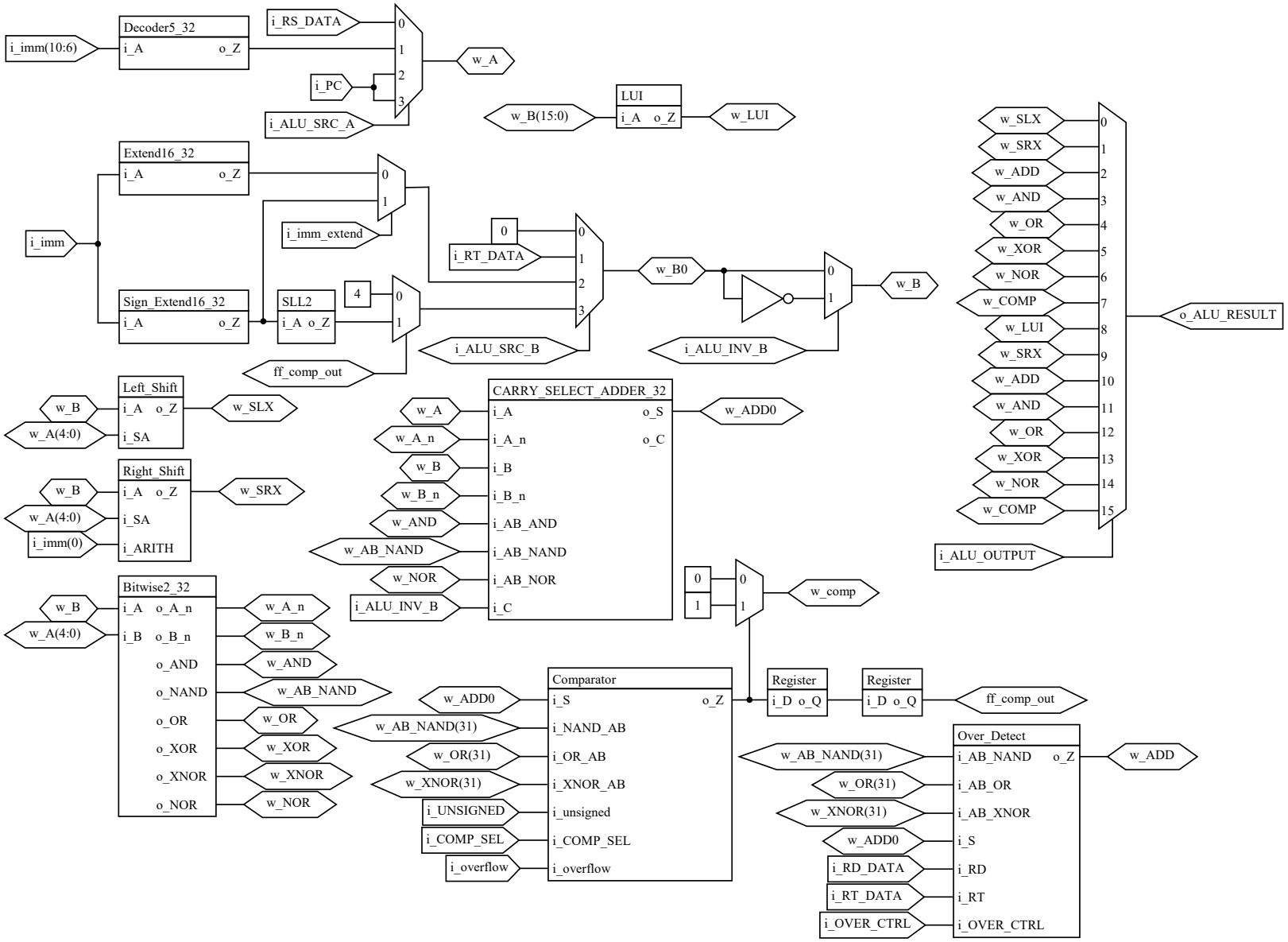

Figure 18. ALU Core Schematic

Table 35. ALU Control Signals Cycle 1

Instruction	i_ALU_ SRC_A	i_ALU_ SRC_B	i_ALU_ INV_B	i_COMP_ SEL	i_OVER_ CTRL	i_ALU_ OUTPUT	i_UNSIGNED	i_overflow
SLL	01	01	0	XXX	00	0000	0	X
NOP	01	01	0	XXX	00	0000	0	X
SRL	01	01	0	XXX	00	0001	0	X
SRA	01	01	0	XXX	00	0001	0	X
SLLV	00	01	0	XXX	00	0000	0	X
SRLV	00	01	0	XXX	00	0001	0	X
SRAV	00	01	0	XXX	00	0001	0	X
ADD	00	01	0	XXX	01	0010	0	X
ADDU	00	01	0	XXX	00	0010	0	X
SUB	00	01	1	XXX	10	0010	0	X
SUBU	00	01	1	XXX	00	0010	0	X
AND	00	01	0	XXX	00	0011	0	X
OR	00	01	0	XXX	00	0100	0	X
XOR	00	01	0	XXX	00	0101	0	X
NOR	00	01	0	XXX	00	0110	0	X
SLT	00	01	1	001	00	0111	0	1
SLTU	00	01	1	001	00	0111	1	1
BGEZ	00	00	0	000	00	XXXX	0	X
BLTZ	00	00	0	001	00	XXXX	0	0
BEQ	00	01	1	010	00	XXXX	0	X
BNE	00	01	1	011	00	XXXX	0	X
BLEZ	00	00	0	100	00	XXXX	0	X
BGTZ	00	00	0	101	00	XXXX	0	X
ADDI	00	10	0	XXX	11	0010	0	X

Table 35 – *Continued on next page*

Table 35 – *Continued from previous page*

Instruction	i_ALU_ SRC_A	i_ALU_ SRC_B	i_ALU_ INV_B	i_COMP_ SEL	i_OVER_ CTRL	i_ALU_ OUTPUT	i_UNSIGNED	i_overflow
ADDIU	00	10	0	XXX	11	0010	0	X
SLTI	00	10	1	001	00	0111	0	1
SLTIU	00	10	1	001	00	0111	1	1
ANDI	00	10	0	XXX	00	0011	0	X
ORI	00	10	0	XXX	00	0100	0	X
XORI	00	10	0	XXX	00	0101	0	X
LUI	00	10	0	XXX	00	1000	0	X
LW	00	10	0	XXX	00	0010	0	X
SW	00	10	0	XXX	00	0010	0	X

Table 36. ALU Control Signals Cycle 2

Instruction	i_ALU_ SRC_A	i_ALU_ SRC_B	i_ALU_ INV_B	i_COMP_ SEL	i_OVER_ CTRL	i_ALU_ OUTPUT	i_UNSIGNED	i_overflow
BGEZ	10 or 11	11	0	XXX	00	0010	0	X
BLTZ	10 or 11	11	0	XXX	00	0010	0	1
BEQ	10 or 11	11	0	XXX	00	0010	0	X
BNE	10 or 11	11	0	XXX	00	0010	0	X
BLEZ	10 or 11	11	0	XXX	00	0010	0	X
BGTZ	10 or 11	11	0	XXX	00	0010	0	X

In order to achieve the desired timing, the output of the comparator for branch instructions is delayed two clock cycles as can be seen in the connection diagram. This ensures that the result of the comparison is available to the multiplexer that determine if $PC = PC + 4$ if the branch condition is false or if $PC = PC + (offset << 2)$ if the branch condition is true.

To further illustrate how the ALU Core operates, consider the subtract instruction. In this case, ALU input A is selected to be the data in the register denoted by RS and ALU input B is selected to be the data in the register denoted by RT. The invert B signal is high so the B input is inverted. The A and inverted B values enter the adder and the carry in of the adder is set to 1. Inverting B and setting the carry-in of the adder to 1 essentially computes the 2's complement value of B, without creating a separate adder to compute the 2's complement of B. The output of the adder enters the overflow detection. If an overflow occurs, the data currently stored in the register denoted by RD is output from the overflow detection module. If no overflow occurs, the sum from the adder is output from the overflow detection module. The output from the overflow detection module is input to the adder input of the output selector multiplexer. The ALU output control selects the adder input to the output selector multiplexer to be the ALU result. While all of the other submodules operate on the A and B inputs, the results of these submodules do not matter because they are not selected by the output selector multiplexer. It is left as an exercise to the reader to trace the flow of data through the ALU Core for the remaining operations.

One final note before concluding the ALU Core description is the connection of the inputs for the output selector multiplexer. There are only nine different outputs that the ALU Core can choose from, but all 16 multiplexer inputs are connected. The choice of data inputs to the remaining seven pins was made so that if a single bit error occurred in the most significant bit of the ALU Output control signal, the

only time an error would occur would be for the left shift and load upper immediate outputs. An error in the most significant bit of the ALU output control signal for a left shift would be interpreted as the ALU output control for the load upper immediate instruction. The reciprocal statement is also true. Before moving to the discussion on the Datapath as a whole, there are some more modules that must be discussed that make up the Datapath.

2.15 5-Bit to 32-Bit Decoder

The purpose of the 5-bit to 32-bit decoder is to take a 5-bit input and set one of 32 outputs equal to 1 and the remaining 31 outputs equal to 0. This is done so that an input of 00000 corresponds to a 1 on output 0 and an input of 11111 corresponds to a 1 on output 31. Basically, the unsinged decimal interpretation of the 5-bit input corresponds to the output that should be 1. The logic for the decoder is shown below.

$$
\begin{aligned}
o_Z(0) &= \overline{i_S(4)}\ \overline{i_S(3)}\ \overline{i_S(2)}\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(0) &= \overline{i_S(4) + i_S(3) + i_S(2) + i_S(1) + i_S(0)} \\
w_dec_00 &= NAND(INV(i_S(4)), INV(i_S(3))) \\
w_dec_01 &= NAND(INV(i_S(2)), INV(i_S(1))) \\
o_Z(0) &= \overline{w_dec_00 + w_dec_01 + i_S(0)} \\
w_dec_02 &= NAND(INV(w_dec_00), INV(w_dec_01)) \\
o_Z(0) &= \overline{w_dec_02 + i_S(0)} \\
o_Z(0) &= INV(NAND(INV(w_dec_02), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(1) &= \overline{i_S(4)}\ \overline{i_S(3)}\ \overline{i_S(2)}\ \overline{i_S(1)}\ i_S(0) \\
o_Z(1) &= \overline{w_dec_02 + \overline{i_S(0)}} \\
o_Z(1) &= INV(NAND(INV(w_dec_02), i_S(0)))
\end{aligned}
\tag{26}
$$

$$
\begin{aligned}
o_Z(2) &= \overline{i_S(4)}\ \overline{i_S(3)}\ \overline{i_S(2)}\ i_S(1)\ \overline{i_S(0)} \\
o_Z(2) &= \overline{w_dec_00 + i_S(2) + \overline{i_S(1)} + i_S(0)} \\
w_dec_03 &= NAND(INV(i_S(2)), i_S(1)) \\
o_Z(2) &= \overline{w_dec_00 + w_dec_03 + i_S(0)} \\
w_dec_04 &= NAND(INV(w_dec_00), INV(w_dec_03)) \\
o_Z(2) &= \overline{w_dec_04 + i_S(0)} \\
o_Z(2) &= INV(NAND(INV(w_dec_04), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(3) &= \overline{i_S(4)}\ \overline{i_S(3)}\ \overline{i_S(2)}\ i_S(1)\ i_S(0) \\
o_Z(3) &= \overline{w_dec_04 + \overline{i_S(0)}} \\
o_Z(3) &= INV(NAND(INV(w_dec_04), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(4) &= \overline{i_S(4)}\ \overline{i_S(3)}\ i_S(2)\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(4) &= \overline{w_dec_00 + \overline{i_S(2)} + i_S(1) + i_S(0)} \\
w_dec_05 &= NAND(i_S(2), INV(i_S(1))) \\
o_Z(4) &= \overline{w_dec_00 + w_dec_05 + i_S(0)} \\
w_dec_06 &= NAND(INV(w_dec_00), INV(w_dec_05)) \\
o_Z(4) &= \overline{w_dec_06 + i_S(0)} \\
o_Z(4) &= INV(NAND(INV(w_dec_06), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(5) &= \overline{i_S(4)}\ \overline{i_S(3)}\ i_S(2)\ \overline{i_S(1)}\ i_S(0) \\
o_Z(5) &= \overline{w_dec_06 + \overline{i_S(0)}} \\
o_Z(5) &= INV(NAND(INV(w_dec_06), i_S(0)))
\end{aligned}
\tag{27}
$$

$$
\begin{aligned}
o_Z(6) &= \overline{i_S(4)}\ \overline{i_S(3)}\ i_S(2)\ i_S(1)\ \overline{i_S(0)} \\
o_Z(6) &= \overline{w_dec_00 + \overline{i_S(2)} + \overline{i_S(1)} + i_S(0)} \\
w_dec_07 &= NAND(i_S(2), i_S(1)) \\
o_Z(6) &= \overline{w_dec_00 + w_dec_07 + i_S(0)} \\
w_dec_08 &= NAND(INV(w_dec_07), INV(w_dec_00)) \\
o_Z(6) &= \overline{w_dec_08 + i_S(0)} \\
o_Z(6) &= INV(NAND(INV(w_dec_08), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(7) &= \overline{i_S(4)}\ \overline{i_S(3)}\ i_S(2)\ i_S(1)\ i_S(0) \\
o_Z(7) &= \overline{w_dec_08 + \overline{i_S(0)}} \\
o_Z(7) &= INV(NAND(INV(w_dec_08), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(8) &= \overline{i_S(4)}\ i_S(3)\ \overline{i_S(2)}\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(8) &= \overline{i_S(4) + \overline{i_S(3)} + i_S(2) + i_S(1) + i_S(0)} \\
o_Z(8) &= \overline{i_S(4) + \overline{i_S(3)} + w_dec_01 + i_S(0)} \\
w_dec_09 &= NAND(INV(i_S(4)), i_S(3)) \\
o_Z(8) &= \overline{w_dec_09 + w_dec_01 + i_S(0)} \\
w_dec_10 &= NAND(INV(w_dec_09), INV(w_dec_01)) \\
o_Z(8) &= \overline{w_dec_10 + i_S(0)} \\
o_Z(8) &= INV(NAND(INV(w_dec_10), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(9) &= \overline{i_S(4)}\ i_S(3)\ \overline{i_S(2)}\ \overline{i_S(1)}\ i_S(0) \\
o_Z(9) &= \overline{w_dec_10 + \overline{i_S(0)}} \\
o_Z(9) &= INV(NAND(INV(w_dec_10), i_S(0)))
\end{aligned}
\tag{28}
$$

$$
\begin{aligned}
o_Z(10) &= \overline{i_S(4)}\ i_S(3)\ \overline{i_S(2)}\ i_S(1)\ \overline{i_S(0)} \\
o_Z(10) &= \overline{w_dec_09 + w_dec_03 + i_S(0)} \\
w_dec_11 &= NAND(INV(w_dec_09), INV(w_dec_03)) \\
o_Z(10) &= \overline{w_dec_11 + i_S(0)} \\
o_Z(10) &= INV(NAND(INV(w_dec_11), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(11) &= \overline{i_S(4)}\ i_S(3)\ \overline{i_S(2)}\ i_S(1)\ i_S(0) \\
o_Z(11) &= \overline{w_dec_11 + \overline{i_S(0)}} \\
o_Z(11) &= INV(NAND(INV(w_dec_11), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
&o_Z(12) = \overline{i_S(4)}\ i_S(3)\ i_S(2)\ \overline{i_S(1)}\ \overline{i_S(0)} \\
&o_Z(12) = \overline{w_dec_09 + w_dec_05 + i_S(0)} \\
&w_dec_12 = NAND(INV(w_dec_09), INV(w_dec_05)) \\
&o_Z(12) = \overline{w_dec_12 + i_S(0)} \\
&o_Z(12) = INV(NAND(INV(w_dec_12), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
&o_Z(13) = \overline{i_S(4)}\ i_S(3)\ i_S(2)\ \overline{i_S(1)}\ i_S(0) \\
&o_Z(13) = \overline{w_dec_12 + \overline{i_S(0)}} \\
&o_Z(13) = INV(NAND(INV(w_dec_12), i_S(0)))
\end{aligned}
\tag{29}
$$

$$
\begin{aligned}
&o_Z(14) = \overline{i_S(4)}\ i_S(3)\ i_S(2)\ i_S(1)\ \overline{i_S(0)} \\
&o_Z(14) = \overline{w_dec_09 + w_dec_07 + i_S(0)} \\
&w_dec_13 = NAND(INV(w_dec_09), INV(w_dec_07)) \\
&o_Z(14) = \overline{w_dec_13 + i_S(0)} \\
&o_Z(14) = INV(NAND(INV(w_dec_13), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
&o_Z(15) = \overline{i_S(4)}\ i_S(3)\ i_S(2)\ i_S(1)\ i_S(0) \\
&o_Z(15) = \overline{w_dec_13 + \overline{i_S(0)}} \\
&o_Z(15) = INV(NAND(INV(w_dec_13), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(16) &= i_S(4)\ \overline{i_S(3)}\ \overline{i_S(2)}\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(16) &= \overline{\overline{i_S(4)} + i_S(3) + i_S(2) + i_S(1) + i_S(0)} \\
o_Z(16) &= \overline{\overline{i_S(4)} + i_S(3) + w_dec_01 + i_S(0)} \\
w_dec_14 &= NAND(i_S(4), INV(i_S(3))) \\
o_Z(16) &= \overline{w_dec_14 + w_dec_01 + i_S(0)} \\
w_dec_15 &= NAND(INV(w_dec_14), INV(w_dec_01)) \\
o_Z(16) &= \overline{w_dec_15 + i_S(0)} \\
o_Z(16) &= INV(NAND(INV(w_dec_15), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(17) &= i_S(4)\ \overline{i_S(3)}\ \overline{i_S(2)}\ \overline{i_S(1)}\ i_S(0) \\
o_Z(17) &= \overline{w_dec_15 + \overline{i_S(0)}} \\
o_Z(17) &= INV(NAND(INV(w_dec_15), i_S(0)))
\end{aligned}
\tag{30}
$$

$$
\begin{aligned}
o_Z(18) &= i_S(4)\ \overline{i_S(3)}\ \overline{i_S(2)}\ i_S(1)\ \overline{i_S(0)} \\
o_Z(18) &= \overline{w_dec_14 + w_dec_03 + i_S(0)} \\
w_dec_16 &= NAND(INV(w_dec_14), INV(w_dec_03)) \\
o_Z(18) &= \overline{w_dec_16 + i_S(0)} \\
o_Z(18) &= INV(NAND(INV(w_dec_16), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(19) &= i_S(4)\ \overline{i_S(3)}\ \overline{i_S(2)}\ i_S(1)\ i_S(0) \\
o_Z(19) &= \overline{w_dec_16 + \overline{i_S(0)}} \\
o_Z(19) &= INV(NAND(INV(w_dec_16), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(20) &= i_S(4)\ \overline{i_S(3)}\ i_S(2)\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(20) &= \overline{w_dec_14 + w_dec_05 + i_S(0)} \\
w_dec_17 &= NAND(INV(w_dec_14), INV(w_dec_05)) \\
o_Z(20) &= \overline{w_dec_17 + i_S(0)} \\
o_Z(20) &= INV(NAND(INV(w_dec_17), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(21) &= i_S(4)\ \overline{i_S(3)}\ i_S(2)\ \overline{i_S(1)}\ i_S(0) \\
o_Z(21) &= \overline{w_dec_17 + \overline{i_S(0)}} \\
o_Z(21) &= INV(NAND(INV(w_dec_17), i_S(0)))
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
o_Z(22) &= i_S(4)\ \overline{i_S(3)}\ i_S(2)\ i_S(1)\ \overline{i_S(0)} \\
o_Z(22) &= \overline{w_dec_14 + w_dec_07 + i_S(0)} \\
w_dec_18 &= NAND(INV(w_dec_14), INV(w_dec_07)) \\
o_Z(22) &= \overline{w_dec_18 + i_S(0)} \\
o_Z(22) &= INV(NAND(INV(w_dec_18), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(23) &= i_S(4)\ \overline{i_S(3)}\ i_S(2)\ i_S(1)\ i_S(0) \\
o_Z(23) &= \overline{w_dec_18 + \overline{i_S(0)}} \\
o_Z(23) &= INV(NAND(INV(w_dec_18), i_S(0)))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(24) &= i_S(4)\ i_S(3)\ \overline{i_S(2)}\ \overline{i_S(1)}\ \overline{i_S(0)} \\
o_Z(24) &= \overline{\overline{i_S(4)} + \overline{i_S(3)} + i_S(2) + i_S(1) + i_S(0)} \\
o_Z(24) &= \overline{\overline{i_S(4)} + \overline{i_S(3)} + w_dec_01 + i_S(0)} \\
w_dec_19 &= NAND(i_S(4), i_S(3)) \\
o_Z(24) &= \overline{w_dec_19 + w_dec_01 + i_S(0)} \\
w_dec_20 &= NAND(INV(w_dec_19), INV(w_dec_01)) \\
o_Z(24) &= \overline{w_dec_20 + i_S(0)} \\
o_Z(24) &= INV(NAND(INV(w_dec_20), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(25) &= i_S(4)\ i_S(3)\ \overline{i_S(2)}\ \overline{i_S(1)}\ i_S(0) \\
o_Z(25) &= \overline{w_dec_20 + \overline{i_S(0)}} \\
o_Z(25) &= INV(NAND(INV(w_dec_20), i_S(0)))
\end{aligned}
\tag{32}
$$

$$
\begin{aligned}
o_Z(26) &= i_S(4)\ i_S(3)\ \overline{i_S(2)}\ i_S(1)\ \overline{i_S(0)} \\
o_Z(26) &= \overline{w_dec_19 + w_dec_03 + i_S(0)} \\
w_dec_21 &= NAND(INV(w_dec_19), INV(w_dec_03)) \\
o_Z(26) &= \overline{w_dec_21 + i_S(0)} \\
o_Z(26) &= INV(NAND(INV(w_dec_21), INV(i_S(0))))
\end{aligned}
$$

$$
\begin{aligned}
o_Z(27) &= i_S(4)\ i_S(3)\ \overline{i_S(2)}\ i_S(1)\ i_S(0) \\
o_Z(27) &= \overline{w_dec_21 + \overline{i_S(0)}} \\
o_Z(27) &= INV(NAND(INV(w_dec_21), i_S(0)))
\end{aligned}
$$

$$o_Z(28) = i_S(4)\ i_S(3)\ i_S(2)\ \overline{i_S(1)}\ \overline{i_S(0)}$$
$$o_Z(28) = \overline{w_dec_19 + w_dec_05 + i_S(0)}$$
$$w_dec_22 = NAND(INV(w_dec_19), INV(w_dec_05))$$
$$o_Z(28) = \overline{w_dec_22 + i_S(0)}$$
$$o_Z(28) = INV(NAND(INV(w_dec_22), INV(i_S(0))))$$

$$o_Z(29) = i_S(4)\ i_S(3)\ i_S(2)\ \overline{i_S(1)}\ i_S(0)$$
$$o_Z(29) = \overline{w_dec_22 + \overline{i_S(0)}}$$
$$o_Z(29) = INV(NAND(INV(w_dec_22), i_S(0)))$$

(33)

$$o_Z(30) = i_S(4)\ i_S(3)\ i_S(2)\ i_S(1)\ \overline{i_S(0)}$$
$$o_Z(30) = \overline{w_dec_19 + w_dec_07 + i_S(0)}$$
$$w_dec_23 = NAND(INV(w_dec_19), INV(w_dec_07))$$
$$o_Z(30) = \overline{w_dec_23 + i_S(0)}$$
$$o_Z(30) = INV(NAND(INV(w_dec_23), INV(i_S(0))))$$

$$o_Z(31) = i_S(4)\ i_S(3)\ i_S(2)\ i_S(1)\ i_S(0)$$
$$o_Z(31) = \overline{w_dec_23 + \overline{i_S(0)}}$$
$$o_Z(31) = INV(NAND(INV(w_dec_23), i_S(0)))$$

Table 37. Decoder5_32 Inputs and Outputs

Port Name	Type	Width	Description
i_S	Input	5	Encoded Input
o_Z	Output	32	Decoded Output

2.16 Registers

2.16.1 N-Bit Register

The N-Bit register (myRegN) consists of N 1-bit registers in parallel. Each of which is connected to the same clock and reset signals, but different input and output data bits. The input and output data signals are N-bit busses.

Table 38. myRegN Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock Input
i_reset	Input	1	Reset Input
i_D	Input	N	Data Input
o_Q	Output	N	Stored Register Value

2.16.2 Enabled Register

The enabled register adds an enable signal to an N-bit register configured as a 32-bit register. The enable signal acts as a select signal for a 32-bit, 2-input multiplexer. When the enable signal is 0, the output of the register is fed back into the input and the stored value of the register remains unchanged. When the enable signal is 1, the incoming data is fed into the register and the stored value of the register is updated. Figure 19 shows how an enabled register is connected.

Table 39. Enabled_Register Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock Input
i_reset	Input	1	Reset Input
i_data	Input	32	Data Input
i_en	Input	1	Enable Input
o_Q	Output	32	Stored Register Value

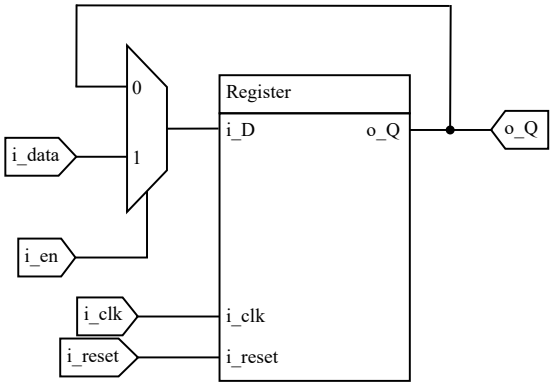

Figure 19. Enabled Register Schematic

2.16.3 General Purpose Register (GPR) Bank

The general purpose register bank consists of 32, 32-bit registers numbered from 0 to 31. Each register is enabled in the same fashion as an enabled register, however they do not use the enabled register. As an aside, the GPR_Bank was created before the enabled register and was the inspiration for the enabled register and no additional logic was used for each register in the GPR_Bank than was used for the enabled register. The input data to all 32 registers is the same. The difference is in the enable signal. The enable signal comes from a Decoder5_32 module which takes a 5-bit signal to select which register should be enabled. The only unique register is the 0 register. This register will retain its value regardless of whether it is enabled or not and this value is a 32-bit 0. The outputs of all 32 registers are available. Figure 20 shows the connections in the GPR Bank with the clock and reset signals omitted.

Table 40. GPR_Bank Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock Input
i_reset	Input	1	Reset Input
i_data	Input	32	Data Input
i_sel	Input	5	Select Input
o_Q00	Output	32	Register 0 Stored Value
o_Q01	Output	32	Register 1 Stored Value
o_Q02	Output	32	Register 2 Stored Value
o_Q03	Output	32	Register 3 Stored Value
o_Q04	Output	32	Register 4 Stored Value
o_Q05	Output	32	Register 5 Stored Value
o_Q06	Output	32	Register 6 Stored Value
o_Q07	Output	32	Register 7 Stored Value
o_Q08	Output	32	Register 8 Stored Value
o_Q09	Output	32	Register 9 Stored Value
o_Q10	Output	32	Register 10 Stored Value
o_Q11	Output	32	Register 11 Stored Value
o_Q12	Output	32	Register 12 Stored Value
o_Q13	Output	32	Register 13 Stored Value
o_Q14	Output	32	Register 14 Stored Value
o_Q15	Output	32	Register 15 Stored Value
o_Q16	Output	32	Register 16 Stored Value
o_Q17	Output	32	Register 17 Stored Value
o_Q18	Output	32	Register 18 Stored Value
o_Q19	Output	32	Register 19 Stored Value
o_Q20	Output	32	Register 20 Stored Value
o_Q21	Output	32	Register 21 Stored Value
o_Q22	Output	32	Register 22 Stored Value
o_Q23	Output	32	Register 23 Stored Value
o_Q24	Output	32	Register 24 Stored Value
o_Q25	Output	32	Register 25 Stored Value
o_Q26	Output	32	Register 26 Stored Value
o_Q27	Output	32	Register 27 Stored Value
o_Q28	Output	32	Register 28 Stored Value
o_Q29	Output	32	Register 29 Stored Value
o_Q30	Output	32	Register 30 Stored Value
o_Q31	Output	32	Register 31 Stored Value

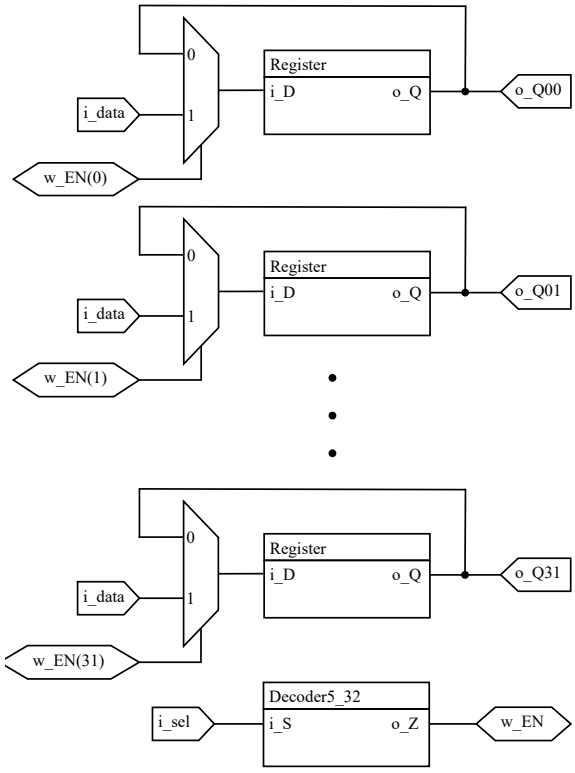

Figure 20. GPR Bank Schematic

2.17 Datapath

The Datapath connects the ALU Core to the GPR Bank and the program counter register. The Datapath also connects to external memory to read data from memory into the GPR Bank and from the GPR Bank to memory. The control signals i_ALU_SRC_A, i_ALU_SRC_B, i_ALU_INV_B, i_COMP_SEL, i_OVER_CTRL, i_ALU_OUTPUT, i_UNSIGNED, and i_imm_extend are inputs to the Datapath, but are directly passed to the ALU Core and perform no other function in the Datapath. A 1-bit control signal called i_STORE_FROM_MEM is the select signal to a 2-input, 32-bit multiplexer that provides data to the GPR Bank. If this signal is 0, the data comes from the result of the ALU Core, w_ALU_RESULT. If this signal is 1, the data comes from external memory. A 2-bit control signal called i_REG_SEL is the

select signal to a 4-input, 5-bit multiplexer that provides the select input to the GPR Bank. If this signal is 00 or 11, the select input to the GPR Bank is a 5-bit 0. If this signal is 01, the select input to the GPR Bank is the RD register specified by the 32-bit instruction. If this signal is 10, the select input to the GPR Bank is the RT register specified by the 32-bit instruction. The RD register is specified by bits 15 downto 11 of the 32-bit instruction. However, since the Datapath operates on both RD and the immediate value and the immediate value contains bits 15 downto 0 of the 32-bit instruction, only the immediate value is passed into the Datapath to reduce redundant signals. The RD register select signal is simply i_imm(15 downto 11). The RT register is specified by bits 20 downto 16 of the 32-bit instruction and is denoted by i_RT_SEL. Another register selection signal is the RS signal denoted by i_RS_SEL and corresponds to bits 25 downto 21 of the 32-bit instruction. In addition to being inputs to the multiplexer that determines the select signal for the GPR Bank, the RD and RT register selection signals each act as the select signal for 32-input, 32-bit multiplexers that determine the w_RD_DATA and w_RT_DATA inputs to the ALU Core. Each of these 32-input, 32-bit multiplexers take in the 32 GPR Bank register outputs as inputs. These inputs are ordered 0 through 31, just as the registers are ordered 0 through 31 so that the select signals correspond to the appropriate register. For example, if i_RD_SEL specifies that the w_RD_DATA should come from GPR Bank register number 18, then i_RD_SEL will be a 5-bit representation of the number 18, namely 10010. In like manner, the i_RS_SEL signal selects the w_RS_DATA that is input to the ALU Core. The immediate value i_imm is also supplied as an input to the ALU Core.

The program counter (PC) register is implemented as an enabled register. The data input may come from one of two locations. The first is the output of the self contained 30-bit carry select adder that is hardwired to always add 1 to bits 2

through 31 of the value stored in the PC register. Bits 0 and 1 are unaffected and appended as the least significant bits of the result of the 30-bit adder so that the result of the addition is PC+4 (w_PC_4). The second is the result of the ALU Core (w_ALU_RESULT). An enable signal called i_PC_EN determines which of these data values are the input to the PC register. The i_PC_EN signal is passed through a 1-bit inverter and this inverted signal is the select signal to the multiplexer choosing between PC+4, data input 0, and ALU result, data input 1. In other words, when i_PC_EN is 0, w_PC_EN_n is 1 and the input to the PC register is the ALU result. When i_PC_EN is 1, w_PC_EN_n is 0 and the input to the PC register is PC+4. The i_PC_EN signal is also an input to a second multiplexer. This second multiplexer is a 2-input, 1-bit multiplexer. If i_PC_EN is 0, then the i_REG_SEL control signal determines whether the PC register is enabled. If i_PC_EN is 1, the PC register is enabled. When i_PC_EN is 1 and the PC register is enabled, the data input is PC+4 as previously discussed. The i_PC_EN signal is only set to 1 at the conclusion of non-branching instructions to increment the program counter by 4. The data 0 input to the 2-input, 1-bit multiplexer is the output of a 4-input, 1-bit multiplexer whose select input is i_REG_SEL. When i_REG_SEL is 00, 01, or 10, the output of this multiplexer is 0 and the PC register is disabled. When i_REG_SEL is 11, the output of this multiplexer is 1 and the PC register is enabled. The only time i_REG_SEL is 11 is at the conclusion of branching instructions so that the PC can be set to $PC + (offset << 2)$ or $PC + 4$ depending on whether the branch condition was true or false respectively as previously discussed in the ALU section. The w_PC signal is also supplied as an input to the ALU Core.

For memory operations, a control signal called i_MEM_ADDRESS_SEL determines whether the address used to access memory comes from the PC or from the ALU result. This 1-bit signal is an input to a 2-input, 32-bit multiplexer whose data 0

input is w_PC and whose data 1 input is w_ALU_RESULT. For most of the time, this signal is set to 1, but at the beginning of each new control cycle in the Controller, it is set to 0 so that instructions can be accessed from memory using the program counter as the address for those instructions. Table 41 shows the inputs and outputs of the Datapath and Table 42 shows how the submodules are interconnected and Figure 21 shows this in a schematic.

Table 41. Datapath Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock Input
i_reset	Input	1	Reset Input
i_ALU_SRC_A	Input	2	Select Source A Input
i_ALU_SRC_B	Input	2	Select Source B Input
i_ALU_INV_B	Input	1	Invert the B Input
i_COMP_SEL	Input	3	Select the Comparator Output
i_OVER_CTRL	Input	2	Overflow Control Signal
i_ALU_OUTPUT	Input	4	Select the ALU Output
i_RS_SEL	Input	5	Select RS Register
i_RT_SEL	Input	5	Select RT Register
i_imm	Input	16	Immediate - also contains the shift amount for the left shift and right shift modules (bits 10 downto 6), RS Register select signal (bits 15 downto 11), and bit 0 determines the difference between a logical right shift and an arithmetic right shift
i_unsigned	Input	1	Comparator Unsigned Control Singal
i_overflow	Input	1	Comparator Overflow Control Singal
i_imm_extend	Input	1	Determine whether immediate is zero or sign extended
i_STORE_FROM_MEM	Input	1	Store ALU Result or memory data to GPR Bank
i_REG_SEL	Input	2	Determine which GPR is enabled and whether the PC register is enabled
i_PC_EN	Input	1	Determine if the PC should store PC+4
i_MEM_ADDRESS_SEL	Input	1	Determine if memory address should be PC register data or ALU result
i_MEM_OUT	Input	32	Output from memory to the datapath
o_MEM_ADDRESS	Output	32	Address used to access memory
o_RT_DATA	Output	32	Data to write to memory if performing a memory write

Table 42. Datapath Connections

GPR Data Input MUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	w_ALU_RESULT
i_1	Input	32	i_MEM_OUT
i_S	Input	1	i_STORE_FROM_MEM
o_Z	Output	32	w_GPR_DATA_IN
GPR Select Signal Selector MUX4_5			
I/O	Type	Width	Connection
i_0	Input	5	0
i_1	Input	5	i_imm(15 downto 11)
i_2	Input	5	i_RT_SEL
i_3	Input	5	0
i_S	Input	2	i_REG_SEL
o_Z	Output	5	w_GPR_SEL
GPR Bank			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_data	Input	32	w_GPR_DATA_IN
i_sel	Input	5	w_GPR_SEL
o_Q00	Output	32	w_Q00
o_Q01	Output	32	w_Q01
o_Q02	Output	32	w_Q02
o_Q03	Output	32	w_Q03
o_Q04	Output	32	w_Q04
o_Q05	Output	32	w_Q05
o_Q06	Output	32	w_Q06
o_Q07	Output	32	w_Q07
o_Q08	Output	32	w_Q08
o_Q09	Output	32	w_Q09
o_Q10	Output	32	w_Q10
o_Q11	Output	32	w_Q11
o_Q12	Output	32	w_Q12
o_Q13	Output	32	w_Q13
o_Q14	Output	32	w_Q14
o_Q15	Output	32	w_Q15
o_Q16	Output	32	w_Q16
o_Q17	Output	32	w_Q17
o_Q18	Output	32	w_Q18
o_Q19	Output	32	w_Q19

Table 42 – *Continued on next page*

Table 42 – *Continued from previous page*

o_Q20	Output	32	w_Q20
o_Q21	Output	32	w_Q21
o_Q22	Output	32	w_Q22
o_Q23	Output	32	w_Q23
o_Q24	Output	32	w_Q24
o_Q25	Output	32	w_Q25
o_Q26	Output	32	w_Q26
o_Q27	Output	32	w_Q27
o_Q28	Output	32	w_Q28
o_Q29	Output	32	w_Q29
o_Q30	Output	32	w_Q30
o_Q31	Output	32	w_Q31
I/O	Type	Width	Connection
i_0	Output	32	w_Q00
i_1	Output	32	w_Q01
i_2	Output	32	w_Q02
i_3	Output	32	w_Q03
i_4	Output	32	w_Q04
i_5	Output	32	w_Q05
i_6	Output	32	w_Q06
i_7	Output	32	w_Q07
i_8	Output	32	w_Q08
i_9	Output	32	w_Q09
i_10	Output	32	w_Q10
i_11	Output	32	w_Q11
i_12	Output	32	w_Q12
i_13	Output	32	w_Q13
i_14	Output	32	w_Q14
i_15	Output	32	w_Q15
i_16	Output	32	w_Q16
i_17	Output	32	w_Q17
i_18	Output	32	w_Q18
i_19	Output	32	w_Q19
i_20	Output	32	w_Q20
i_21	Output	32	w_Q21
i_22	Output	32	w_Q22
i_23	Output	32	w_Q23
i_24	Output	32	w_Q24
i_25	Output	32	w_Q25
i_26	Output	32	w_Q26

Table 42 – *Continued on next page*

Table 42 – *Continued from previous page*

i_27	Output	32	w_Q27
i_28	Output	32	w_Q28
i_29	Output	32	w_Q29
i_30	Output	32	w_Q30
i_31	Output	32	w_Q31
i_sel	Input	5	i_RS_SEL
o_Z	Output	32	w_RS_DATA
RT Data Selector MUX32_32			
I/O	Type	Width	Connection
i_0	Output	32	w_Q00
i_1	Output	32	w_Q01
i_2	Output	32	w_Q02
i_3	Output	32	w_Q03
i_4	Output	32	w_Q04
i_5	Output	32	w_Q05
i_6	Output	32	w_Q06
i_7	Output	32	w_Q07
i_8	Output	32	w_Q08
i_9	Output	32	w_Q09
i_10	Output	32	w_Q10
i_11	Output	32	w_Q11
i_12	Output	32	w_Q12
i_13	Output	32	w_Q13
i_14	Output	32	w_Q14
i_15	Output	32	w_Q15
i_16	Output	32	w_Q16
i_17	Output	32	w_Q17
i_18	Output	32	w_Q18
i_19	Output	32	w_Q19
i_20	Output	32	w_Q20
i_21	Output	32	w_Q21
i_22	Output	32	w_Q22
i_23	Output	32	w_Q23
i_24	Output	32	w_Q24
i_25	Output	32	w_Q25
i_26	Output	32	w_Q26
i_27	Output	32	w_Q27
i_28	Output	32	w_Q28
i_29	Output	32	w_Q29
i_30	Output	32	w_Q30

Table 42 – *Continued on next page*

Table 42 – *Continued from previous page*

i_31	Output	32	w_Q31
i_sel	Input	5	i_RT_SEL
o_Z	Output	32	w_RT_DATA
RD Data Selector MUX32_32			
I/O	Type	Width	Connection
i_0	Output	32	w_Q00
i_1	Output	32	w_Q01
i_2	Output	32	w_Q02
i_3	Output	32	w_Q03
i_4	Output	32	w_Q04
i_5	Output	32	w_Q05
i_6	Output	32	w_Q06
i_7	Output	32	w_Q07
i_8	Output	32	w_Q08
i_9	Output	32	w_Q09
i_10	Output	32	w_Q10
i_11	Output	32	w_Q11
i_12	Output	32	w_Q12
i_13	Output	32	w_Q13
i_14	Output	32	w_Q14
i_15	Output	32	w_Q15
i_16	Output	32	w_Q16
i_17	Output	32	w_Q17
i_18	Output	32	w_Q18
i_19	Output	32	w_Q19
i_20	Output	32	w_Q20
i_21	Output	32	w_Q21
i_22	Output	32	w_Q22
i_23	Output	32	w_Q23
i_24	Output	32	w_Q24
i_25	Output	32	w_Q25
i_26	Output	32	w_Q26
i_27	Output	32	w_Q27
i_28	Output	32	w_Q28
i_29	Output	32	w_Q29
i_30	Output	32	w_Q30
i_31	Output	32	w_Q31
i_sel	Input	5	i_imm(15 downto 11)
o_Z	Output	32	w_RD_DATA
ALU Core			

Table 42 – *Continued on next page*

Table 42 – *Continued from previous page*

I/O	Type	Width	Connection
i_ALU_SRC_A	Input	2	i_ALU_SRC_A
i_ALU_SRC_B	Input	2	i_ALU_SRC_B
i_ALU_INV_B	Input	1	i_ALU_INV_B
i_COMP_SEL	Input	3	i_COMP_SEL
i_OVER_CTRL	Input	2	i_OVER_CTRL
i_ALU_OUTPUT	Input	4	i_ALU_OUTPUT
i_RS_DATA	Input	32	w_RS_DATA
i_RT_DATA	Input	32	w_RT_DATA
i_RD_DATA	Input	32	w_RD_DATA
i_PC	Input	32	w_PC
i_imm	Input	16	i_imm
i_unsigned	Input	1	i_unsigned
i_overflow	Input	1	i_overflow
i_imm_extend	Input	1	i_imm_extend
o_ALU_Result	Output	32	o_ALU_RESULT
PC Data Input MUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	w_PC_4
i_1	Input	32	w_ALU_RESULT
i_S	Input	1	w_PC_EN_n
o_Z	Output	32	w_PC_IN
PC Enable Based on i_REG_SEL MUX4_1			
I/O	Type	Width	Connection
i_0	Input	1	0
i_1	Input	1	0
i_2	Input	1	0
i_3	Input	1	1
i_S	Input	2	i_REG_SEL
o_Z	Output	1	w_PC_ENABLE0
PC Enable Based on i_PC_EN MUX2_1			
I/O	Type	Width	Connection
i_0	Input	1	w_PC_ENABLE0
i_1	Input	1	1
i_S	Input	1	i_PC_EN
o_Z	Output	1	w_PC_ENABLE
Invert i_PC_EN myINV			
I/O	Type	Width	Connection
i_A	Input	1	i_PC_EN
o_Z	Output	1	w_PC_EN_n

Table 42 – *Continued on next page*

Table 42 – *Continued from previous page*

PC Register Enabled_Register			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_data	Input	32	w_PC_IN
i_en	Input	1	w_PC_ENABLE
o_Q	Output	32	w_PC
PC Adder Carry_SELECT_ADDER_30_SC			
I/O	Type	Width	Connection
i_A	Input	30	w_PC(31 downto 2)
i_B	Input	30	1
i_C	Input	1	0
o_S	Output	30	w_PC_4(31 downto 2)
o_C	Output	1	OPEN - unconnected
Standalone PC Signal Assignment			
w_PC_4(1 downto 0)	Connection	2	w_PC(1 downto 0)
Memory Address Selector MUX2_32			
I/O	Type	Width	Connection
i_0	Input	32	w_PC
i_1	Input	32	w_ALU_RESULT
i_S	Input	1	i_MEM_ADDRESS_SEL
o_Z	Output	32	o_MEM_ADDRESS
Remaining Standalone Signal Assignments			
o_RT_DATA	Connection	32	w_RT_DATA

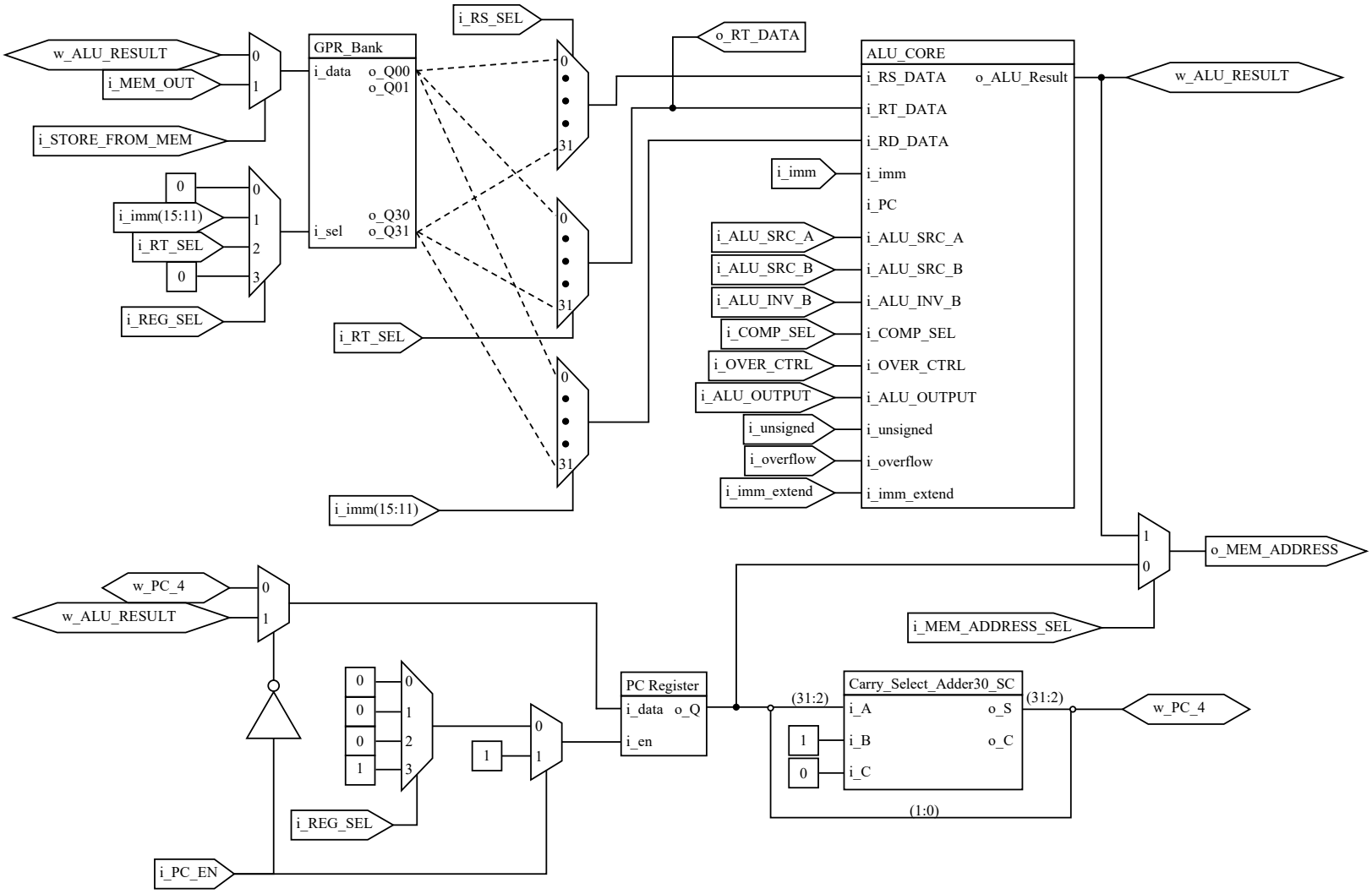

Figure 21. Datapath Schematic

2.18 Instruction Decoder

The instruction decoder takes a 32-bit instruction and determines how to set the control signals affecting the Datapath. Within each 32-bit instruction, the bits 31 downto 26 comprise the opcode which may partially or completely define the function to be performed. If the function is not completely defined by the opcode and the opcode is a 6-bit 0, the bits 5 downto 0 define the function. If the opcode corresponds to BGEZ and BLTZ functions, bit 16 determines which of these two functions is performed (bit 16 = 1 for BGEZ and bit 16 = 0 for BLTZ). Table 43 highlights the bits of each instruction that uniquely define the instruction.

Table 43. Instruction Defining Characteristics

	Opcode						RT					Function					
Instruction	31	30	29	28	27	26	20	19	18	17	16	5	4	3	2	1	0
SLL	0	0	0	0	0	0	RT					0	0	0	0	0	0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SRL	0	0	0	0	0	0	RT					0	0	0	0	1	0
SRA	0	0	0	0	0	0	RT					0	0	0	0	1	1
SLLV	0	0	0	0	0	0	RT					0	0	0	1	0	0
SRLV	0	0	0	0	0	0	RT					0	0	0	1	1	0
SRAV	0	0	0	0	0	0	RT					0	0	0	1	1	1
ADD	0	0	0	0	0	0	RT					1	0	0	0	0	0
ADDU	0	0	0	0	0	0	RT					1	0	0	0	0	1
SUB	0	0	0	0	0	0	RT					1	0	0	0	1	0
SUBU	0	0	0	0	0	0	RT					1	0	0	0	1	1
AND	0	0	0	0	0	0	RT					1	0	0	1	0	0
OR	0	0	0	0	0	0	RT					1	0	0	1	0	1
XOR	0	0	0	0	0	0	RT					1	0	0	1	1	0
NOR	0	0	0	0	0	0	RT					1	0	0	1	1	1
SLT	0	0	0	0	0	0	RT					1	0	1	0	1	0
SLTU	0	0	0	0	0	0	RT					1	0	1	0	1	1
BGEZ	0	0	0	0	0	1	0	0	0	0	1	Offset					
BLTZ	0	0	0	0	0	1	0	0	0	0	0	Offset					
BEQ	0	0	0	1	0	0	RT					Offset					
BNE	0	0	0	1	0	1	RT					Offset					
BGEZ	0	0	0	1	1	0	0	0	0	0	0	Offset					
BLTZ	0	0	0	1	1	1	0	0	0	0	0	Offset					
ADDI	0	0	1	0	0	0	RT					Immediate					
ADDIU	0	0	1	0	0	1	RT					Immediate					
SLTI	0	0	1	0	1	0	RT					Immediate					
SLTIU	0	0	1	0	1	1	RT					Immediate					
ANDI	0	0	1	1	0	0	RT					Immediate					
ORI	0	0	1	1	0	1	RT					Immediate					
XORI	0	0	1	1	1	0	RT					Immediate					
LUI	0	0	1	1	1	1	RT					Immediate					
LW	1	0	0	0	1	1	RT					Immediate					
SW	1	0	1	0	1	1	RT					Immediate					

The next step is to examine each instruction dependent Datapath control and determine its value based on the instruction. The first Datapath control to examine is

ALU_SRC_A. For nearly all of the instructions, ALU_SRC_A should be 00 to select the RS input. This is even true for the first iteration/cycle of the branching instructions. The only instructions not using the RS input are SLL, NOP, SRL, and SRA which would have ALU_SRC_A set to 01. This suggests that ALU_SRC_A(0) should be equal to the some logical combination of the operation and function codes for the SLL, NOP, SRL, and SRA instructions. It also suggests that ALU_SRC_A(1) should depend on the state of the Controller so that ALU Source A is only PC on the second Datapath iteration/cycle of a branching instruction. ALU_SRC_A(1) is therefore not set within the Instruction Decoder since it is state dependent rather than instruction dependent. ALU_SRC_A(0) is computed as shown in Equation 34. Note that there is no instruction whose opcode is 000000 and function code is 000001. The logic simplification takes advantage of this. The four instructions have the same opcode and the function code is the same for bits 5 downto 2. The four instructions only differ in bits 1 and 0. This means that ALU_SRC_A(0) is not dependent on these bits. Also note that NOP is really SLL with RT set to 0 so that the SLL is being performed on the 0 register, resulting in a 0 output, which is stored back to the 0 register.

$$
\begin{aligned}
o_ALU_SRC_A0 &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
&\quad\ \overline{i_instr26}\ \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2} \\
o_ALU_SRC_A0 &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}} \\
&\quad\ \overline{\overline{i_instr26}\ \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}} \\
o_ALU_SRC_A0 &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + \cdots} \\
&\quad\ \overline{i_instr27 + i_instr26 + i_instr5 + i_instr4 + \cdots} \\
&\quad\ \overline{i_instr3 + i_instr2} \\
w_ctrl_000 &= NAND(INV(i_instr31), INV(i_instr30)) \\
w_ctrl_001 &= NAND(INV(i_instr29), INV(i_instr28)) \\
w_ctrl_002 &= NAND(INV(i_instr27), INV(i_instr26)) \\
w_ctrl_003 &= NAND(INV(i_instr5), INV(i_instr4)) \\
w_ctrl_004 &= NAND(INV(i_instr3), INV(i_instr2)) \\
o_ALU_SRC_A0 &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002 + \cdots} \\
&\quad\ \overline{w_ctrl_003 + w_ctrl_004} \\
w_ctrl_005 &= NAND(INVw_ctrl_000), INV(w_ctrl_001)) \\
w_ctrl_006 &= NAND(INVw_ctrl_002), INV(w_ctrl_003)) \\
o_ALU_SRC_A0 &= \overline{w_ctrl_005 + w_ctrl_006 + w_ctrl_004} \\
w_ctrl_007 &= NAND(INVw_ctrl_005), INV(w_ctrl_006)) \\
o_ALU_SRC_A0 &= \overline{w_ctrl_007 + w_ctrl_004} \\
w_ctrl_008 &= NAND(INVw_ctrl_007), INV(w_ctrl_004)) \\
o_ALU_SRC_A0 &= \overline{w_ctrl_008} \\
o_ALU_SRC_A0 &= INV(w_ctrl_008)
\end{aligned}
\tag{34}
$$

The next control to be examined is ALU_SRC_B. This control selects between 0 (00), RT (01), immediate/offset (10), and the offset << 2 or 4 (11). ALU_SRC_B

will only be 11 on the second iteration/cycle of a branching instruction so this value of the signal is state dependent rather than instruction dependent. This suggests that ALU_SRC_B(0) should be 1 for instructions whose ALU B input is RT and that ALU_SRC_B(1) should be 1 for instructions whose ALU B input is the immediate/offset value. Since the instructions that have RT as their ALU B input are mutually exclusive of instructions that have immediate/offset as their ALU B input and also mutually exclusive of instructions with 0 as their ALU B input, ALU_SRC_B(0) and ALU_SRC_B(1) can be set independently of one another and ALU_SRC_B(0) is 1 for instructions whose ALU B input is RT and ALU_SRC_B(1) is 1 for instructions whose ALU B input is the immediate/offset. When both of these are 0, ALU_SRC_B(0) is 0. When the offset << 2 or 4 need to be selected, an overriding state dependent control signal will force ALU_SRC_B to 11. Equation 35 shows the logic for determining ALU_SRC_B(0) and Equation 36 shows the logic for determining ALU_SRC_B(1). All of the instructions with 000000 as an opcode use RT as the ALU B input. The only other instructions that use RT as the ALU B input are the BEQ and BNE instructions with opcodes that start with 00010. The instructions that use immediate/offset as the ALU B input have opcodes that start with 001 or 1. Since no other instructions have a 1 in the bit 31 or 1 in bit 29, the logic for selecting the immediate/offset is greatly simplified.

$$
\begin{aligned}
o_ALU_SRC_B(0) &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29} \cdot \cdot \\
&\quad (\overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} + i_instr28\ \overline{i_instr27}) \\
o_ALU_SRC_B(0) &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29} \cdot \cdot} \\
&\quad \overline{(\overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} + i_instr28\ \overline{i_instr27})} \\
o_ALU_SRC_B(0) &= \overline{i_instr31 + i_instr30 + i_instr29 + (i_instr28 + \cdots} \\
&\quad \overline{i_instr27 + i_instr26)(\overline{i_instr28} + i_instr27)} \\
o_ALU_SRC_B(0) &= \overline{w_ctrl_000 + i_instr29 + (i_instr28 + \cdots} \\
&\quad \overline{w_ctrl_002)(\overline{i_instr28} + i_instr27)} \\
w_ctrl_009 &= NAND(INV(w_ctrl_000), INV(i_instr29)) \\
w_ctrl_010 &= NAND(INV(i_instr28), INV(w_ctrl_002)) \\
w_ctrl_011 &= NAND(i_instr28, INV(i_instr27)) \\
o_ALU_SRC_B(0) &= \overline{w_ctrl_009 + w_ctrl_010\ w_ctrl_011} \\
w_ctrl_012 &= NAND(w_ctrl_010, w_ctrl_011) \\
o_ALU_SRC_B(0) &= \overline{w_ctrl_009 + \overline{w_ctrl_012}} \\
w_ctrl_013 &= NAND(INV(w_ctrl_009), w_ctrl_012) \\
o_ALU_SRC_B(0) &= \overline{w_ctrl_013} \\
o_ALU_SRC_B(0) &= INV(w_ctrl_013)
\end{aligned}
\tag{35}
$$

$$
\begin{aligned}
o_ALU_SRC_B(1) &= i_instr31 + i_instr29 \\
o_ALU_SRC_B(1) &= \overline{\overline{i_instr31}\ \overline{i_instr29}} \\
o_ALU_SRC_B(1) &= NAND(INV(i_instr31), INV(i_instr29))
\end{aligned}
\tag{36}
$$

The next control signal is ALU_INV_B. This signal is 1 for instructions that invert the B input and 0 for those that do not. The instructions that invert the B input are SUB, SUBU, SLT, SLTU, BEQ, BNE, SLTI, and SLTIU. The ALU_INV_B signal is also overridden on the second cycle of branch instructions so that ALU_INV_B is

forced to be 0. This is a state dependent override and does not factor into the logic here.

$$
\begin{aligned}
o_ALU_INV_B = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
& \overline{i_instr26}\ i_instr5\ \overline{i_instr4} \\
& \overline{i_instr3}\ \overline{i_instr2}\ i_instr1 + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
& \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3 + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27} + \\
& \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ i_instr27 \\
o_ALU_INV_B = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
& \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ (i_instr3 + \cdots \\
& \overline{i_instr3}\ \overline{i_instr2}\ i_instr1) + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27} + \\
& \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ i_instr27 \\
o_ALU_INV_B = {} & \overline{i_instr31}\ \overline{i_instr30}\ [i_instr29\ \overline{i_instr28}\ i_instr27 + \cdots \\
& \overline{i_instr29}\ \overline{i_instr27}\ (i_instr28 + \cdots \\
& \overline{i_instr28}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4} \\
& \{i_instr3 + \overline{i_instr3}\ \overline{i_instr2}\ i_instr1\})] \\
o_ALU_INV_B = {} & \overline{\overline{\overline{i_instr31}\ \overline{i_instr30}\ [i_instr29\ \overline{i_instr28}\ i_instr27 + \cdots}} \\
& \overline{\overline{\overline{i_instr29}\ \overline{i_instr27}\ (i_instr28 + \cdots}} \\
& \overline{\overline{\overline{i_instr28}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}}} \\
& \overline{\overline{\{i_instr3 + \overline{i_instr3}\ \overline{i_instr2}\ i_instr1\})]}} \\
o_ALU_INV_B = {} & \overline{i_instr31 + i_instr30 + \overline{[\cdots]}} \\
o_ALU_INV_B = {} & \overline{w_ctrl_000 + [\overline{i_instr29} + i_instr28 + \overline{i_instr27}]} \\
& \overline{[i_instr29 + i_instr27 + \overline{(\cdots)}]}
\end{aligned}
\tag{37}
$$

$$w_ctrl_014 = NAND(INV(i_instr29), INV(i_instr27))$$

$$w_ctrl_029 = NAND(i_instr29, i_instr27)$$

$$
\begin{aligned}
o_ALU_INV_B &= \overline{w_ctrl_000 + [w_ctrl_029 + i_instr28][w_ctrl_014 + \overline{(\cdots)}]} \\
w_ctrl_084 &= NAND(INV(w_ctrl_029), INV(i_instr28)) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \overline{(\cdots)}]} \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (i_instr28 + i_instr26 + \overline{i_instr5} + \cdots} \\
&\quad \overline{i_instr4 + \overline{\{\cdots\}})]} \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (i_instr26 + \overline{i_instr5} + i_instr4 + \overline{\{\cdots\}})]} \\
w_ctrl_018 &= NAND(INV(i_instr26), i_instr5) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_018 + i_instr4 + \overline{\{\cdots\}})]} \\
w_ctrl_020 &= NAND(INV(w_ctrl_018), INV(i_instr4)) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_020 + \overline{\{\cdots\}})]} \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_020 + \overline{i_instr3}\{i_instr3 + \cdots} \\
&\quad \overline{i_instr2 + \overline{i_instr1}\})]} \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_020 + \overline{i_instr3}\{i_instr2 + \cdots} \\
&\quad \overline{\overline{i_instr1}\})]} \\
w_ctrl_110 &= NAND(INV(i_instr2), i_instr1) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_020 + \overline{i_instr3}\ w_ctrl_110)]} \\
w_ctrl_111 &= NAND(INV(i_instr3), w_ctrl_110)
\end{aligned}
\tag{38}
$$

$$
\begin{aligned}
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ (w_ctrl_020 + \overline{w_ctrl_111})]} \\
w_ctrl_112 &= NAND(INV(w_ctrl_020), w_ctrl_111) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \cdots} \\
&\quad \overline{\overline{i_instr28}\ w_ctrl_0112]} \\
w_ctrl_113 &= NAND(INV(i_instr28), w_ctrl_112) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084[w_ctrl_014 + \overline{w_ctrl_113}]} \\
w_ctrl_114 &= NAND(INV(w_ctrl_014), w_ctrl_113) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + w_ctrl_084\ w_ctrl_114} \\
w_ctrl_115 &= NAND(w_ctrl_084, w_ctrl_114) \\
o_ALU_INV_B &= \overline{w_ctrl_000 + \overline{w_ctrl_115}} \\
w_ctrl_116 &= NAND(INV(w_ctrl_000), , w_ctrl_115) \\
o_ALU_INV_B &= \overline{w_ctrl_116} \\
o_ALU_INV_B &= INV(w_ctrl_116)
\end{aligned}
\tag{39}
$$

The next control signal to implement is the comparison select signal COMP_SEL. Signals are created for LTZ, EQ, NE, LEZ, and GTZ and these signals are encoded as a 3-bit signal called COMP_SEL for the ALU Core. Note that no signal is created for GEZ because the six individual comparison signals are mutually exclusive such that if the first five signals are 0, it can be assumed the GEZ is 1. If all six signals should be 0 because no comparison is being performed, the ALU Output select signal is such that the output of the comparator is ignored anyway. Logic is implemented to determine LTZ, EQ, NE, LEZ, and GTZ and an external 6 to 3 encoder is created that encodes LTZ as 001, EQ as 010, NE as 011, and GTZ as 100 as required by the ALU Core. All of the signals being 0 would cause the COMP_SEL signal to be

000 for GEZ. This 6 to 3 encoder is part of a higher level module connected to the outputs from the Instruction Decoder Module.

LTZ is utilized by the SLT, SLTU, BLTZ, SLTI, and SLTIU instructions. The logic to determine LTZ is shown in Equation 40. Note that since the comparator output is masked by the ALU output select control signal when the opcode is 000000 and the instruction is not SLT or SLTU, the function bits of the SLT and SLTU instructions can be ignored.

$$
\begin{aligned}
o_LTZ &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} + \cdots \\
&\quad \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ i_instr26 \\
&\quad \overline{i_instr16} + \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ i_instr27 \\
&\quad \overline{i_instr26} \\
o_LTZ &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ [i_instr29\ i_instr27 + \cdots \\
&\quad \overline{i_instr29}\ \overline{i_instr27}\ (\overline{i_instr26} + i_instr26\ \overline{i_instr16})] \\
o_LTZ &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ [i_instr29\ i_instr27 + \cdots} \\
&\quad \overline{\overline{i_instr29}\ \overline{i_instr27}\ (\overline{i_instr26} + i_instr26\ \overline{i_instr16})]} \\
o_LTZ &= \overline{i_instr31 + i_instr30 + i_instr28 + \overline{[\cdots]}} \\
o_LTZ &= \overline{w_ctrl_000 + i_instr28 + \overline{[\cdots]}} \\
w_ctrl_028 &= NAND(INV(w_ctrl_000), INV(i_instr28)) \\
o_LTZ &= \overline{w_ctrl_028 + \overline{[\cdots]}} \\
o_LTZ &= \overline{w_ctrl_028 + [\overline{i_instr29} + \overline{i_instr27}][i_instr29 + \cdots} \\
&\quad \overline{i_instr27 + \overline{(\cdots)}]} \\
o_LTZ &= \overline{w_ctrl_028 + w_ctrl_029[w_ctrl_014 + \overline{(\cdots)}]} \\
o_LTZ &= \overline{w_ctrl_028 + w_ctrl_029[w_ctrl_014 + \cdots} \\
&\quad \overline{i_instr26\ (\overline{i_instr26} + i_instr16)]} \\
o_LTZ &= \overline{w_ctrl_028 + w_ctrl_029[w_ctrl_014 + i_instr26\ i_instr16]} \\
w_ctrl_030 &= NAND(i_instr26, i_instr16) \\
o_LTZ &= \overline{w_ctrl_028 + w_ctrl_029[w_ctrl_014 + \overline{w_ctrl_030}]} \\
w_ctrl_031 &= NAND(INV(w_ctrl_014), w_ctrl_030) \\
o_LTZ &= \overline{w_ctrl_028 + w_ctrl_029\ w_ctrl_031} \\
w_ctrl_032 &= NAND(w_ctrl_029, w_ctrl_031) \\
o_LTZ &= \overline{w_ctrl_028 + \overline{w_ctrl_032}}
\end{aligned}
\tag{40}
$$

$$
\begin{aligned}
&w_ctrl_033 = NAND(INV(w_ctrl_028), w_ctrl_032) \\
&o_LTZ = \overline{w_ctrl_033} \\
&o_LTZ = INV(w_ctrl_033)
\end{aligned}
\tag{41}
$$

EQ is only utilized by the BEQ instruction and the logic to determine EQ is shown here:

$$
\begin{aligned}
&o_EQ = \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27}\ \overline{i_instr26} \\
&o_EQ = \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27}\ \overline{i_instr26}} \\
&o_EQ = \overline{i_instr31 + i_instr30 + i_instr29 + \overline{i_instr28} + \cdots} \\
&\qquad \overline{i_instr27 + i_instr26} \\
&o_EQ = \overline{w_ctrl_000 + w_ctrl_011 + i_instr29 + i_instr26} \\
&w_ctrl_015 = NAND(INV(w_ctrl_011), INV(i_instr29)) \\
&o_EQ = \overline{w_ctrl_000 + w_ctrl_015 + i_instr26} \\
&w_ctrl_034 = NAND(INV(w_ctrl_000), INV(w_ctrl_015)) \\
&o_EQ = \overline{w_ctrl_034 + i_instr26} \\
&w_ctrl_035 = NAND(INV(w_ctrl_034), INV(i_instr26)) \\
&o_EQ = \overline{w_ctrl_035} \\
&o_EQ = INV(w_ctrl_035)
\end{aligned}
\tag{42}
$$

NE is only utilized by the BNE instruction and the logic to determine NE is shown here:

$$
\begin{aligned}
o_NE &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27}\ i_instr26 \\
o_NE &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ \overline{i_instr27}\ i_instr26} \\
o_NE &= \overline{i_instr31 + i_instr30 + i_instr29 + \overline{i_instr28} + \cdots} \\
&\quad \overline{i_instr27 + \overline{i_instr26}} \\
o_NE &= \overline{w_ctrl_000 + w_ctrl_011 + i_instr29 + \overline{i_instr26}} \\
o_NE &= \overline{w_ctrl_000 + w_ctrl_015 + \overline{i_instr26}} \\
o_NE &= \overline{w_ctrl_034 + \overline{i_instr26}} \\
w_ctrl_036 &= NAND(INV(w_ctrl_034), _instr26) \\
o_NE &= \overline{w_ctrl_036} \\
o_NE &= INV(w_ctrl_036)
\end{aligned}
\tag{43}
$$

LEZ is only utilized by the BLEZ instruction and the logic to determine LEZ is shown here:

$$
\begin{aligned}
o_LEZ &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ i_instr27\ \overline{i_instr26} \\
o_LEZ &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ i_instr27\ \overline{i_instr26}} \\
o_LEZ &= \overline{i_instr31 + i_instr30 + i_instr29 + \overline{i_instr28} + \cdots} \\
&\quad \overline{\overline{i_instr27} + i_instr26} \\
w_ctrl_037 &= NAND(INV(i_instr29), i_instr28) \\
w_ctrl_038 &= NAND(i_instr27, INV(i_instr26)) \\
o_LEZ &= \overline{w_ctrl_000 + w_ctrl_037 + w_ctrl_038} \\
w_ctrl_039 &= NAND(INV(w_ctrl_000), INV(w_ctrl_037)) \\
o_LEZ &= \overline{w_ctrl_039 + w_ctrl_038} \\
w_ctrl_040 &= NAND(INV(w_ctrl_039), INV(w_ctrl_038)) \\
o_LEZ &= \overline{w_ctrl_040} \\
o_LEZ &= INV(w_ctrl_040)
\end{aligned}
\tag{44}
$$

GTZ is only utilized by the BGTZ instruction and the logic to determine GTZ is shown here:

$$
\begin{aligned}
o_GTZ &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ i_instr27\ i_instr26 \\
o_GTZ &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ i_instr27\ i_instr26} \\
o_GTZ &= \overline{i_instr31 + i_instr30 + i_instr29 + \overline{i_instr28} + \cdots} \\
&\quad \overline{\overline{i_instr27} + \overline{i_instr26}} \\
w_ctrl_041 &= NAND(i_instr27, i_instr26) \\
o_GTZ &= \overline{w_ctrl_000 + w_ctrl_037 + w_ctrl_041} \\
o_GTZ &= \overline{w_ctrl_039 + w_ctrl_041} \\
w_ctrl_042 &= NAND(INV(w_ctrl_039), INV(w_ctrl_041)) \\
o_GTZ &= \overline{w_ctrl_042} \\
o_GTZ &= INV(w_ctrl_042)
\end{aligned}
\tag{45}
$$

The next control signal is the ALU_OUTPUT control signal responsible for selecting which ALU Core submodule will have its output selected. Just as for the COMP_SEL signal, a signal was created for RIGHT_SHIFT, ADDER, AND, OR, XOR, NOR, COMPARE, and LUI, but not LEFT_SHIFT. These signals were encoded those signals as a 4-bit signal called ALU_OUTPUT. If these eight signals were all 0, then the ninth signal (LEFT_SHIFT) could be assumed to be 1. Logic was implemented to determine RIGHT_SHIFT, ADDER, AND, OR, XOR, NOR, COMPARE, and LUI and an external nine to four encoder was created that encoded RIGHT_SHIFT as 0001, ADDER as 0010, AND as 0011, OR as 0100, XOR as 0101, NOR as 0110, COMPARE as 0111, and LUI as 1000 as required by the ALU Core. All of the signals being 0 would cause the ALU_OUTPUT signal to be 0000 for LEFT_SHIFT. This 9 to 4 encoder is part of a higher level module connected to the outputs from the Instruction Decoder module. The ALU_OUTPUT can also be overridden during

the second cycle of branch operations. In this case, ALU_OUTPUT is forced to be 0010, or the adder output. Since this override is state dependent and not instruction dependent, this does not factor into the logic here.

RIGHT_SHIFT is used by the SRL, SRA, SRLV, and SRAV instructions. The logic to determine RIGHT_SHIFT is shown in Equation 46. Notice that all the bits in the opcode and function code of the instruction are identical for these four instructions except bits 2 and 0. Since all four possible combinations of bit 2 and 0 are exercised for the four instructions, they can be ignored for determining when a right shift occurs according to DeMorgan's Laws.

$$
\begin{aligned}
o_RIGHT_SHIFT &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
&\quad\ \overline{i_instr26}\ \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr1 \\
o_RIGHT_SHIFT &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}} \\
&\quad\ \overline{\overline{i_instr26}\ \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr1} \\
o_RIGHT_SHIFT &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + \cdots} \\
&\quad\ \overline{i_instr27 + i_instr26 + i_instr5 + i_instr4 + \cdots} \\
&\quad\ \overline{i_instr3 + \overline{i_instr1}} \\
w_ctrl_046 &= NAND(INV(i_instr3), i_instr1) \\
o_RIGHT_SHIFT &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002 + \cdots} \\
&\quad\ \overline{w_ctrl_003 + w_ctrl_046} \\
o_RIGHT_SHIFT &= \overline{w_ctrl_005 + w_ctrl_006 + w_ctrl_046} \\
o_RIGHT_SHIFT &= \overline{w_ctrl_007 + w_ctrl_046} \\
w_ctrl_047 &= NAND(INV(w_ctrl_007), INV(w_ctrl_046)) \\
o_RIGHT_SHIFT &= \overline{w_ctrl_047} \\
o_RIGHT_SHIFT &= INV(w_ctrl_047)
\end{aligned}
\tag{46}
$$

ADDER is used by the ADD, ADDU, SUB, SUBU, ADDI, ADDIU, LW, and SW

instructions. The logic to determine ADDER is shown in Equation 47. For the ADD, ADDU, SUB, and SUBU functions, all of the opcode and instruction bits are identical except bits 1 and 0. Since all four possible combinations of bit 1 and 0 are exercised for these four instructions, they can be ignored according to DeMorgan's Laws. Additionally, for the ADDI and ADDIU functions, the opcodes are identical except for bit 26, and therefore this bit is ignored. Finally, since the only instructions that have a 1 in bit 31 are the LW and SW instructions, bit 31 only needs to be examined for these instructions.

$$
\begin{aligned}
o_ADDER &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
&\quad \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2} + \cdots \\
&\quad \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ \overline{i_instr27} + \cdots \\
&\quad i_instr31 \\
o_ADDER &= i_instr31 + \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ \overline{i_instr27} \\
&\quad [i_instr29 + \overline{i_instr29}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4} \\
&\quad \overline{i_instr3}\ \overline{i_instr2}] \\
o_ADDER &= \overline{\overline{i_instr31 + \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ \overline{i_instr27}}} \\
&\quad \overline{\overline{[i_instr29 + \overline{i_instr29}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}}} \\
&\quad \overline{\overline{\overline{i_instr3}\ \overline{i_instr2}]}} \\
o_ADDER &= \overline{\overline{i_instr31}\ \overline{[\cdots]}} \\
o_ADDER &= \overline{\overline{i_instr31}\ [i_instr31 + i_instr30 + i_instr28 + \cdots} \\
&\quad \overline{i_instr27 + \overline{(\cdots)}]} \qquad (47) \\
o_ADDER &= \overline{\overline{i_instr31}\ [i_instr30 + i_instr28 + i_instr27 + \overline{(\cdots)}]} \\
o_ADDER &= \overline{\overline{i_instr31}\ [i_instr30 + i_instr28 + i_instr27 + \cdots} \\
&\quad \overline{\overline{i_instr29}\ (i_instr29 + i_instr26 + \overline{i_instr5} + \cdots} \\
&\quad \overline{i_instr4 + i_instr3 + i_instr2)]} \\
o_ADDER &= \overline{\overline{i_instr31}\ [i_instr30 + i_instr28 + i_instr27 + \cdots} \\
&\quad \overline{\overline{i_instr29}\ (i_instr26 + \overline{i_instr5} + \cdots} \\
&\quad \overline{i_instr4 + i_instr3 + i_instr2)]} \\
w_ctrl_048 &= NAND(INV(i_instr30), INV(i_instr27)) \\
o_ADDER &= \overline{\overline{i_instr31}\ [w_ctrl_048 + i_instr28 + \cdots} \\
&\quad \overline{\overline{i_instr29}\ (w_ctrl_018 + i_instr4 + w_ctrl_004)]} \\
w_ctrl_051 &= NAND(INV(i_instr28), INV(w_ctrl_048) \\
o_ADDER &= \overline{\overline{i_instr31}\ [w_ctrl_051 + \overline{i_instr29}\ (w_ctrl_020 + w_ctrl_004)]}
\end{aligned}
$$

$$
\begin{aligned}
w_ctrl_049 &= NAND(INV(w_ctrl_020), INV(w_ctrl_004) \\
o_ADDER &= \overline{\overline{i_instr31}\ [w_ctrl_051 + \overline{i_instr29}\ w_ctrl_049]} \\
w_ctrl_050 &= NAND(INV(i_instr29), w_ctrl_049 \\
o_ADDER &= \overline{\overline{i_instr31}\ [w_ctrl_051 + \overline{w_ctrl_050}]} \\
w_ctrl_052 &= NAND(INV(w_ctrl_051), w_ctrl_050) \\
o_ADDER &= \overline{\overline{i_instr31}\ w_ctrl_052} \\
o_ADDER &= NAND(INV(i_instr31), w_ctrl_052)
\end{aligned}
\tag{48}
$$

AND is used by the AND and ANDI instructions. The logic to determine AND is shown here:

$$
\begin{aligned}
o_AND &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ \overline{i_instr0} + \cdots \\
&\quad \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ \overline{i_instr27}\ \overline{i_instr26} \\
o_AND &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad [i_instr29\ i_instr28 + \overline{i_instr29}\ \overline{i_instr28}\ i_instr5\ \overline{i_instr4} \\
&\quad \overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ \overline{i_instr0}] \\
o_AND &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr27}\ \overline{i_instr26}} \\
&\quad \overline{[i_instr29\ i_instr28 + \overline{i_instr29}\ \overline{i_instr28}\ i_instr5\ \overline{i_instr4}} \\
&\quad \overline{\overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ \overline{i_instr0}]} \\
o_AND &= \overline{i_instr31 + i_instr30 + i_instr27 + i_instr26 + \overline{[\cdots]}} \\
o_AND &= \overline{w_ctrl_000 + w_ctrl_002 + \overline{[\cdots]}} \\
w_ctrl_053 &= NAND(INV(w_ctrl_000), INV(w_ctrl_002)) \\
o_AND &= \overline{w_ctrl_053 + \overline{[\cdots]}} \\
o_AND &= \overline{w_ctrl_053 + [\overline{i_instr29} + \overline{i_instr28}]} \\
&\quad \overline{[i_instr29 + i_instr28 + \overline{i_instr5} + i_instr4 + i_instr3 + \cdots} \\
&\quad \overline{\overline{i_instr2} + i_instr1 + i_instr0]} \\
w_ctrl_043 &= NAND(INV(i_instr1), INV(i_instr0)) \\
w_ctrl_054 &= NAND(i_instr29, i_instr28) \\
w_ctrl_055 &= NAND(i_instr5, INV(i_instr4)) \\
w_ctrl_056 &= NAND(INV(i_instr3), i_instr2) \\
o_AND &= \overline{w_ctrl_053 + w_ctrl_054\ [w_ctrl_001 + w_ctrl_055 + \cdots} \\
&\quad \overline{w_ctrl_056 + w_ctrl_043]} \\
w_ctrl_057 &= NAND(INV(w_ctrl_001), INV(w_ctrl_055)) \\
w_ctrl_058 &= NAND(INV(w_ctrl_056), INV(w_ctrl_043)) \\
o_AND &= \overline{w_ctrl_053 + w_ctrl_054\ [w_ctrl_057 + w_ctrl_058]}
\end{aligned}
\tag{49}
$$

$$
\begin{aligned}
&w_ctrl_059 = NAND(INV(w_ctrl_057), INV(w_ctrl_058)) \\
&o_AND = \overline{w_ctrl_053 + w_ctrl_054\ w_ctrl_059} \\
&w_ctrl_060 = NAND(w_ctrl_054, w_ctrl_059) \\
&o_AND = \overline{w_ctrl_053 + \overline{w_ctrl_060}} \\
&w_ctrl_061 = NAND(INV(w_ctrl_053), w_ctrl_060) \\
&o_AND = \overline{w_ctrl_061} \\
&o_AND = INV(w_ctrl_061)
\end{aligned}
\tag{50}
$$

OR is used by the OR and ORI instructions. The logic to determine OR is shown here:

$$
\begin{aligned}
o_OR &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ i_instr0 + \cdots \\
&\quad \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ \overline{i_instr27}\ i_instr26 \\
o_OR &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr27} \\
&\quad [i_instr29\ i_instr28\ i_instr26 + \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr26} \\
&\quad i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ i_instr0] \\
o_OR &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr27}} \\
&\quad \overline{[i_instr29\ i_instr28\ i_instr26 + \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr26}} \\
&\quad \overline{i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr1}\ i_instr0]} \\
o_OR &= \overline{i_instr31 + i_instr30 + i_instr27 + \overline{[\cdots]}} \\
o_OR &= \overline{w_ctrl_000 + i_instr27 + \overline{[\cdots]}} \\
w_ctrl_063 &= NAND(INV(w_ctrl_000), INV(i_instr27)) \\
o_OR &= \overline{w_ctrl_063 + \overline{[\cdots]}} \\
o_OR &= \overline{w_ctrl_063 + [\overline{i_instr29} + \overline{i_instr28} + \overline{i_instr26}]} \\
&\quad \overline{[i_instr29 + i_instr28 + i_instr26 + \overline{i_instr5} + i_instr4 + \cdots} \\
&\quad \overline{i_instr3 + \overline{i_instr2} + i_instr1 + \overline{i_instr0}]} \\
w_ctrl_062 &= NAND(INV(i_instr1), i_instr0) \\
o_OR &= \overline{w_ctrl_063 + [w_ctrl_054 + \overline{i_instr26}]} \\
&\quad \overline{[w_ctrl_001 + i_instr26 + w_ctrl_055 + w_ctrl_056 + w_ctrl_062]} \\
w_ctrl_064 &= NAND(INV(w_ctrl_054), i_instr26) \\
w_ctrl_065 &= NAND(INV(w_ctrl_056), INV(w_ctrl_062)) \\
o_OR &= \overline{w_ctrl_063 + w_ctrl_064\ [w_ctrl_057 + i_instr26 + w_ctrl_065]} \\
w_ctrl_066 &= NAND(INV(w_ctrl_057), INV(i_instr26)) \\
o_OR &= \overline{w_ctrl_063 + w_ctrl_064\ [w_ctrl_066 + w_ctrl_065]} \\
w_ctrl_067 &= NAND(INV(w_ctrl_066), INV(w_ctrl_065))
\end{aligned}
\tag{51}
$$

$$
\begin{aligned}
o_OR &= \overline{w_ctrl_063 + w_ctrl_064\ w_ctrl_067} \\
w_ctrl_068 &= NAND(w_ctrl_064, w_ctrl_067) \\
o_OR &= \overline{w_ctrl_063 + \overline{w_ctrl_068}} \\
w_ctrl_069 &= NAND(INV(w_ctrl_063), w_ctrl_068) \\
o_OR &= \overline{w_ctrl_069} \\
o_OR &= INV(w_ctrl_069)
\end{aligned}
\tag{52}
$$

XOR is used by the XOR and XORI instructions. The logic to determine XOR is shown here:

$$
\begin{aligned}
o_XOR &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ i_instr1\ \overline{i_instr0} + \cdots \\
&\quad \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ i_instr27\ \overline{i_instr26} \\
o_XOR &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr26}\ [i_instr29\ i_instr28\ i_instr27 + \cdots \\
&\quad \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ i_instr5\ \overline{i_instr4}\ \overline{i_instr3} \\
&\quad i_instr2\ i_instr1\ \overline{i_instr0}] \\
o_XOR &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr26}\ [i_instr29\ i_instr28\ i_instr27 + \cdots} \\
&\quad \overline{\overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ i_instr5\ \overline{i_instr4}\ \overline{i_instr3}} \\
&\quad \overline{\overline{i_instr2}\ i_instr1\ \overline{i_instr0}]} \\
o_XOR &= \overline{i_instr31 + i_instr30 + i_instr26 + \overline{[\cdots]}} \\
o_XOR &= \overline{w_ctrl_000 + i_instr26 + \overline{[\cdots]}} \\
w_ctrl_071 &= NAND(INV(w_ctrl_000), INV(i_instr26)) \\
o_XOR &= \overline{w_ctrl_071 + \overline{[\cdots]}} \\
o_XOR &= \overline{w_ctrl_071 + [\overline{i_instr29} + \overline{i_instr28} + \overline{i_instr27}][i_instr29 + \cdots} \\
&\quad \overline{i_instr28 + i_instr27 + \overline{i_instr5} + i_instr4 + i_instr3 + \cdots} \\
&\quad \overline{\overline{i_instr2} + \overline{i_instr1} + i_instr0]} \\
w_ctrl_070 &= NAND(i_instr1, INV(i_instr0)) \\
o_XOR &= \overline{w_ctrl_071 + [w_ctrl_054 + \overline{i_instr27}][w_ctrl_001 + i_instr27} \\
&\quad \overline{+w_ctrl_055 + w_ctrl_056 + w_ctrl_070]} \\
w_ctrl_072 &= NAND(INV(w_ctrl_054), i_instr27) \\
w_ctrl_073 &= NAND(INV(w_ctrl_056), INV(w_ctrl_070)) \\
o_XOR &= \overline{w_ctrl_071 + w_ctrl_072[w_ctrl_057 + i_instr27 + w_ctrl_073]} \\
w_ctrl_074 &= NAND(INV(w_ctrl_057), INV(w_ctrl_073)) \\
o_XOR &= \overline{w_ctrl_071 + w_ctrl_072[w_ctrl_074 + i_instr27]} \\
w_ctrl_148 &= NAND(INV(w_ctrl_074), INV(i_instr27))
\end{aligned}
\tag{53}
$$

$$
\begin{aligned}
o_XOR &= \overline{w_ctrl_071 + w_ctrl_072\ w_ctrl_148} \\
w_ctrl_75 &= NAND(w_ctrl_072, w_ctrl_148) \\
o_XOR &= \overline{w_ctrl_071 + \overline{w_ctrl_075}} \\
w_ctrl_76 &= NAND(INV(w_ctrl_071), w_ctrl_075) \\
o_XOR &= \overline{w_ctrl_076} \\
o_XOR &= INV(w_ctrl_076)
\end{aligned}
\tag{54}
$$

NOR is only used by the NOR instruction and the logic to determine NOR is shown here:

$$
\begin{aligned}
o_NOR &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad\ i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ i_instr1\ i_instr0 \\
o_NOR &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26}} \\
&\quad\ \overline{i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ i_instr1\ i_instr0} \\
o_NOR &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + i_instr27 + \cdots} \\
&\quad\ \overline{i_instr26 + i_instr5 + i_instr4 + i_instr3 + \overline{i_instr2} + \cdots} \\
&\quad\ \overline{\overline{i_instr1} + \overline{i_instr0}} \\
w_ctrl_077 &= NAND(i_instr1, i_instr0) \\
o_NOR &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002 + w_ctrl_055 + \cdots} \\
&\quad\ \overline{w_ctrl_056 + w_ctrl_077} \\
w_ctrl_078 &= NAND(INV(w_ctrl_002), INV(w_ctrl_055)) \\
w_ctrl_079 &= NAND(INV(w_ctrl_056), INV(w_ctrl_077)) \\
o_NOR &= \overline{w_ctrl_005 + w_ctrl_078 + w_ctrl_079} \\
w_ctrl_080 &= NAND(INV(w_ctrl_005), INV(w_ctrl_078)) \\
o_NOR &= \overline{w_ctrl_080 + w_ctrl_079} \\
w_ctrl_081 &= NAND(INV(w_ctrl_080), INV(w_ctrl_079)) \\
o_NOR &= \overline{w_ctrl_081} \\
o_NOR &= INV(w_ctrl_081)
\end{aligned}
\tag{55}
$$

COMPARE is used by the SLT, SLTU, SLTI, and SLTIU instructions. It causes the ALU Core to output a 32-bit 1 if the comparator LTZ output is 1 and a 32-bit 0 if the comparator LTZ output is 0. The value of the comparator LTZ output determines the value of w_COMP in the ALU Core. COMPARE simply selects w_COMP as the ALU Core output. The logic to determine COMPARE is shown here:

$$
\begin{aligned}
o_COMPARE = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
& \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3\ \overline{i_instr2} \\
& i_instr1 + \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28} \\
& i_instr27 \\
o_COMPARE = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ [i_instr29\ i_instr27 + \cdots \\
& \overline{i_instr29}\ \overline{i_instr27}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3 \\
& \overline{i_instr2}\ i_instr1] \\
o_COMPARE = {} & \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}\ [i_instr29\ i_instr27 + \cdots} \\
& \overline{\overline{i_instr29}\ \overline{i_instr27}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3} \\
& \overline{\overline{i_instr2}\ i_instr1]} \\
o_COMPARE = {} & \overline{i_instr31 + i_instr30 + i_instr28 + \overline{[\cdots]}} \\
o_COMPARE = {} & \overline{w_ctrl_000 + i_instr28 + \overline{[\cdots]}} \\
o_COMPARE = {} & \overline{w_ctrl_028 + \overline{[\cdots]}} \\
o_COMPARE = {} & \overline{w_ctrl_028 + [\overline{i_instr29} + \overline{i_instr27}][i_instr29 + \cdots} \\
& \overline{i_instr27 + i_instr26 v\overline{instr5} + i_instr4 + \cdots} \\
& \overline{\overline{i_instr3} + instr2 + \overline{instr1}]}
\end{aligned}
\tag{56}
$$

$$
w_ctrl_085 = NAND(i_instr3, INV(i_instr2))
$$

$$
w_ctrl_087 = NAND(INV(i_instr26), i_instr1)
$$

$$
\begin{aligned}
o_COMPARE = {} & \overline{w_ctrl_028 + w_ctrl_029\ [w_ctrl_014 + w_ctrl_055 + \cdots} \\
& \overline{w_ctrl_085 + w_ctrl_087]}
\end{aligned}
$$

$$
w_ctrl_086 = NAND(INV(w_ctrl_055), INV(w_ctrl_085))
$$

$$
w_ctrl_106 = NAND(INV(w_ctrl_014), INV(w_ctrl_087))
$$

$$
o_COMPARE = \overline{w_ctrl_028 + w_ctrl_029\ [w_ctrl_086 + w_ctrl_106]}
$$

$$
w_ctrl_107 = NAND(INV(w_ctrl_086), INV(w_ctrl_106))
$$

$$
o_COMPARE = \overline{w_ctrl_028 + w_ctrl_029\ w_ctrl_107}
$$

$$
\begin{aligned}
&w_ctrl_108 = NAND(w_ctrl_029, w_ctrl_107) \\
&o_COMPARE = \overline{w_ctrl_028 + \overline{w_ctrl_108}} \\
&w_ctrl_109 = NAND(INV(w_ctrl_028), w_ctrl_108) \\
&o_COMPARE = \overline{w_ctrl_109} \\
&o_COMPARE = INV(w_ctrl_109)
\end{aligned}
\tag{57}
$$

LUI is only used by the LUI instruction and the logic to determine LUI is shown here:

$$
\begin{aligned}
&o_LUI = \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ i_instr27\ i_instr26 \\
&o_LUI = \overline{\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ i_instr27\ i_instr26} \\
&o_LUI = \overline{i_instr31 + i_instr30 + \overline{i_instr29} + \overline{i_instr28} + \cdots} \\
&\qquad \overline{\overline{i_instr27} + \overline{i_instr26}} \\
&o_LUI = \overline{w_ctrl_000 + w_ctrl_054 + w_ctrl_041} \\
&w_ctrl_082 = NAND(INV(w_ctrl_000), INV(w_ctrl_054)) \\
&o_LUI = \overline{w_ctrl_082 + w_ctrl_041} \\
&w_ctrl_083 = NAND(INV(w_ctrl_082), INV(w_ctrl_041)) \\
&o_LUI = \overline{w_ctrl_083} \\
&o_LUI = INV(w_ctrl_083)
\end{aligned}
\tag{58}
$$

The next control to be examined is REG_SEL to select the register to which the ALU Core result will be stored. REG_SEL is overridden external to the instruction decoder by state dependent control signals. The purpose of these overrides is to prevent writing to a register when the data to be stored has not yet stabilized or a final result has not yet been calculated. This section discusses how REG_SEL is determined based solely on the instructions and only on the first cycle/iteration of

branch instructions. The REG_SEL 0 bit is set to 1 when the instruction has a 000000 opcode. The REG_SEL 1 bit is set to 1 when the instruction has a 001XXX or a 1X0XXX opcode. Since the instructions are mutually exclusive, REG_SEL is 00 when REG_SEL(0) and REG_SEL(1) are both 0. REG_SEL can only be set to 11 by a state dependent override signal and the override does not occur within the Instruction Decoder module. The logic for determining REG_SEL 0 bit is shown here:

$$
\begin{aligned}
o_REG_SEL(0) &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28} \\
&\quad\ \overline{i_instr27}\ \overline{i_instr26} \\
o_REG_SEL(0) &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}} \\
&\quad\ \overline{\overline{i_instr27}\ \overline{i_instr26}} \\
o_REG_SEL(0) &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + \cdots} \\
&\quad\ \overline{i_instr27 + i_instr26} \\
o_REG_SEL(0) &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002} \\
o_REG_SEL(0) &= \overline{w_ctrl_005 + w_ctrl_002} \\
w_ctrl_094 &= NAND(INV(w_ctrl_005), INV(w_ctrl_002)) \\
o_REG_SEL(0) &= \overline{w_ctrl_094} \\
o_REG_SEL(0) &= INV(w_ctrl_094)
\end{aligned}
\tag{59}
$$

The logic for determining REG_SEL 1 bit is shown here:

$$
\begin{aligned}
o_REG_SEL(1) &= i_instr31\ \overline{i_instr29} + \overline{i_instr31}\ \overline{i_instr30}\ i_instr29 \\
o_REG_SEL(1) &= \overline{\overline{i_instr31\ \overline{i_instr29} + \overline{i_instr31}\ \overline{i_instr30}\ i_instr29}} \\
o_REG_SEL(1) &= \overline{[\overline{i_instr31} + i_instr29][i_instr31 + i_instr30 + \overline{i_instr29}]} \\
w_ctrl_098 &= NAND(i_instr31, INV(i_instr29)) \\
o_REG_SEL(1) &= \overline{w_ctrl_098[w_ctrl_000 + \overline{i_instr29}]} \\
w_ctrl_099 &= NAND(INV(w_ctrl_000), i_instr29) \\
o_REG_SEL(1) &= \overline{w_ctrl_098\ w_ctrl_099} \\
o_REG_SEL(1) &= NAND(w_ctrl_098, w_ctrl_099)
\end{aligned}
\tag{60}
$$

The next control to be examined is OVER_CTRL to determine whether overflow detection should be used on the output of the adder. OVER_CTRL is used for the ADD, SUB, and ADDI instructions. In order for OVER_CTRL to take on the value 01 for ADD instructions, 10 for SUB instructions, 11 for ADDI instructions, and 00 for all other instructions, $OVER_CTRL(0) = ADD + ADDI$ and $OVER_CTRL(1) = SUB + ADDI$. The following logic shows how the ADD instruction is determined:

$$
\begin{aligned}
ADD &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad\ i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ \overline{i_instr1}\ \overline{i_instr0} \\
ADD &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26}} \\
&\quad\ \overline{\overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ \overline{i_instr1}\ \overline{i_instr0}} \\
ADD &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + \cdots} \\
&\quad\ \overline{i_instr27 + i_instr26 + \overline{i_instr5} + i_instr4 + \cdots} \\
&\quad\ \overline{i_instr3 + i_instr2 + i_instr1 + i_instr0} \\
ADD &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002 + w_ctrl_055 + \cdots} \\
&\quad\ \overline{w_ctrl_004 + w_ctrl_043} \\
w_ctrl_044 &= NAND(INV(w_ctrl_004), INV(w_ctrl_043)) \\
ADD &= \overline{w_ctrl_005 + w_ctrl_078 + w_ctrl_044} \\
ADD &= \overline{w_ctrl_080 + w_ctrl_044} \\
w_ctrl_117 &= NAND(INV(w_ctrl_080), INV(w_ctrl_044)) \\
ADD &= \overline{w_ctrl_117} \\
ADD &= INV(w_ctrl_117)
\end{aligned}
\tag{61}
$$

The following logic shows how the SUB instruction is determined:

$$
\begin{aligned}
SUB &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
&\quad i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ i_instr1\ \overline{i_instr0} \\
SUB &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26}} \\
&\quad \overline{i_instr5\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ i_instr1\ \overline{i_instr0}} \\
SUB &= \overline{i_instr31 + i_instr30 + i_instr29 + i_instr28 + \cdots} \\
&\quad \overline{i_instr27 + i_instr26 + \overline{i_instr5} + i_instr4 + i_instr3 + \cdots} \\
&\quad \overline{i_instr2 + \overline{i_instr1} + i_instr0} \\
SUB &= \overline{w_ctrl_000 + w_ctrl_001 + w_ctrl_002 + w_ctrl_055 + \cdots} \\
&\quad \overline{w_ctrl_004 + w_ctrl_070} \\
w_ctrl_118 &= NAND(INV(w_ctrl_004), INV(w_ctrl_070)) \\
SUB &= \overline{w_ctrl_005 + w_ctrl_078 + w_ctrl_118} \\
SUB &= \overline{w_ctrl_080 + w_ctrl_118} \\
w_ctrl_119 &= NAND(INV(w_ctrl_080), INV(w_ctrl_118)) \\
SUB &= \overline{w_ctrl_119} \\
SUB &= INV(w_ctrl_119)
\end{aligned}
\tag{62}
$$

The following logic shows how the ADDI instruction is determined:

$$
\begin{aligned}
ADDI &= \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
ADDI &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26}} \\
ADDI &= \overline{i_instr31 + i_instr30 + \overline{i_instr29} + i_instr28 + \cdots} \\
&\quad \overline{i_instr27 + i_instr26} \\
w_ctrl_102 &= NAND(i_instr29, INV(i_instr28)) \\
ADDI &= \overline{w_ctrl_000 + w_ctrl_102 + w_ctrl_002} \\
ADDI &= \overline{w_ctrl_053 + w_ctrl_102} \\
w_ctrl_103 &= NAND(INV(w_ctrl_053), INV(w_ctrl_102)) \\
ADDI &= \overline{w_ctrl_103} \\
ADDI &= INV(w_ctrl_103)
\end{aligned}
\tag{63}
$$

The following logic shows how OVER_CTRL(0) is determined:

$$
\begin{aligned}
o_OVER_CTRL(0) &= ADD + ADDI \\
o_OVER_CTRL(0) &= \overline{w_ctrl_117} + \overline{w_ctrl_103} \\
o_OVER_CTRL(0) &= \overline{\overline{w_ctrl_117} + \overline{w_ctrl_103}} \\
o_OVER_CTRL(0) &= \overline{w_ctrl_117\ w_ctrl_103} \\
o_OVER_CTRL(0) &= NAND(w_ctrl_117, w_ctrl_103)
\end{aligned}
\tag{64}
$$

The following logic shows how OVER_CTRL(1) is determined:

$$
\begin{aligned}
o_OVER_CTRL(1) &= SUB + ADDI \\
o_OVER_CTRL(1) &= \overline{w_ctrl_119} + \overline{w_ctrl_103} \\
o_OVER_CTRL(1) &= \overline{\overline{w_ctrl_119} + \overline{w_ctrl_103}} \\
o_OVER_CTRL(1) &= \overline{w_ctrl_119\ w_ctrl_103} \\
o_OVER_CTRL(1) &= NAND(w_ctrl_119, w_ctrl_103)
\end{aligned}
\tag{65}
$$

In the case that an invalid instruction, one whose opcode, function code, and RT register value if necesary do not match one of the 33 Basic MIPS instructions, is present in the instruction register due to some error, a control signal needs to be generated that will reset the Controller finite state machine (FSM) back to the zero state so that the instruction can be re-read from memory to correct the error. If the error were not corrected, the behavior of the Datapath and ALU Core would not be as expected because the control signals from the Instruction_Decoder would be in error. While the control signals from the Instruction_Decoder are predicatable for a given input pattern, the Datapath would not perform the instruction as designed before the error occurred. This reset process assumes the instruction in memory was not in error and that the error occurred in the combinatorial logic leading up to the instruction register or inside the instruction register itself. This also does not protect against an error that might result in the erroneous instruction still being a valid instruction, but not the instruction that was stored in memory. In other words, a single bit error in bit 1 of an ADD instruction would cause it to become a SUB instruction without detecting an invalid instruction. To detect an invalid instruction, logic was first implemented to create a signal when an instruction is valid. That signal is then negated to create a signal for an invalid instruction. The logic for determining a valid instruction and an invalid instruction is shown here:

$$
\begin{aligned}
VALID = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
& [\overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ \overline{i_instr1}\ \overline{i_instr0} + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ i_instr1 + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr0} + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ i_instr1\ i_instr0 + \cdots \\
& i_instr5\ \overline{i_instr4}\ \overline{i_instr3} + \cdots \\
& i_instr5\ \overline{i_instr4}\ i_instr3\ \overline{i_instr2}\ i_instr1] + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ i_instr26 \\
& \overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17} + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28\ [\overline{i_instr27} + \cdots \\
& i_instr27\ \overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17}\ \overline{i_instr16}] + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ i_instr29 + \cdots \\
& i_instr31\ \overline{i_instr30}\ \overline{i_instr28}\ i_instr27\ i_instr26
\end{aligned}
\tag{66}
$$

$$
\begin{aligned}
VALID_1 = {} & \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ \overline{i_instr1}\ \overline{i_instr0} + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ \overline{i_instr2}\ i_instr1 + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ \overline{i_instr0} + \cdots \\
& \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}\ i_instr2\ i_instr1\ i_instr0 + \cdots \\
& i_instr5\ \overline{i_instr4}\ \overline{i_instr3} + \cdots \\
& i_instr5\ \overline{i_instr4}\ i_instr3\ \overline{i_instr2}\ i_instr1]
\end{aligned}
$$

$$
\begin{aligned}
VALID_1 = {} & \overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}(\overline{i_instr2}\ \{i_instr1 + \cdots \\
& \overline{i_instr1}\ \overline{i_instr0}\} + i_instr2\ \{\overline{i_instr0} + \cdots \\
& i_instr1\ i_instr0\}) + i_instr5\ \overline{i_instr4}\ (\overline{i_instr3} + \cdots \\
& i_instr3\ \overline{i_instr2}\ i_instr1)
\end{aligned}
$$

$$
\begin{aligned}
VALID_1 = {} & \overline{\overline{\overline{i_instr5}\ \overline{i_instr4}\ \overline{i_instr3}(\overline{i_instr2}\ \{i_instr1 + \cdots} \\
& \overline{\overline{\overline{i_instr1}\ \overline{i_instr0}\}} + i_instr2\ \{\overline{i_instr0} + \cdots} \\
& \overline{\overline{i_instr1}\ i_instr0\}) + i_instr5\ \overline{i_instr4}\ (\overline{i_instr3} + \cdots} \\
& \overline{\overline{i_instr3}\ \overline{i_instr2}\ i_instr1)} \\
VALID_1 = {} & \overline{[i_instr5 + i_instr4 + i_instr3 + \overline{(\cdots)}]} \\
& \overline{[\overline{i_instr5} + i_instr4 + \overline{(\cdots)}]} \\
VALID_1 = {} & \overline{[w_ctrl_003 + i_instr3 + \overline{(\cdots)}][w_ctrl_055 + \overline{(\cdots)}]} \\
w_ctrl_120 = {} & NAND(INV(w_ctrl_003), INV(i_instr3)) \\
VALID_1 = {} & \overline{[w_ctrl_120 + \overline{(\cdots)}][w_ctrl_055 + \overline{(\cdots)}]} \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{\{\cdots\}})(\overline{i_instr2} + \overline{\{\cdots\}})]} \\
& \overline{[w_ctrl_055 + i_instr3\ (\overline{i_instr3} + i_instr2 + \overline{i_instr1})]} \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{\{\cdots\}})(\overline{i_instr2} + \overline{\{\cdots\}})]} \\
& \overline{[w_ctrl_055 + i_instr3\ (i_instr2 + \overline{i_instr1})]} \qquad (67) \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{\{\cdots\}})(\overline{i_instr2} + \overline{\{\cdots\}})]} \\
& \overline{[w_ctrl_055 + i_instr3\ w_ctrl_110]} \\
w_ctrl_121 = {} & NAND(i_instr3, w_ctrl_110) \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{\{\cdots\}})(\overline{i_instr2} + \overline{\{\cdots\}})]} \\
& \overline{[w_ctrl_055 + \overline{i_instr121}]} \\
w_ctrl_122 = {} & NAND(INV(w_ctrl_055), w_ctrl_121) \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{\{\cdots\}})(\overline{i_instr2} + \overline{\{\cdots\}})]w_ctrl_122} \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{i_instr1}\ \{i_instr1 + \cdots} \\
& \overline{i_instr0\})(\overline{i_instr2} + i_instr0\ \{\overline{i_instr1} + \cdots} \\
& \overline{\overline{i_instr0}\})]w_ctrl_122} \\
VALID_1 = {} & \overline{[w_ctrl_120 + (i_instr2 + \overline{i_instr1}\ i_instr0)} \\
& \overline{(\overline{i_instr2} + i_instr0\ \overline{i_instr1})]w_ctrl_122}
\end{aligned}
$$

$$
\begin{aligned}
VALID_1 &= \overline{[w_ctrl_120 + i_instr2\ \overline{i_instr2} + \cdots} \\
&\quad \overline{i_instr2\ \overline{i_instr1}\ i_instr0 + \cdots} \\
&\quad \overline{\overline{i_instr2}\ \overline{i_instr1}\ i_instr0 + \overline{i_instr1}\ i_instr0]w_ctrl_122} \\
VALID_1 &= \overline{[w_ctrl_120 + \overline{i_instr1}\ i_instr0 + \cdots} \\
&\quad \overline{\overline{i_instr1}\ i_instr0]w_ctrl_122} \\
VALID_1 &= \overline{[w_ctrl_120 + \overline{i_instr1}\ i_instr0]w_ctrl_122} \\
VALID_1 &= \overline{[w_ctrl_120 + \overline{w_ctrl_062}]w_ctrl_122} \\
w_ctrl_123 &= NAND(INV(w_ctrl_120), w_ctrl_062) \\
VALID_1 &= \overline{w_ctrl_123\ w_ctrl_122} \\
w_ctrl_124 &= NAND(w_ctrl_123, w_ctrl_122) \\
VALID_1 &= w_ctrl_124
\end{aligned}
\tag{68}
$$

$$
\begin{aligned}
VALID_2 &= \overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17} \\
VALID_2 &= \overline{\overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17}} \\
VALID_2 &= \overline{i_instr20 + i_instr19 + i_instr18 + i_instr17} \\
w_ctrl_125 &= NAND(INV(i_instr20), INV(i_instr19)) \\
w_ctrl_126 &= NAND(INV(i_instr18), INV(i_instr17)) \\
VALID_2 &= \overline{w_ctrl_125 + w_ctrl_126} \\
w_ctrl_127 &= NAND(INV(w_ctrl_125), INV(w_ctrl_126)) \\
VALID_2 &= \overline{w_ctrl_127}
\end{aligned}
\tag{69}
$$

$$
\begin{aligned}
VALID_3 &= \overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17}\ \overline{i_instr16} \\
VALID_3 &= \overline{\overline{i_instr20}\ \overline{i_instr19}\ \overline{i_instr18}\ \overline{i_instr17}\ \overline{i_instr16}} \\
VALID_3 &= \overline{i_instr20 + i_instr19 + i_instr18 + i_instr17 + i_instr16} \\
VALID_3 &= \overline{w_ctrl_125 + w_ctrl_126 + i_instr16} \\
VALID_3 &= \overline{w_ctrl_127 + i_instr16} \\
w_ctrl_128 &= NAND(INV(w_ctrl_127), INV(i_instr16)) \\
VALID_3 &= \overline{w_ctrl_128}
\end{aligned}
\tag{70}
$$

$$
\begin{aligned}
VALID = {} & \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}\ \overline{i_instr26} \\
& VALID_1 + \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
& i_instr26\ VALID_2 + \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28 \\
& [\overline{i_instr27} + i_instr27\ VALID_3] + \cdots \\
& \overline{i_instr31}\ \overline{i_instr30}\ i_instr29 + \cdots \\
& i_instr31\ \overline{i_instr30}\ \overline{i_instr28}\ i_instr27\ i_instr26 \\
VALID = {} & \overline{i_instr30}\ [i_instr31\ \overline{i_instr28}\ i_instr27\ i_instr26 + \cdots \\
& \overline{i_instr31}(i_instr29 + \overline{i_instr29}\{i_instr28\ [\overline{i_instr27} + \cdots \\
& i_instr27\ VALID_3] + \overline{i_instr28}\ \overline{i_instr27} \\
& [\overline{i_instr26}\ VALID_1 + i_instr26\ VALID_2]\})] \\
VALID = {} & \overline{i_instr30}\ [i_instr31\ \overline{i_instr28}\ i_instr27\ i_instr26 + \cdots \\
& \overline{i_instr31}(i_instr29 + \overline{i_instr29}\{i_instr28\ [\overline{i_instr27} + \cdots \\
& i_instr27\ \overline{w_ctrl_128}] + \overline{i_instr28}\ \overline{i_instr27} \\
& [\overline{i_instr26}\ w_ctrl_124 + i_instr26\ \overline{w_ctrl_127}]\})] \\
VALID = {} & \overline{\overline{i_instr30}\ [i_instr31\ \overline{i_instr28}\ i_instr27\ i_instr26 + \cdots} \\
& \overline{\overline{i_instr31}(i_instr29 + \overline{i_instr29}\{i_instr28\ [\overline{i_instr27} + \cdots} \\
& \overline{\overline{i_instr27\ \overline{w_ctrl_128}] + \overline{i_instr28}\ \overline{i_instr27}}} \\
& \overline{\overline{[\overline{i_instr26}\ w_ctrl_124 + i_instr26\ \overline{w_ctrl_127}]\})]} \\
VALID = {} & \overline{i_instr30 + [\overline{i_instr31} + i_instr28 + \overline{i_instr27} + \overline{i_instr26}]} \\
& \overline{[i_instr31 + \overline{(\cdots)}]} \\
w_ctrl_130 = {} & NAND(i_instr31, INV(i_instr28)) \\
VALID = {} & \overline{i_instr30 + [w_ctrl_130 + w_ctrl_041][i_instr31 + \overline{(\cdots)}]} \\
w_ctrl_131 = {} & NAND(INV(w_ctrl_130), INV(w_ctrl_041)) \\
VALID = {} & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \overline{(\cdots)}]}
\end{aligned}
\tag{71}
$$

$$
\begin{aligned}
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ (i_instr29 + \overline{\{\cdots\}})]} \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \overline{i_instr29}\ \overline{\{\cdots\}}]} \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ \{\overline{i_instr28} + \overline{[\cdots]}\}\{i_instr28 + \cdots} \\
& \overline{i_instr27 + \overline{[\cdots]}\}]} \\
w_ctrl_132 = & NAND(INV(i_instr28), INV(i_instr27) \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ \{\overline{i_instr28} + i_instr27\ [\overline{i_instr27} + w_ctrl_128]\}} \\
& \overline{\{w_ctrl_132 + [i_instr26 + \overline{w_ctrl_124}]} \\
& \overline{[\overline{i_instr26} + w_ctrl_127]\}]} \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ \{\overline{i_instr28} + i_instr27\ w_ctrl_128\}} \\
& \overline{\{w_ctrl_132 + [i_instr26 + \overline{w_ctrl_124}]} \\
& \overline{[\overline{i_instr26} + w_ctrl_127]\}]} \\
w_ctrl_129 = & NAND(i_instr27, w_ctrl_128) \\
w_ctrl_134 = & NAND(INV(i_instr26), w_ctrl_124) \\
w_ctrl_135 = & NAND(i_instr26, INV(w_ctrl_127) \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ \{\overline{i_instr28} + w_ctrl_129\}} \\
& \overline{\{w_ctrl_132 + w_ctrl_134\ w_ctrl_135\}]} \\
w_ctrl_133 = & NAND(i_instr28, w_ctrl_129) \\
w_ctrl_136 = & NAND(w_ctrl_134, w_ctrl_135) \\
VALID = & \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \cdots} \\
& \overline{\overline{i_instr29}\ w_ctrl_133\ \{w_ctrl_132 + \overline{w_ctrl_136}\}]}
\end{aligned}
\tag{72}
$$

$$
\begin{aligned}
w_ctrl_137 &= NAND(INV(i_instr29), w_ctrl_133) \\
w_ctrl_138 &= NAND(INV(w_ctrl_132), w_ctrl_136) \\
VALID &= \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \overline{w_ctrl_137}\ w_ctrl_138]} \\
w_ctrl_139 &= NAND(INV(w_ctrl_137), w_ctrl_138) \\
VALID &= \overline{i_instr30 + w_ctrl_131\ [i_instr31 + \overline{w_ctrl_139}]} \\
w_ctrl_140 &= NAND(INV(i_instr31), w_ctrl_139) \\
VALID &= \overline{i_instr30 + w_ctrl_131\ w_ctrl_140} \\
w_ctrl_141 &= NAND(w_ctrl_131, w_ctrl_140) \\
VALID &= \overline{i_instr30 + \overline{w_ctrl_141}} \\
o_INVALID_INSTR &= \overline{\overline{i_instr30 + \overline{w_ctrl_141}}} \\
o_INVALID_INSTR &= \overline{\overline{i_instr30}\ w_ctrl_141} \\
o_INVALID_INSTR &= NAND(INV(i_instr30), w_ctrl_141)
\end{aligned}
\tag{73}
$$

The next control signal to discuss is the UNSIGNED control signal used by the comparator in the ALU Core. Unsigned is 1 for SLTU and SLTIU instructions and 0 otherwise. The logic for determining UNSIGNED is shown here:

$$o_UNSIGNED = \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27}$$
$$\overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3\ \overline{i_instr2}\ i_instr1$$
$$i_instr0 + \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ \overline{i_instr28}$$
$$i_instr27\ i_instr26$$
$$o_UNSIGNED = \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}$$
$$[i_instr29\ i_instr27\ i_instr26 + \cdots$$
$$\overline{i_instr29}\ \overline{i_instr27}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3$$
$$\overline{i_instr2}\ i_instr1\ i_instr0]$$
$$o_UNSIGNED = \overline{\overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr28}}}$$
$$\overline{\overline{[i_instr29\ i_instr27\ i_instr26 + \cdots}}$$
$$\overline{\overline{\overline{i_instr29}\ \overline{i_instr27}\ \overline{i_instr26}\ i_instr5\ \overline{i_instr4}\ i_instr3}}$$
$$\overline{\overline{\overline{i_instr2}\ i_instr1\ i_instr0]}}$$
$$o_UNSIGNED = \overline{i_instr31 + i_instr30 + i_instr28 + [\overline{i_instr29} + \cdots} \tag{74}$$
$$\overline{\overline{i_instr27} + \overline{i_instr26}][i_instr29 + i_instr27 + \cdots}$$
$$\overline{i_instr26 + \overline{i_instr5} + i_instr4 + \overline{i_instr3} + \cdots}$$
$$\overline{i_instr2 + \overline{i_instr1} + \overline{i_instr0}]}$$
$$o_UNSIGNED = \overline{w_ctrl_000 + i_instr28 + [w_ctrl_029 + \overline{i_instr26}]}$$
$$\overline{[w_ctrl_014 + i_instr26 + w_ctrl_055 + \cdots}$$
$$\overline{w_ctrl_085 + w_ctrl_077]}$$
$$w_ctrl_142 = NAND(INV(w_ctrl_029), i_instr26)$$
$$w_ctrl_143 = NAND(INV(w_ctrl_014), i_instr26)$$
$$o_UNSIGNED = \overline{w_ctrl_028 + w_ctrl_142[w_ctrl_143 + \cdots}$$
$$\overline{w_ctrl_086 + w_ctrl_077]}$$
$$w_ctrl_144 = NAND(INV(w_ctrl_143), INV(w_ctrl_086))$$
$$o_UNSIGNED = \overline{w_ctrl_028 + w_ctrl_142[w_ctrl_144 + w_ctrl_077]}$$

$$
\begin{aligned}
&w_ctrl_145 = NAND(INV(w_ctrl_144), INV(w_ctrl_077)) \\
&o_UNSIGNED = \overline{w_ctrl_028 + w_ctrl_142\ w_ctrl_145} \\
&w_ctrl_146 = NAND(w_ctrl_142, w_ctrl_145) \\
&o_UNSIGNED = \overline{w_ctrl_028 + \overline{w_ctrl_146}} \\
&w_ctrl_147 = NAND(INV(w_ctrl_028), w_ctrl_146) \\
&o_UNSIGNED = \overline{w_ctrl_147} \\
&o_UNSIGNED = INV(w_ctrl_147)
\end{aligned}
\tag{75}
$$

The next control signal to discuss is the overflow control signal used by the comparator to determine whether an instruction requires overflow detection in the case of SLT, SLTU, SLTI, and SLTIU instrcutions or no overflow detection for BLTZ instructions. The logic for determining overflow is shown here:

$$
o_overflow = w_ctrl_025 \tag{76}
$$

The next control signal to discuss is the imm_extend control signal used in the Datapath to determine whether the immediate value should be sign extended or zero extended in the ALU Core. If imm_extend is 1, the immediate is sign extended and if it is 0, the immediate is zero extended. Instructions requiring sign extension of the immediate are BGEZ, BLTZ, BEQ, BNE, BLEZ, BGTZ, ADDI, ADDIU, SLTI, SLTIU and those requiring zero extension are ANDI, ORI, and XORI. The logic for determining imm_extend is shown here:

$$
\begin{aligned}
zero_extend &= \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ \overline{i_instr27}\ \overline{i_instr26} \\
&+\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ \overline{i_instr27}\ i_instr26 \\
&+\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ i_instr27\ \overline{i_instr26} \\
&+\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28\ i_instr27\ i_instr26 \\
zero_extend &= \overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28 \\
zero_extend &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ i_instr29\ i_instr28} \\
zero_extend &= \overline{i_instr31 + i_instr30 + \overline{i_instr29} + \overline{i_instr28}} \\
zero_extend &= \overline{w_ctrl_000 + w_ctrl_059} \\
zero_extend &= \overline{w_ctrl_082} \\
o_imm_extend &= \overline{\overline{w_ctrl_082}} \\
o_imm_extend &= w_ctrl_082
\end{aligned}
\tag{77}
$$

The next control to discuss impacts the finite state machine (FSM) in the Controller. Based on the current instruction, it determines which state should come next after the instruction is read from memory. There are four possible next states after an instruction is read from memory. Branch instructions, LW, and SW each require special states in the FSM that are distinct from the states that the other 25 instructions follow. Therefore, the FSM_CTRL signal is a 2-bit signal that can take on four values. These values are 00 for the 25 standard instructions, 01 for the six branch instructions, 10 for LW, and 11 for SW. The following logic shows how the FSM_CTRL signal is determined based on the type of instruction:

$$
\begin{aligned}
o_FSM_CTRL(0) &= BRANCH + SW \\
o_FSM_CTRL(1) &= SW + LW
\end{aligned}
\tag{78}
$$

The logic to determine BRANCH is:

$$
\begin{aligned}
BRANCH &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ \overline{i_instr28}\ \overline{i_instr27} \\
&\quad\ i_instr26 + \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ i_instr28 \\
BRANCH &= \overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ [i_instr28 + \cdots \\
&\quad\ \overline{i_instr28}\ \overline{i_instr27}\ i_instr26] \\
BRANCH &= \overline{\overline{i_instr31}\ \overline{i_instr30}\ \overline{i_instr29}\ [i_instr28 + \cdots} \\
&\quad\ \overline{\overline{i_instr28}\ \overline{i_instr27}\ i_instr26]} \\
BRANCH &= \overline{i_instr31 + i_instr30 + i_instr29 + \cdots} \\
&\quad\ \overline{\overline{i_instr28}[i_instr28 + i_instr27 + \overline{i_instr26}]} \\
BRANCH &= \overline{w_ctrl_000 + i_instr29 + \overline{i_instr28}[i_instr27 + \overline{i_instr26}]} \\
w_ctrl_025 &= NAND(INV(i_instr27), i_instr26) \\
BRANCH &= \overline{w_ctrl_009 + \overline{i_instr28}\ w_ctrl_025} \\
w_ctrl_100 &= NAND(INV(i_instr28), w_ctrl_025) \\
BRANCH &= \overline{w_ctrl_009 + \overline{w_ctrl_100}} \\
w_ctrl_101 &= NAND(INV(w_ctrl_009), w_ctrl_100) \\
BRANCH &= \overline{w_ctrl_101}
\end{aligned}
\tag{79}
$$

The logic to determine LW is:

$$
\begin{aligned}
LW &= i_instr31\ \overline{i_instr29} \\
LW &= \overline{\overline{i_instr31\ \overline{i_instr29}}} \\
w_ctrl_098 &= NAND(i_instr31, INV(i_instr29)) \\
LW &= \overline{w_ctrl_098}
\end{aligned}
\tag{80}
$$

The logic to determine SW is:

$$\begin{aligned} SW &= i_instr31\ i_instr29 \\ SW &= \overline{\overline{i_instr31\ i_instr29}} \\ w_ctrl_095 &= NAND(i_instr31, i_instr29) \\ SW &= \overline{w_ctrl_095} \end{aligned} \tag{81}$$

Finally, the logic to determine FSM_CTRL(0) is:

$$\begin{aligned} o_FSM_CTRL(0) &= BRANCH + SW \\ o_FSM_CTRL(0) &= \overline{w_ctrl_101} + \overline{w_ctrl_095} \\ o_FSM_CTRL(0) &= \overline{\overline{\overline{w_ctrl_101} + \overline{w_ctrl_095}}} \\ o_FSM_CTRL(0) &= \overline{w_ctrl_101\ w_ctrl_095} \\ o_FSM_CTRL(0) &= NAND(w_ctrl_101, w_ctrl_095) \end{aligned} \tag{82}$$

The logic to determine FSM_CTRL(1) is:

$$\begin{aligned} o_FSM_CTRL(1) &= SW + LW \\ o_FSM_CTRL(1) &= \overline{w_ctrl_095} + \overline{w_ctrl_098} \\ o_FSM_CTRL(1) &= \overline{\overline{\overline{w_ctrl_095} + \overline{w_ctrl_098}}} \\ o_FSM_CTRL(1) &= \overline{w_ctrl_095\ w_ctrl_098} \\ o_FSM_CTRL(1) &= NAND(w_ctrl_095, w_ctrl_098) \end{aligned} \tag{83}$$

Table 44 summarizes all of the control signals introduced in this section.

Table 44. Instruction_Decoder Signal Summary

Signal Name	Logic
w_ctrl_000	NAND(INV(i_instr31),INV(i_instr30))
w_ctrl_001	NAND(INV(i_instr29),INV(i_instr28))
w_ctrl_002	NAND(INV(i_instr27),INV(i_instr26))

Table 44 – *Continued on next page*

Table 44 – *Continued from previous page*

Signal Name	Logic
w_ctrl_003	NAND(INV(i_instr05),INV(i_instr04))
w_ctrl_004	NAND(INV(i_instr03),INV(i_instr02))
w_ctrl_005	NAND(INV(w_ctrl_000),INV(w_ctrl_001))
w_ctrl_006	NAND(INV(w_ctrl_002),INV(w_ctrl_003))
w_ctrl_007	NAND(INV(w_ctrl_005),INV(w_ctrl_006))
w_ctrl_008	NAND(INV(w_ctrl_007),INV(w_ctrl_004))
w_ctrl_009	NAND(INV(w_ctrl_000),INV(i_instr29))
w_ctrl_010	NAND(INV(i_instr28),INV(w_ctrl_002))
w_ctrl_011	NAND(i_instr28,INV(i_instr(27))
w_ctrl_012	NAND(w_ctrl_010,w_ctrl_011)
w_ctrl_013	NAND(INV(w_ctrl_009),w_ctrl_012)
w_ctrl_014	NAND(INV(i_instr29),INV(i_instr27))
w_ctrl_015	NAND(INV(i_instr29),INV(w_ctrl_011))
w_ctrl_018	NAND(INV(i_instr26),i_instr05)
w_ctrl_020	NAND(INV(w_ctrl_018),INV(i_instr04))
w_ctrl_025	NAND(INV(i_instr27),i_instr26)
w_ctrl_028	NAND(INV(w_ctrl_000),INV(i_instr28))
w_ctrl_029	NAND(i_instr29,i_instr27)
w_ctrl_030	NAND(i_instr26,i_instr16)
w_ctrl_031	NAND(INV(w_ctrl_014),w_ctrl_030)
w_ctrl_032	NAND(w_ctrl_029,w_ctrl_031)
w_ctrl_033	NAND(INV(w_ctrl_028),w_ctrl_032)
w_ctrl_034	NAND(INV(w_ctrl_000),INV(w_ctrl_015))
w_ctrl_035	NAND(INV(w_ctrl_034),INV(i_instr26))
w_ctrl_036	NAND(INV(w_ctrl_034),i_instr26)
w_ctrl_037	NAND(INV(in29),i_instr(28))
w_ctrl_038	NAND(i_instr27,INV(i_instr26))
w_ctrl_039	NAND(INV(w_ctrl_000),INV(w_ctrl_037))
w_ctrl_040	NAND(INV(w_ctrl_039),INV(w_ctrl_038))
w_ctrl_041	NAND(i_instr27,i_instr26)
w_ctrl_042	NAND(INV(w_ctrl_039),INV(w_ctrl_041))
w_ctrl_043	NAND(INV(i_instr01),INV(i_instr00))
w_ctrl_044	NAND(INV(w_ctrl_004),INV(w_ctrl_043))
w_ctrl_046	NAND(INV(i_instr03),i_instr01)
w_ctrl_047	NAND(INV(w_ctrl_007),INV(w_ctrl_046))
w_ctrl_048	NAND(INV(i_instr30),INV(i_instr27))
w_ctrl_049	NAND(INV(w_ctrl_020),INV(w_ctrl_004))
w_ctrl_050	NAND(INV(i_instr29),w_ctrl_049)
w_ctrl_051	NAND(INV(i_instr28),INV(w_ctrl_048))

Table 44 – *Continued on next page*

Table 44 – *Continued from previous page*

Signal Name	Logic
w_ctrl_052	NAND(INV(w_ctrl_051),w_ctrl_050)
w_ctrl_053	NAND(INV(w_ctrl_000),INV(w_ctrl_002))
w_ctrl_054	NAND(i_instr29,i_instr28)
w_ctrl_055	NAND(i_instr05,INV(i_instr04))
w_ctrl_056	NAND(INV(i_instr03),i_instr02)
w_ctrl_057	NAND(INV(w_ctrl_001),INV(w_ctrl_055))
w_ctrl_058	NAND(INV(w_ctrl_056),INV(w_ctrl_043))
w_ctrl_059	NAND(INV(w_ctrl_057),INV(w_ctrl_058))
w_ctrl_060	NAND(w_ctrl_054,w_ctrl_059)
w_ctrl_061	NAND(INV(w_ctrl_053),w_ctrl_060)
w_ctrl_062	NAND(INV(i_instr01),i_instr00)
w_ctrl_063	NAND(INV(w_ctrl_000),INV(i_instr27))
w_ctrl_064	NAND(INV(w_ctrl_054),i_instr26)
w_ctrl_065	NAND(INV(w_ctrl_056),INV(w_ctrl_062))
w_ctrl_066	NAND(INV(w_ctrl_057),INV(i_instr26))
w_ctrl_067	NAND(INV(w_ctrl_066),INV(w_ctrl_065))
w_ctrl_068	NAND(w_ctrl_064,w_ctrl_067)
w_ctrl_069	NAND(INV(w_ctrl_063),w_ctrl_068)
w_ctrl_070	NAND(i_instr01,INV(i_instr00))
w_ctrl_071	NAND(INV(w_ctrl_000),INV(i_instr26))
w_ctrl_072	NAND(INV(w_ctrl_054),i_instr27)
w_ctrl_073	NAND(INV(w_ctrl_056),INV(w_ctrl_070))
w_ctrl_074	NAND(INV(w_ctrl_057),INV(w_ctrl_073))
w_ctrl_075	NAND(w_ctrl_072,w_ctrl_148)
w_ctrl_076	NAND(INV(w_ctrl_071),w_ctrl_075)
w_ctrl_077	NAND(i_instr01,i_instr00)
w_ctrl_078	NAND(INV(w_ctrl_002),INV(w_ctrl_055))
w_ctrl_079	NAND(INV(w_ctrl_056),INV(w_ctrl_077))
w_ctrl_080	NAND(INV(w_ctrl_005),INV(w_ctrl_078))
w_ctrl_081	NAND(INV(w_ctrl_080),INV(w_ctrl_079))
w_ctrl_082	NAND(INV(w_ctrl_000),INV(w_ctrl_054))
w_ctrl_083	NAND(INV(w_ctrl_082),INV(w_ctrl_041))
w_ctrl_084	NAND(INV(w_ctrl_029),INV(i_instr28))
w_ctrl_085	NAND(i_instr03,INV(i_instr02))
w_ctrl_086	NAND(INV(w_ctrl_055),INV(w_ctrl_085))
w_ctrl_087	NAND(INV(i_instr26),i_instr01)
w_ctrl_094	NAND(INV(w_ctrl_005),INV(w_ctrl_002))
w_ctrl_095	NAND(i_instr31,i_instr29)
w_ctrl_098	NAND(i_instr31,INV(i_instr29))

Table 44 – *Continued on next page*

Table 44 – *Continued from previous page*

Signal Name	Logic
w_ctrl_099	NAND(INV(w_ctrl_000),i_instr29)
w_ctrl_100	NAND(INV(i_instr28),w_ctrl_025)
w_ctrl_101	NAND(INV(w_ctrl_009),w_ctrl_100)
w_ctrl_102	NAND(i_instr29,INV(28))
w_ctrl_103	NAND(INV(w_ctrl_053),INV(w_ctrl_102))
w_ctrl_106	NAND(INV(w_ctrl_014),INV(w_ctrl_087))
w_ctrl_107	NAND(INV(w_ctrl_086),INV(w_ctrl_106))
w_ctrl_108	NAND(w_ctrl_029,w_ctrl_107)
w_ctrl_109	NAND(INV(w_ctrl_028),w_ctrl_108)
w_ctrl_110	NAND(INV(i_instr02),i_instr01)
w_ctrl_111	NAND(INV(in3),w_ctrl_110)
w_ctrl_112	NAND(INV(w_ctrl_020),w_ctrl_111)
w_ctrl_113	NAND(INV(i_instr28),w_ctrl_112)
w_ctrl_114	NAND(INV(w_ctrl_014),w_ctrl_113)
w_ctrl_115	NAND(w_ctrl_084,w_ctrl_114)
w_ctrl_116	NAND(INV(w_ctrl_000),w_ctrl_115)
w_ctrl_117	NAND(INV(w_ctrl_080),INV(w_ctrl_044))
w_ctrl_118	NAND(INV(w_ctrl_004),INV(w_ctrl_070))
w_ctrl_119	NAND(INV(w_ctrl_080),INV(w_ctrl_118))
w_ctrl_120	NAND(INV(w_ctrl_003),INV(i_instr03))
w_ctrl_121	NAND(i_instr03,w_ctrl_110)
w_ctrl_122	NAND(INV(w_ctrl_055),w_ctrl_121)
w_ctrl_123	NAND(INV(w_ctrl_120),w_ctrl_062)
w_ctrl_124	NAND(w_ctrl_122,w_ctrl_123)
w_ctrl_125	NAND(INV(i_instr20),INV(i_instr19))
w_ctrl_126	NAND(INV(i_instr18),INV(i_instr17))
w_ctrl_127	NAND(INV(w_ctrl_125),INV(w_ctrl_126))
w_ctrl_128	NAND(INV(w_ctrl_127),INV(i_instr16))
w_ctrl_129	NAND(i_instr27,w_ctrl_128)
w_ctrl_130	NAND(i_instr31,INV(i_instr28))
w_ctrl_131	NAND(INV(w_ctrl_041),INV(w_ctrl_130))
w_ctrl_132	NAND(INV(i_instr27),INV(i_instr28))
w_ctrl_133	NAND(i_instr28,w_ctrl_129)
w_ctrl_134	NAND(INV(i_instr26),w_ctrl_124)
w_ctrl_135	NAND(i_instr26,INV(w_ctrl_127)
w_ctrl_136	NAND(w_ctrl_134,w_ctrl_135)
w_ctrl_137	NAND(INV(i_instr29),w_ctrl_133)
w_ctrl_138	NAND(INV(w_ctrl_132),w_ctrl_136)
w_ctrl_139	NAND(INV(w_ctrl_137),w_ctrl_138)

Table 44 – *Continued on next page*

Table 44 – *Continued from previous page*

Signal Name	Logic
w_ctrl_140	NAND(INV(i_instr31),w_ctrl_139)
w_ctrl_141	NAND(w_ctrl_131,w_ctrl_140)
w_ctrl_142	NAND(INV(w_ctrl_029),i_instr26)
w_ctrl_143	NAND(INV(w_ctrl_014),INV(i_instr26))
w_ctrl_144	NAND(INV(w_ctrl_143),INV(w_ctrl_086))
w_ctrl_145	NAND(INV(w_ctrl_144),INV(w_ctrl_077))
w_ctrl_146	NAND(w_ctrl_142,w_ctrl_145)
w_ctrl_147	NAND(INV(w_ctrl_028),w_ctrl_146)
w_ctrl_148	NAND(INV(w_ctrl_074),INV(i_instr27))
o_ALU_SRC_A0	INV(w_ctrl_008)
o_ALU_SRC_B(1)	NAND(INV(i_instr31),INV(i_instr29))
o_ALU_SRC_B(0)	INV(w_ctrl_013)
o_ALU_INV_B	INV(w_ctrl_116)
o_LTZ	INV(w_ctrl_033)
o_EQ	INV(w_ctrl_035)
o_NE	INV(w_ctrl_036)
o_LEZ	INV(w_ctrl_040)
o_GTZ	INV(w_ctrl_042)
o_RIGHT_SHIFT	INV(w_ctrl_047)
o_ADDER	NAND(INV(i_instr31),w_ctrl_052)
o_AND	INV(w_ctrl_061)
o_OR	INV(w_ctrl_069)
o_XOR	INV(w_ctrl_076)
o_NOR	INV(w_ctrl_081)
o_COMPARE	INV(w_ctrl_109)
o_LUI	INV(w_ctrl_083)
o_REG_SEL(1)	NAND(w_ctrl_098,w_ctrl_099)
o_REG_SEL(0)	INV(w_ctrl_094)
o_OV_CTRL(1)	NAND(w_ctrl_119,w_ctrl_103)
o_OV_CTRL(0)	NAND(w_ctrl_117,w_ctrl_103)
o_UNSIGNED	INV(w_ctrl_147)
o_overflow	w_ctrl_025
o_imm_extend	INV(w_ctrl_082)
o_FSM_CTRL(1)	NAND(w_ctrl_095,w_ctrl_098)
o_FSM_CTRL(0)	NAND(w_ctrl_095,w_ctrl_101)
o_INVALID_INSTR	NAND(INV(i_instr30),w_ctrl_141)

Table 45. Instruction_Decoder Inputs and Outputs

Port Name	Type	Width	Description
i_instr31	Input	1	Instruction bit 31
i_instr30	Input	1	Instruction bit 30
i_instr29	Input	1	Instruction bit 29
i_instr28	Input	1	Instruction bit 28
i_instr27	Input	1	Instruction bit 27
i_instr26	Input	1	Instruction bit 26
i_instr20	Input	1	Instruction bit 20
i_instr19	Input	1	Instruction bit 19
i_instr18	Input	1	Instruction bit 18
i_instr17	Input	1	Instruction bit 17
i_instr16	Input	1	Instruction bit 16
i_instr05	Input	1	Instruction bit 5
i_instr04	Input	1	Instruction bit 4
i_instr03	Input	1	Instruction bit 3
i_instr02	Input	1	Instruction bit 2
i_instr01	Input	1	Instruction bit 1
i_instr00	Input	1	Instruction bit 0
o_ALU_SRC_A0	Output	1	ALU Source A Control Signal 0 bit
o_ALU_SRC_B	Output	2	ALU Source B Control Signal
o_INV_B	Output	1	ALU Invert B Control Signal
o_LTZ	Output	1	Less Than Zero Comparison Signal
o_EQ	Output	1	Equal Comparison Signal
o_NE	Output	1	Not Equal Comparison Signal
o_LEZ	Output	1	Less Than or Equal to Zero Comparison Signal
o_GTZ	Output	1	Greater Than Zero Comparison Signal
o_RIGHT_SHIFT	Output	1	ALU Output Right Shift Signal
o_ADDER	Output	1	ALU Output Adder Signal
o_AND	Output	1	ALU Output AND Signal
o_OR	Output	1	ALU Output OR Signal
o_XOR	Output	1	ALU Output XOR Signal
o_NOR	Output	1	ALU Output NOR Signal
o_COMPARE	Output	1	ALU Output Compare Signal
o_LUI	Output	1	ALU Output Load Upper

Table 45 – *Continued on next page*

Table 45 – *Continued from previous page*

Port Name	Type	Width	Description
			Immediate Signal
o_REG_SEL	Output	2	Destination Register Select Signal
o_OVER_CTRL	Output	2	Overflow Control Signal
o_UNSIGNED	Output	1	Comparator Unsigned Control Signal
o_overflow	Output	1	Comparator Overflow Control Signal
o_imm_extend	Output	1	Determine whether immediate is zero or sign extended
o_FSM_CTRL	Output	2	Finite State Machine Next State Control Signal
o_INVALID_INSTR	Output	1	Invalid Instruction Control Signal

2.19 Encoders

2.19.1 Encoder6_3

As mentioned previously in the Instruction Decoder section, the Encoder6_3 is used to encode the comparison signals into a single 3-bit control signal to be used by the ALU Core. Input 1 of the encoder is LTZ, input 2 EQ, input 3 NE, input 4 LEZ, and input 5 is GTZ. The corresponding outpus are 001, 010, 011, 100, and 101 respectively with 000 indicating none of the inputs were high and the output will select the GEZ comparison in the comparator. Table 46 shows the Encoder6_3 truth table.

Table 46. Encoder6_3 Truth Table

Inputs					Output		
i_1 LTZ	i_2 EQ	i_3 NE	i_4 LEZ	i_5 GTZ	o_Z(2)	o_Z(1)	o_Z(0)
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	0	0	1	1	0	1

The logic for determining output bit 0 is therefore:

$$\begin{aligned} o_Z(0) &= i_1 + i_3 + i_5 \\ o_Z(0) &= \overline{\overline{i_1 + i_3 + i_5}} \\ o_Z(0) &= \overline{\overline{i_1}\ \overline{i_3}\ \overline{i_5}} \\ w_sig_0 &= NAND(INV(i_1), INV(i_3)) o_Z(0) = \overline{\overline{w_sig_0}\ \overline{i_5}} \\ o_Z(0) &= NAND(INV(w_sig_0), INV(i_5)) \end{aligned} \tag{84}$$

The logic for determining output bit 1 is therefore:

$$\begin{aligned} o_Z(1) &= i_2 + i_3 \\ o_Z(1) &= \overline{\overline{i_2 + i_3}} \\ o_Z(1) &= \overline{\overline{i_2}\ \overline{i_3}} \\ o_Z(1) &= NAND(INV(i_2), INV(i_3)) \end{aligned} \tag{85}$$

The logic for determining output bit 2 is therefore:

$$
\begin{aligned}
o_Z(2) &= i_4 + i_5 \\
o_Z(2) &= \overline{\overline{i_4 + i_5}} \\
o_Z(2) &= \overline{\overline{i_4}\ \overline{i_5}} \\
o_Z(2) &= NAND(INV(i_4), INV(i_5))
\end{aligned}
\tag{86}
$$

Table 47. Encoder6_3 Inputs and Outputs

Port Name	Type	Width	Description
i_1	Input	1	Input 1 (LTZ)
i_2	Input	1	Input 2 (EQ)
i_3	Input	1	Input 3 (NE)
i_4	Input	1	Input 4 (LEZ)
i_5	Input	1	Input 5 (GTZ)
o_Z	Output	3	Encoded Output (COMP_SEL)

2.19.2 Encoder9_4

As mentioned previously in the Instruction Decoder section, the Encoder9_4 is used to encode the ALU output into a single 4-bit control signal to be used by the ALU Core. Input 1 of the encoder is RIGHT_SHIFT, input 2 ADDER, input 3 AND, input 4 OR, input 5 XOR, input 6 NOR, input 7 COMP, and input 8 LUI. The corresponding outputs are 0001, 0010, 0011, 0100, 0101, 0110, 0111, and 1000 respectively with 0000 indicating none of the inputs were high and the output will select the LEFT_SHIFT output. Table 48 shows the Encoder9_4 truth table.

Table 48. Encoder9_4 Truth Table

Inputs								Output			
i_1 RIGHT_SHIFT	i_2 ADDER	i_3 AND	i_4 OR	i_5 XOR	i_6 NOR	i_7 COMP	i_8 LUI	o_Z(3)	o_Z(2)	o_Z(1)	o_Z(0)
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	1	1
0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	1	0	1
0	0	0	0	0	1	0	0	0	1	1	0
0	0	0	0	0	0	1	0	0	1	1	1
0	0	0	0	0	0	0	1	1	0	0	0

The logic for determining output bit 0 is therefore:

$$
\begin{aligned}
o_Z(0) &= i_7 + i_5 + i_3 + i_1 \\
o_Z(0) &= \overline{\overline{i_7 + i_5 + i_3 + i_1}} \\
o_Z(0) &= \overline{\overline{i_7}\ \overline{i_5}\ \overline{i_3}\ \overline{i_1}} \\
w_sig_3 &= NAND(INV(i_7), INV(i_5)) \\
w_sig_4 &= NAND(INV(i_3), INV(i_1)) \\
o_Z(0) &= \overline{\overline{w_sig_3}\ \overline{w_sig_4}} \\
o_Z(0) &= NAND(INV(w_sig_3), INV(w_sig_4))
\end{aligned}
\tag{87}
$$

The logic for determining output bit 1 is therefore:

$$
\begin{aligned}
o_Z(1) &= i_7 + i_6 + i_3 + i_2 \\
o_Z(1) &= \overline{\overline{i_7 + i_6 + i_3 + i_2}} \\
o_Z(1) &= \overline{\overline{i_7}\ \overline{i_6}\ \overline{i_3}\ \overline{i_2}} \\
w_sig_0 &= NAND(INV(i_7), INV(i_6)) \\
w_sig_2 &= NAND(INV(i_3), INV(i_2)) \\
o_Z(1) &= \overline{\overline{w_sig_0}\ \overline{w_sig_2}} \\
o_Z(1) &= NAND(INV(w_sig_0), INV(w_sig_2))
\end{aligned}
\tag{88}
$$

The logic for determining output bit 2 is therefore:

$$
\begin{aligned}
o_Z(2) &= i_7 + i_6 + i_5 + i_4 \\
o_Z(2) &= \overline{\overline{i_7 + i_6 + i_5 + i_4}} \\
o_Z(2) &= \overline{\overline{i_7}\ \overline{i_6}\ \overline{i_53}\ \overline{i_4}} \\
w_sig_1 &= NAND(INV(i_5), INV(i_4)) \\
o_Z(2) &= \overline{\overline{w_sig_0}\ \overline{w_sig_1}} \\
o_Z(2) &= NAND(INV(w_sig_0), INV(w_sig_1))
\end{aligned}
\tag{89}
$$

The logic for determining output bit 3 is therefore:

$$o_Z(3) = i_8 \tag{90}$$

Table 49. Encoder9_4 Inputs and Outputs

Port Name	Type	Width	Description
i_1	Input	1	Input 1 (RIGHT_SHIFT)
i_2	Input	1	Input 2 (ADDER)
i_3	Input	1	Input 3 (AND)
i_4	Input	1	Input 4 (OR)
i_5	Input	1	Input 5 (XOR)
i_6	Input	1	Input 6 (NOR)
i_7	Input	1	Input 7 (COMP)
i_8	Input	1	Input 8 (LUI)
o_Z	Output	4	Encoded Output (ALU_OUTPUT)

2.20 Instruction Decoder/Encoder

The Instruction_Dec_Enc module takes in the same inputs as the Instruction Decoder, but it also takes in a BRANCH_OVERRIDE signal, a DO_NOT_STORE signal, and a PC_STORE signal. These all function as override signals that override the normal function of the Instruction Decoder. The BRANCH_OVERRIDE signal forces the ALU_SRC_B and ALU_INV_B signals to be 11 and 0 respectively. It also sets ALU_SRC_A(1) to 1 so that the ALU A input is the PC register. This makes it so the adder can add the current value of the PC register to the branch offset << 2 or 4 depending on the result of the branch condition. The ALU_OUTPUT is also modified so that the result of the adder is selected. The DO_NOT_STORE signal forces the REG_SEL value to 00 so that the result is stored to the GPR Bank zero register. Because the zero register cannot be overwritten, it retains its value of 0. This is used so that no register in the GPR Bank nor the PC register may be overwritten while

output of the ALU or the incoming data from memory is in an unstable or metastable state. The states of the finite state machine are set up in such a way that the ALU output and incoming data from memory must be stable before the DO_NOT_STORE signal is set to 0 so that REG_SEL can send the data to the appropriate register. Finally, PC_STORE forces the value of REG_SEL to 11 to ensure the ALU output is written to the PC register. This is necessary for the last step in branch instructions. The Instruction_Dec_Enc outputs are the ALU_SRC_A, ALU_SRC_B, ALU_INV_B, COMP_SEL, OVER_CTRL, ALU_OUTPUT, REG_SEL, UNSIGNED, imm_extend, FSM_CTRL, and INVALID_INSTR signals.

The instruction inputs (instr31, ..., instr00) are passed directly to the Instruction Decoder inputs of the same name and do not affect any of the other workings of the Instruction Decoder/Encoder. The LTZ, EQ, NE, LEZ, and GTZ outputs from the Instruction Decoder are inputs 1 through 5 respectively to the Encoder6_3 and the output is the COMP_SEL signal. The RIGHT_SHIFT, ADDER, AND, OR, XOR, NOR, COMPARE, and LUI outputs from the Instruction Decoder are inputs 1 through 8 respectively to the Encoder9_4 and the output is a preliminary ALU_OUTPUT signal. The preliminary ALU_OUTPUT signal goes through a 2-input, 4-bit multiplexer as the 0 input and a constant 0010 is the 1 input. The select signal for this multiplexer is the OVERRIDE_B signal. The ALU_SRC_B signal passes thorugh a 2-input, 2-bit multiplexer as the 0 input and a constant 11 is the 1 input. The select signal is also OVERRIDE_B. The ALU_INV_B signal passes through a 2-input, 1-bit multiplexer as the 0 input and a constant 0 is the 1 input. The select signal is also OVERRIDE_B. The REG_SEL signal exiting the Instruction Decoder, called REG_SEL0, is the 0 input to a 2-input, 2-bit multiplexer whose 1 input is a constant 11 and whose select signal is PC_STORE. The output is the REG_SEL1 signal. REG_SEL1 is the 0 input to another 2-input, 2-bit multiplexer whose 1 input is a constant 00 and whose

select signal is DO_NOT_STORE. The output of this multiplexer is the REG_SEL signal that goes to the Datapath. Finally, the ALU_SRC_A bit 0 is set equal to the BRANCH_OVERRIDE signal. These connections are further clarified in Table 50 and illustrated in Figure 22.

Table 50. Instruction_Dec_Enc Connections

Instruction_Decoder			
I/O	Type	Width	Connection
i_instr31	Input	1	i_instr31
i_instr30	Input	1	i_instr30
i_instr29	Input	1	i_instr29
i_instr28	Input	1	i_instr28
i_instr27	Input	1	i_instr27
i_instr26	Input	1	i_instr26
i_instr20	Input	1	i_instr20
i_instr19	Input	1	i_instr19
i_instr18	Input	1	i_instr18
i_instr17	Input	1	i_instr17
i_instr16	Input	1	i_instr16
i_instr05	Input	1	i_instr05
i_instr04	Input	1	i_instr04
i_instr03	Input	1	i_instr03
i_instr02	Input	1	i_instr02
i_instr01	Input	1	i_instr01
i_instr00	Input	1	i_instr00
o_ALU_SRC_A0	Output	1	o_ALU_SRC_A(0)
o_ALU_SRC_B	Output	2	w_ALU_SRC_B
o_INV_B	Output	1	w_ALU_INV_B
o_LTZ	Output	1	w_LTZ
o_EQ	Output	1	w_EQ
o_NE	Output	1	w_NE
o_LEZ	Output	1	w_LEZ
o_GTZ	Output	1	w_GTZ
o_RIGHT_SHIFT	Output	1	w_RIGHT_SHIFT
o_ADDER	Output	1	w_ADDER
o_AND	Output	1	w_AND
o_OR	Output	1	w_OR

Table 50 – *Continued on next page*

Table 50 – *Continued from previous page*

o_XOR	Output	1	w_XOR
o_NOR	Output	1	w_NOR
o_COMPARE	Output	1	w_COMPARE
o_LUI	Output	1	w_LUI
o_REG_SEL	Output	2	w_REG_SEL0
o_OVER_CTRL	Output	2	o_OVER_CTRL
o_UNSIGNED	Output	1	o_UNSIGNED
o_overflow	Output	1	o_overflow
o_imm_extend	Output	1	o_imm_extend
o_FSM_CTRL	Output	2	o_FSM_CTRL
o_INVALID_INSTR	Output	1	o_INVALID_INSTR
Encoder6_3			
I/O	Type	Width	Connection
i_1	Input	1	w_LTZ
i_2	Input	1	w_EQ
i_3	Input	1	w_NE
i_4	Input	1	w_LEZ
i_5	Input	1	w_GTZ
o_Z	Output	3	o_COMP_SEL
Encoder9_4			
I/O	Type	Width	Connection
i_1	Input	1	w_RIGHT_SHIFT
i_2	Input	1	w_ADDER
i_3	Input	1	w_AND
i_4	Input	1	w_OR
i_5	Input	1	w_XOR
i_6	Input	1	w_NOR
i_7	Input	1	w_COMP
i_8	Input	1	w_LUI
o_Z	Output	4	w_ALU_OUTPUT
ALU_SRC_B Determined by BRANCH_OVERRIDE MUX2_2			
I/O	Type	Width	Connection
i_0	Input	2	w_ALU_SRC_B
i_1	Input	2	11
i_S	Input	1	i_BRANCH_OVERRIDE
o_Z	Output	2	o_ALU_SRC_B
ALU_INV_B Determined by BRANCH_OVERRIDE MUX2_1			
I/O	Type	Width	Connection
i_0	Input	1	w_ALU_INV_B
i_1	Input	1	0

Table 50 – *Continued on next page*

Table 50 – *Continued from previous page*

i_S	Input	1	i_BRANCH_OVERRIDE
o_Z	Output	1	o_ALU_INV_B
ALU_OUTPUT Determined by BRANCH_OVERRIDE MUX2_4			
I/O	Type	Width	Connection
i_0	Input	4	w_ALU_OUTPUT
i_1	Input	4	0010
i_S	Input	1	i_BRANCH_OVERRIDE
o_Z	Output	4	o_ALU_OUTPUT
REG_SEL1 Determined by PC_STORE MUX2_2			
I/O	Type	Width	Connection
i_0	Input	2	w_REG_SEL0
i_1	Input	2	11
i_S	Input	1	i_PC_STORE
o_Z	Output	2	w_REG_SEL1
REG_SEL Determined by DO_NOT_STORE MUX2_2			
I/O	Type	Width	Connection
i_0	Input	2	w_REG_SEL1
i_1	Input	2	00
i_S	Input	1	i_DO_NOT_STORE
o_Z	Output	2	o_REG_SEL

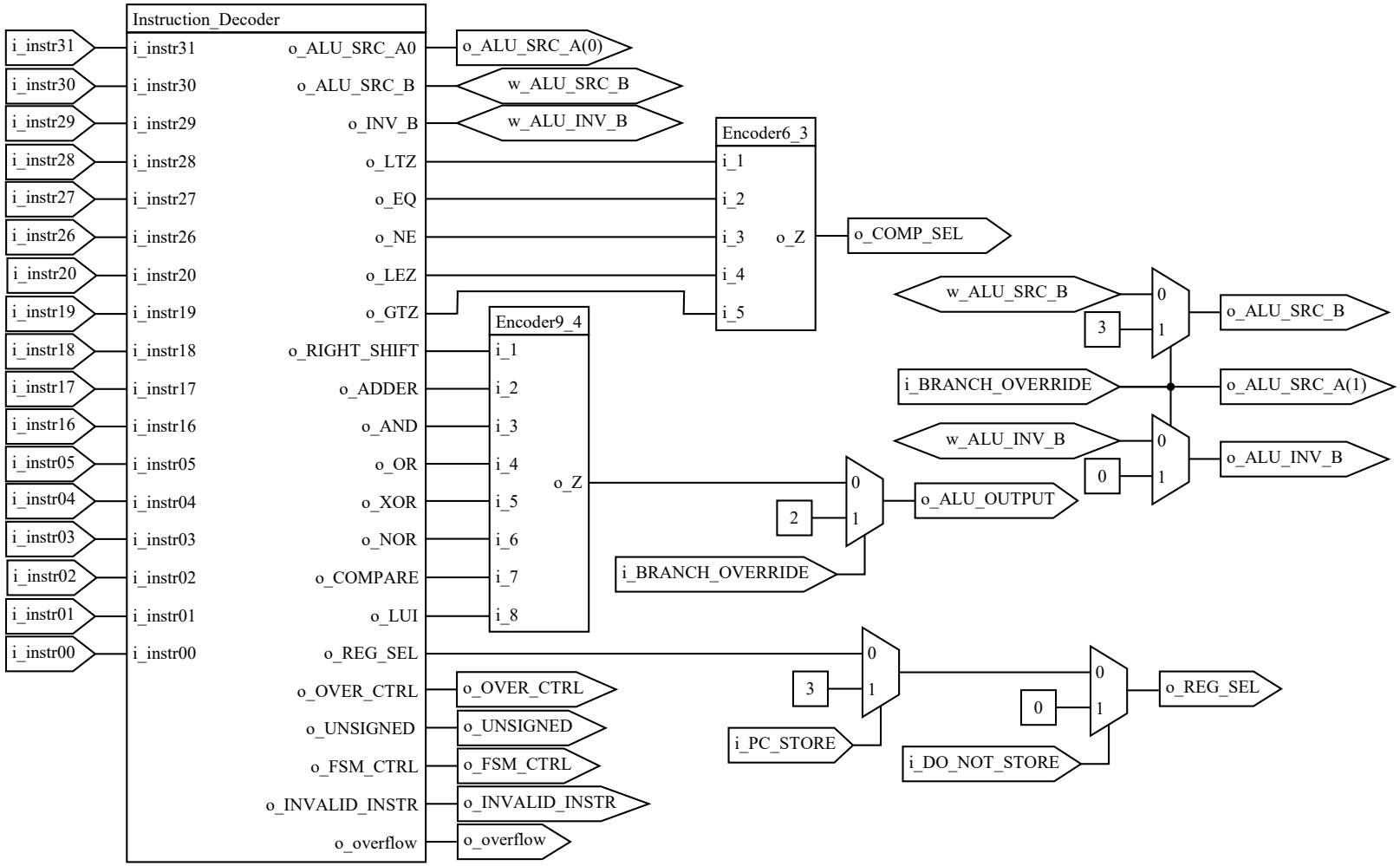

Figure 22. Instruction Decoder/Encoder Schematic

Table 51. Instruction_Dec_Enc Inputs and Outputs

Port Name	Type	Width	Description
i_instr31	Input	1	Instruction bit 31
i_instr30	Input	1	Instruction bit 30
i_instr29	Input	1	Instruction bit 29
i_instr28	Input	1	Instruction bit 28
i_instr27	Input	1	Instruction bit 27
i_instr26	Input	1	Instruction bit 26
i_instr20	Input	1	Instruction bit 20
i_instr19	Input	1	Instruction bit 19
i_instr18	Input	1	Instruction bit 18
i_instr17	Input	1	Instruction bit 17
i_instr16	Input	1	Instruction bit 16
i_instr05	Input	1	Instruction bit 5
i_instr04	Input	1	Instruction bit 4
i_instr03	Input	1	Instruction bit 3
i_instr02	Input	1	Instruction bit 2
i_instr01	Input	1	Instruction bit 1
i_instr00	Input	1	Instruction bit 0
i_BRANCH_OVERRIDE	Input	1	Branch Override
i_DO_NOT_STORE	Input	1	Do Not Store
i_PC_STORE	Input	1	PC Store
o_ALU_SRC_A	Output	2	ALU Source A Select Control Signal
o_ALU_SRC_B	Output	2	ALU Source B Select Control Signal
o_ALU_INV_B	Output	1	ALU Invert B Select Control Signal
o_COMP_SEL	Output	3	Comparison Select Control Signal
o_ALU_OUTPUT	Output	4	ALU Output Select Control Signal
o_REG_SEL	Output	2	Destination Register Select Control Signal
o_UNSIGNED	Output	1	Comparator Unsigned Control Signal
o_overflow	Output	1	Comparator Overflow Control Signal
o_imm_extend	Output	1	Determine whether immediate is zero or sign extended
o_FSM_CTRL	Output	2	FSM Next State Control Signal
o_INVALID_INSTR	Output	1	FSM Invalid Intruction Control Signal

2.21 Finite State Machine Control Signal Generator

In addition to the instruction dependent controls, there are also state dependent controls that must be determined. These control signals either modify the instruction dependent controls as seen in the Instruction Decoder/Encoder, enable the Controller's instruction register, or directly impact the flow of data through the Datapath. The MEM_READ signal enables memory to read a 32-bit word from the specified memory address. The MEM_WRITE signal enables memory to write a 32-bit word to the specified memory address. The MEM_ADDRESS_SEL signal determines whether the memory address is from the output of the ALU or from the PC register. The STORE_FROM_MEM signal determines if the data input to the GPR Bank should be data from memory (1) or from the ALU output (0). PC_STORE forces the REG_SEL signal to be 11 when PC_STORE is 1. PC_EN causes the PC register in the Datapath to be enabled and to store the value $PC = PC + 4$ to the PC register. DO_NOT_STORE forces the REG_SEL signal to be 00 when DO_NOT_STORE is 1. BRANCH_OVERRIDE causes the ALU Source A to be the PC register, ALU Source B to be either the specified offset $<<$ 2 or 4 depending on the result of the branch condition, ALU Invert B to be 0, and ALU Output to be from the adder. STORE_INSTRUCTION enables the instruction register so that an instruction stored in memory can be stored to the instruction register. Table 52 shows the values of these signals depending on the state of the finite state machine.

Table 52. State Dependent Control Signals

Instruction	State															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MEM_READ	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
MEM_WRITE	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
MEM_ADDRESS_SEL	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
STORE_FROM_MEM	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
PC_STORE	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
PC_EN	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
DO_NOT_STORE	1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1
BRANCH_OVERRIDE	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
STORE_INSTRUCTION	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The logic to determine MEM_READ is shown here:

$$
\begin{aligned}
o_MEM_READ &= \overline{i_state(3)}\ \overline{i_state(2)}\ \overline{i_state(1)}\ i_state(0) + \cdots \\
&\quad \overline{i_state(3)}\ i_state(2)\ i_state(1)\ \overline{i_state(0)} \\
o_MEM_READ &= \overline{i_state(3)}\ [\overline{i_state(2)}\ \overline{i_state(1)} + \cdots \\
&\quad i_state(2)\ i_state(1)]\ \overline{i_state(0)} \\
o_MEM_READ &= \overline{\overline{i_state(3)}\ [\overline{i_state(2)}\ \overline{i_state(1)}} + \cdots \\
&\quad \overline{i_state(2)\ i_state(1)]\ \overline{i_state(0)}} \\
o_MEM_READ &= \overline{i_state(3) + [i_state(2) + i_state(1)]} \\
&\quad \overline{[\overline{i_state(2)} + \overline{i_state(1)}] + i_state(0)} \\
o_MEM_READ &= \overline{i_state(3) + i_state(2)\ \overline{i_state(2)} + \cdots} \\
&\quad \overline{\overline{i_state(1)}\ i_state(2) + i_state(1)\ \overline{i_state(2)} + \cdots} \\
&\quad \overline{i_state(1)\ \overline{i_state(1)} + i_state(0)} \\
o_MEM_READ &= \overline{i_state(3) + \overline{i_state(1)}\ i_state(2) + \cdots} \\
&\quad \overline{i_state(1)\ \overline{i_state(2)} + i_state(0)}
\end{aligned}
\tag{91}
$$

$$w_fsm_00 = NAND(INV(i_state(1)), i_state(2))$$

$$w_fsm_01 = NAND(i_state(1), INV(i_state(2)))$$

$$o_MEM_READ = \overline{i_state(3) + \overline{w_fsm_00} + \overline{w_fsm_01} + i_state(0)}$$

$$w_fsm_02 = NAND(INV(i_state(3)), INV(i_state(0)))$$

$$w_fsm_03 = NAND(w_fsm_00, w_fsm_01)$$

$$o_MEM_READ = \overline{w_fsm_03 + w_fsm_02}$$

$$w_fsm_22 = NAND(INV(w_fsm_03), INV(w_fsm_02))$$

$$o_MEM_READ = \overline{w_fsm_22}$$

$$o_MEM_READ = INV(w_fsm_22)$$

The logic to determine MEM_WRITE is shown here:

$$
\begin{aligned}
o_MEM_WRITE &= i_state(3)\ \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)} \\
o_MEM_WRITE &= \overline{\overline{i_state(3)\ \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)}}} \\
o_MEM_WRITE &= \overline{\overline{i_state(3)} + i_state(2) + i_state(1) + i_state(0)} \\
w_fsm_04 &= NAND(i_state(3), INV(i_state(2))) \\
w_fsm_05 &= NAND(INV(i_state(1)), INV(i_state(0))) \\
o_MEM_WRITE &= \overline{w_fsm_04 + w_fsm_05} \\
w_fsm_12 &= NAND(INV(w_fsm_04), INV(w_fsm_05)) \\
o_MEM_WRITE &= \overline{w_fsm_12} \\
o_MEM_WRITE &= INV(w_fsm_12)
\end{aligned}
\tag{92}
$$

To determine MEM_ADDRESS_SEL, a Karnaugh Map was used as shown in Table 53.

Table 53. MEM_ADDRESS_SEL Karnaugh Map

i_state(1)/ i_state(0)	i_state(3)/i_state(2)			
	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	0	0	0	0
10	1	0	0	0

The logic to determine MEM_ADDRESS_SEL is:

$$
\begin{aligned}
o_MEM_ADDRESS_SEL &= \overline{i_state(3)}\ i_state(2) + \cdots \\
&\quad \overline{i_state(3)}\ i_state(0) + \overline{i_state(3)}\ i_state(1) + \cdots \\
&\quad i_state(3)\ \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)} \\
o_MEM_ADDRESS_SEL &= \overline{i_state(3)}\ [i_state(2) + i_state(1) + \cdots \\
&\quad i_state(0)] + i_state(3)\ \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)} \\
o_MEM_ADDRESS_SEL &= \overline{\overline{i_state(3)}\ [i_state(2) + i_state(1) + \cdots} \\
&\quad \overline{i_state(0)] + i_state(3)\ \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)}} \\
o_MEM_ADDRESS_SEL &= \overline{[i_state(3) + \overline{i_state(2)}\ \overline{i_state(1)}\ \overline{i_state(0)}]} \\
&\quad \overline{[\overline{i_state(3)} + i_state(2) + i_state(1) + i_state(0)]} \\
o_MEM_ADDRESS_SEL &= \overline{[i_state(3) + \overline{i_state(2)}\ \overline{w_fsm_05}]} \\
&\quad \overline{[w_fsm_04 + w_fsm_05]} \\
w_fsm_15 &= NAND(INV(i_state(2)), INV(w_fsm_05)) \\
o_MEM_ADDRESS_SEL &= \overline{[i_state(3) + \overline{w_fsm_15}]\ w_fsm_12} \\
w_fsm_06 &= NAND(INV(i_state(3)), w_fsm_15) \\
o_MEM_ADDRESS_SEL &= \overline{w_fsm_06\ w_fsm_12} \\
o_MEM_ADDRESS_SEL &= NAND(w_fsm_06, w_fsm_12)
\end{aligned}
\tag{93}
$$

The logic to determine STORE_FROM_MEM is:

$$
\begin{aligned}
o_STORE_FROM_MEM &= \overline{i_state(3)}\ i_state(2)\ i_state(1)\ i_state(0) \\
o_STORE_FROM_MEM &= \overline{\overline{i_state(3)}\ i_state(2)\ i_state(1)\ i_state(0)} \\
o_STORE_FROM_MEM &= \overline{i_state(3) + \overline{i_state(2)} + \cdots} \\
&\quad\ \overline{\overline{i_state(1)} + \overline{i_state(0)}} \\
w_fsm_07 &= NAND(INV(i_state(3)), i_state(2)) \\
w_fsm_08 &= NAND(i_state(1), i_state(0)) \\
o_STORE_FROM_MEM &= \overline{w_fsm_07 + w_fsm_08} \\
w_fsm_09 &= NAND(INV(w_fsm_07), INV(w_fsm_08)) \\
o_STORE_FROM_MEM &= \overline{w_fsm_09} \\
o_STORE_FROM_MEM &= INV(w_fsm_09)
\end{aligned}
\tag{94}
$$

The logic to determine PC_STORE is:

$$
\begin{aligned}
o_PC_STORE &= \overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)}\ i_state(0) \\
o_PC_STORE &= \overline{\overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)}\ i_state(0)} \\
o_PC_STORE &= \overline{i_state(3) + \overline{i_state(2)} + i_state(1) + \overline{i_state(0)}} \\
w_fsm_10 &= NAND(INV(i_state(1)), i_state(0)) \\
o_PC_STORE &= \overline{w_fsm_07 + w_fsm_10} \\
w_fsm_11 &= NAND(INV(w_fsm_07), INV(w_fsm_10)) \\
o_PC_STORE &= \overline{w_fsm_11} \\
o_PC_STORE &= INV(w_fsm_11)
\end{aligned}
\tag{95}
$$

The logic to determine PC_EN is:

$$
\begin{aligned}
o_PC_EN &= \overline{i_state(3)}\ \overline{i_state(2)}\ i_state(1)\ i_state(0) + \cdots \\
&\quad\ i_state(3)\ \overline{i_state(2)}\ i_state(1)\ \overline{i_state(0)} \\
o_PC_EN &= \overline{i_state(2)}\ i_state(1)[\overline{i_state(3)}\ i_state(0) + \cdots \\
&\quad\ i_state(3)\ \overline{i_state(0)}] \\
o_PC_EN &= \overline{\overline{i_state(2)}\ i_state(1)[\overline{i_state(3)}\ i_state(0) + \cdots} \\
&\quad\ \overline{\overline{i_state(3)}\ \overline{i_state(0)}]} \\
o_PC_EN &= \overline{i_state(2) + \overline{i_state(1)} + \overline{[\overline{i_state(3)}\ i_state(0) + \cdots}} \\
&\quad\ \overline{\overline{i_state(3)}\ \overline{i_state(0)}]} \\
o_PC_EN &= \overline{i_state(2) + \overline{i_state(1)} + \cdots} \\
&\quad\ \overline{[\overline{i_state(3)}\ i_state(0)]}\,\overline{[i_state(3)\ \overline{i_state(0)}]} \\
o_PC_EN &= \overline{i_state(2) + \overline{i_state(1)} + [i_state(3) + \overline{i_state(0)}]} \\
&\quad\ \overline{[\overline{i_state(3)}\ i_state(0)]} \\
o_PC_EN &= \overline{i_state(2) + \overline{i_state(1)} + i_state(3)\ i_state(0) + \cdots} \\
&\quad\ \overline{\overline{i_state(3)}\ \overline{i_state(0)}} \\
w_fsm_24 &= NAND(i_state(3), i_state(0)) \\
o_PC_EN &= \overline{w_fsm_01 + \overline{w_fsm_24} + \overline{w_fsm_02}} \\
w_fsm_25 &= NAND(w_fsm_24, w_fsm_02) \\
o_PC_EN &= \overline{w_fsm_01 + w_fsm_25} \\
w_fsm_26 &= NAND(INV(w_fsm_01), INV(w_fsm_25)) \\
o_PC_EN &= \overline{w_fsm_26}
\end{aligned}
\tag{96}
$$

To determine DO_NOT_STORE, a Karnaugh Map was used as shown in Table 54.

Table 54. DO_NOT_STORE Karnaugh Map

i_state(1)/ i_state(0)	i_state(3)/i_state(2)			
	00	01	11	10
00	1	1	1	1
01	1	0	1	1
11	0	0	1	1
10	1	1	1	1

The logic to determine DO_NOT_STORE is:

$$
\begin{aligned}
o_DO_NOT_STORE &= i_state(3) + \overline{i_state(0)} + \cdots \\
&\quad \overline{i_state(3)}\ \overline{i_state(2)}\ \overline{i_state(1)} \\
o_DO_NOT_STORE &= \overline{\overline{i_state(3) + \overline{i_state(0)} + \cdots}} \\
&\quad \overline{\overline{i_state(3)}\ \overline{i_state(2)}\ \overline{i_state(1)}} \\
o_DO_NOT_STORE &= \overline{\overline{i_state(3)}\ i_state(0)\ [i_state(3) + \cdots} \\
&\quad \overline{i_state(2) + i_state(1)]} \\
o_DO_NOT_STORE &= \overline{\overline{i_state(3)}\ i_state(0)\ [i_state(2) + i_state(1)]} \\
w_fsm_27 &= NAND(INV(i_state(3)), i_state(0)) \\
o_DO_NOT_STORE &= \overline{\overline{w_fsm_27}\ w_fsm_14} \\
o_DO_NOT_STORE &= NAND(INV(w_fsm_27), w_fsm_14)
\end{aligned}
\tag{97}
$$

The logic to determine BRANCH_OVERRIDE is:

$$
\begin{aligned}
o_BRANCH_OVERRIDE &= \overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)}\ \overline{i_state(0)} \\
&\quad + \overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)}\ i_state(0) \\
o_BRANCH_OVERRIDE &= \overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)} \\
o_BRANCH_OVERRIDE &= \overline{\overline{i_state(3)}\ i_state(2)\ \overline{i_state(1)}} \\
o_BRANCH_OVERRIDE &= \overline{i_state(3) + \overline{i_state(2)} + i_state(1)} \\
o_BRANCH_OVERRIDE &= \overline{w_fsm_07 + i_state(1)} \\
w_fsm_19 &= NAND(INV(w_fsm_07), INV(i_state(1))) \\
o_BRANCH_OVERRIDE &= \overline{w_fsm_19} \\
o_BRANCH_OVERRIDE &= INV(w_fsm_19)
\end{aligned}
\tag{98}
$$

The logic to determine STORE_INSTRUCTION is:

$$
\begin{aligned}
o_STORE_INSTRUCTION &= \overline{i_state(3)}\ \overline{i_state(2)}\ \overline{i_state(1)}\ i_state(0) \\
o_STORE_INSTRUCTION &= \overline{\overline{i_state(3)}\ \overline{i_state(2)}\ \overline{i_state(1)}\ i_state(0)} \\
o_STORE_INSTRUCTION &= \overline{i_state(3) + _state(2) + \cdots} \\
&\quad \overline{i_state(1) + \overline{i_state(0)}} \\
w_fsm_20 &= NAND(INV(i_state(3)), INV(i_state(2))) \\
o_STORE_INSTRUCTION &= \overline{w_fsm_20 + w_fsm_10} \\
w_fsm_21 &= NAND(INV(w_fsm_20), INV(w_fsm_10)) \\
o_STORE_INSTRUCTION &= \overline{w_fsm_21} \\
o_STORE_INSTRUCTION &= INV(w_fsm_21)
\end{aligned}
\tag{99}
$$

Table 55. FSM_Control_Signal_Generator Signal Summary

Signal Name	Logic
w_fsm_00	NAND(i_state(2),INV(i_state(1)))
w_fsm_01	NAND(INV(i_state(2)),i_state(1))
w_fsm_02	NAND(INV(i_state(3)),INV(i_state(0)))
w_fsm_03	NAND(w_fsm_00,w_fsm_01)
w_fsm_04	NAND(i_state(3),INV(i_state(2))
w_fsm_05	NAND(INV(i_state(1)),INV(i_state(0)))
w_fsm_06	NAND(INV(i_state(3)),w_fsm_15)
w_fsm_07	NAND(INV(i_state(3)),i_state(2))
w_fsm_08	NAND(i_state(1),i_state(0))
w_fsm_09	NAND(INV(w_fsm_07),INV(w_fsm_08))
w_fsm_10	NAND(INV(i_state(1)),i_state(0))
w_fsm_11	NAND(INV(w_fsm_07),INV(w_fsm_10))
w_fsm_12	NAND(INV(w_fsm_04),INV(w_fsm_05))
w_fsm_14	NAND(INV(i_state(2)),INV(i_state(1)))
w_fsm_15	NAND(INV(i_state(2)),INV(w_fsm_05))
w_fsm_19	NAND(INV(w_fsm_07),INV(i_state(1)))
w_fsm_20	NAND(INV(i_state(3)),INV(i_state(2)))
w_fsm_21	NAND(INV(w_fsm_20),INV(w_fsm_10))
w_fsm_22	NAND(INV(w_fsm_03),INV(w_fsm_02))
w_fsm_24	NAND(i_state(3),i_state(0))
w_fsm_25	NAND(w_fsm_24,w_fsm_02)
w_fsm_26	NAND(INV(w_fsm_01),INV(w_fsm_25))
w_fsm_27	NAND(INV(i_state(3)),i_state(0))
MEM_READ	INV(w_fsm_22)
MEM_WRITE	INV(w_fsm_12)
MEM_ADDRESS_SEL	NAND(w_fsm_06,w_fsm_12)
STORE_FROM_MEM	INV(w_fsm_09)
PC_STORE	INV(w_fsm_11)
PC_EN	INV(w_fsm_26)
DO_NOT_STORE	NAND(INV(w_fsm_27),w_fsm_14)
BRANCH_OVERRIDE	INV(w_fsm_19)
STORE_INSTRUCTION	INV(w_fsm_21)

Table 56. FSM_Control_Signal_Generator Inputs and Outputs

Port Name	Type	Width	Description
i_state	Input	4	Current State
o_MEM_READ	Output	1	Memory Read Enable
o_MEM_WRITE	Output	1	Memory Write Enable
o_MEM_ADDRESS_SEL	Output	1	Memory Address
o_STORE_FROM_MEM	Output	1	Store Memory Data to Register in GPR Bank
o_PC_STORE	Output	1	Store ALU Output to PC Register
o_PC_EN	Output	1	Store PC+4 to the PC Register
o_DO_NOT_STORE	Output	1	Do Not Store the ALU Output to the GPR Bank
o_BRANCH_OVERRIDE	Output	1	Override ALU Source A, Source B, Invert B, and Output Signals
o_STORE_INSTRUCTION	Output	1	Store Instruction From Memory

2.22 Controller Finite State Machine (FSM)

The Finite State Machine is a Moore Model state machine with 10 states (0 through 9) contained within the Controller FSM module. The current state is held in a 4-bit register and is always output to the FSM Control Signal Generator discussed in the previous section. The FSM Control Signal Generator, Instruction Decoder/Encoder, and an Enabled Register to store instructions are contained in the Controller FSM module. A series of multiplexers determine the next state of the FSM based on the current state as well as control signals from the Instruction Decoder/Encoder. Table 57 describes the FSM states. In this table, i_MEM_READY is a signal from memory indicating that the requested memory read or write operation was completed. The w_FSM_CTRL signal is output from the Instruction Decoder/Encoder. It takes on the value 01 for branch instructions (BGEZ, BLTZ, BEQ, BNE, BLEZ, and BGTZ), 10 for load word (LW), 11 for store word (SW), and 00 for all other

instructions. The states are designed so that i_MEM_READY can return to 0 before it is sampled in state 0 to determine if execution can proceed to state 1. States with no condition will proceed to the next state on the next clock cycle. States with a w_FSM_CTRL condition will proceed to the appropriate next state on the next clock cycle as well. The only condition which may result in the next state transition being held up for a few clock cycles is the i_MEM_READY condition as it may take some time for memory to complete an operation.

Table 57. FSM States

Current State	Next State	Next State Condition	Instructions
0	1	$i_MEM_READY = 1$	All
1	2	$i_MEM_READY = 0$	All
2	3	$w_FSM_CTRL = 00$	All
	4	$w_FSM_CTRL = 01$	
	6	$w_FSM_CTRL = 10$	
	8	$w_FSM_CTRL = 11$	
3	0	None	SLL, NOP, SRL, SRA, SLLV, SRLV, SRAV, ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU, ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI
4	5	None	BGEZ, BLTZ, BEQ, BNE, BLEZ, BGTZ
5	0	None	
6	7	$i_MEM_READY = 1$	LW
7	10	$i_MEM_READY = 0$	
8	9	$i_MEM_READY = 1$	SW
9	10	$i_MEM_READY = 0$	
10	0	None	LW, SW

The following table shows the connections inside the Controller FSM module. Note the use of constants denoted k_zero_4, k_one_4, k_two_4, k_three_4, k_four_4, k_five_4, k_six_4, k_seven_4, k_eight_4, k_nine_4, and k_ten_4that represent the numbers 0 thorugh 9 in four bit format. Figure 23 shows a schematic of these connections.

Table 58. Controller FSM Connections

w_next_state01 MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	k_zero_4
i_1	Input	1	k_one_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state01
w_next_state12 MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	k_two_4
i_1	Input	1	k_one_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state12
w_next_state2 MUX4_4			
I/O	Type	Width	Connection
i_0	Input	1	k_three_4
i_1	Input	1	k_four_4
i_2	Input	1	k_six_4
i_3	Input	1	k_eight_4
i_S	Input	1	w_FSM_CTRL
o_Z	Output	1	w_next_state2
w_next_state67 MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	k_six_4
i_1	Input	1	k_seven_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state67
w_next_state70 MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	k_ten_4
i_1	Input	1	k_seven_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state70
w_next_state89 MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	k_eight_4
i_1	Input	1	k_nine_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state89
w_next_state90 MUX2_4			
I/O	Type	Width	Connection

Table 58 – *Continued on next page*

Table 58 – *Continued from previous page*

i_0	Input	1	k_ten_4
i_1	Input	1	k_nine_4
i_S	Input	1	i_MEM_READY
o_Z	Output	1	w_next_state90
w_next_state01 MUX16_4			
I/O	Type	Width	Connection
i_0	Input	1	w_next_state_01
i_1	Input	1	w_next_state_12
i_2	Input	1	w_next_state_2
i_3	Input	1	k_zero_4
i_4	Input	1	k_five_4
i_5	Input	1	k_zero_4
i_6	Input	1	w_next_state_67
i_7	Input	1	w_next_state_70
i_8	Input	1	w_next_state_89
i_9	Input	1	w_next_state_90
i_10	Input	1	k_zero_4
i_11	Input	1	k_zero_4
i_12	Input	1	k_zero_4
i_13	Input	1	k_zero_4
i_14	Input	1	k_zero_4
i_15	Input	1	k_zero_4
i_S	Input	1	w_state
o_Z	Output	1	w_next_state0
w_next_state MUX2_4			
I/O	Type	Width	Connection
i_0	Input	1	w_next_state0
i_1	Input	1	k_zero_4
i_S	Input	1	w_INVALID_INSTR
o_Z	Output	1	w_next_state
State Register myRegN			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_D	Input	4	w_next_state
o_Q	Output	4	w_state
FSM Control Signal Generator			
I/O	Type	Width	Connection
i_state	Input	4	w_state
o_MEM_READ	Output	1	o_MEM_READ

Table 58 – *Continued on next page*

Table 58 – *Continued from previous page*

o_MEM_WRITE	Output	1	o_MEM_WRITE
o_MEM_ADDRESS_SEL	Output	1	o_MEM_ADDRESS_SEL
o_STORE_FROM_MEM	Output	1	o_STORE_FROM_MEM
o_PC_STORE	Output	1	w_PC_STORE
o_PC_EN	Output	1	o_PC_EN
o_DO_NOT_STORE	Output	1	w_DO_NOT_STORE
o_BRANCH_OVERRIDE	Output	1	w_BRANCH_OVERRIDE
o_STORE_INSTRUCTION	Output	1	w_STORE_INSTRUCTION
Instruction Register Enabled_Register			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_en	Input	1	w_STORE_INSTRUCTION
i_data	Input	32	i_MEM_OUT
o_Q	Output	32	w_instruction
Instruction Decoder/Encoder			
I/O	Type	Width	Connection
i_instr31	Input	1	w_instruction(31)
i_instr30	Input	1	w_instruction(30)
i_instr29	Input	1	w_instruction(29)
i_instr28	Input	1	w_instruction(28)
i_instr27	Input	1	w_instruction(27)
i_instr26	Input	1	w_instruction(26)
i_instr20	Input	1	w_instruction(20)
i_instr19	Input	1	w_instruction(19)
i_instr18	Input	1	w_instruction(18)
i_instr17	Input	1	w_instruction(17)
i_instr16	Input	1	w_instruction(16)
i_instr05	Input	1	w_instruction(5)
i_instr04	Input	1	w_instruction(4)
i_instr03	Input	1	w_instruction(3)
i_instr02	Input	1	w_instruction(2)
i_instr01	Input	1	w_instruction(1)
i_instr00	Input	1	w_instruction(0)
i_BRANCH_OVERRIDE	Input	1	w_BRANCH_OVERRIDE
i_DO_NOT_STORE	Input	1	w_DO_NOT_STORE
i_PC_STORE	Input	1	w_PC_STORE
o_ALU_SRC_A	Output	2	o_ALU_SRC_A
o_ALU_SRC_B	Output	2	o_ALU_SRC_B
o_ALU_INV_B	Output	1	o_ALU_INV_B

Table 58 – *Continued on next page*

Table 58 – *Continued from previous page*

o_COMP_SEL	Output	3	o_COMP_SEL
o_ALU_OUTPUT	Output	4	o_ALU_OUTPUT
o_REG_SEL	Output	2	o_REG_SEL
o_UNSIGNED	Output	1	o_UNSIGNED
o_overflow	Output	1	o_overflow
o_imm_extend	Output	1	o_imm_extend
o_FSM_CTRL	Output	2	w_FSM_CTRL
o_INVALID_INSTR	Output	1	w_INVALID_INSTR

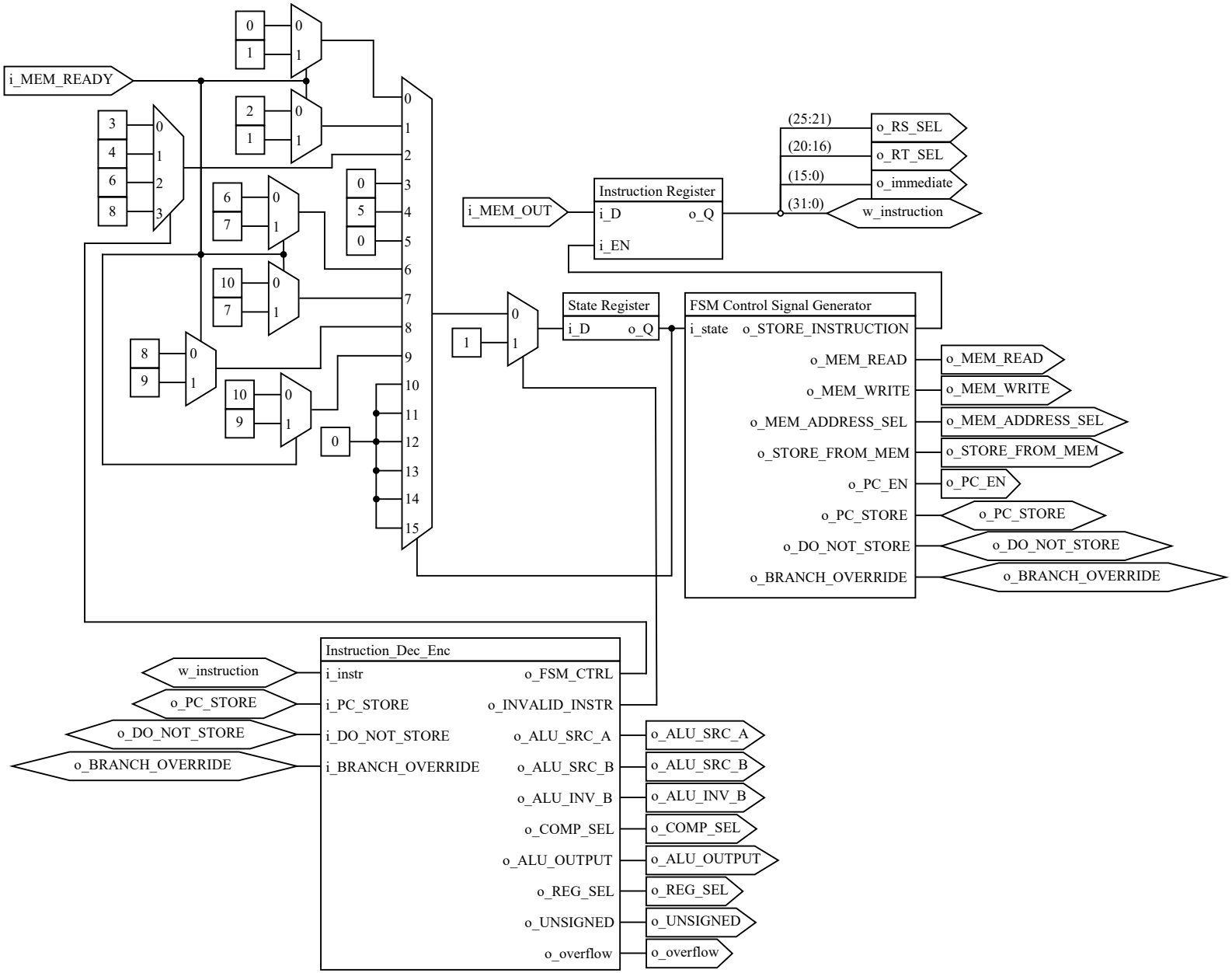

Figure 23. Controller FSM Schematic

Table 59. Controller FSM Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock
i_reset	Input	1	Reset
i_MEM_OUT	Input	32	Output from memory
i_MEM_READY	Input	1	Memory Completed read/write operation
o_MEM_READ	Output	1	Enable memory read operation
o_MEM_WRITE	Output	1	Enable memory Write Operation
o_MEM_ADDRESS_SEL	Output	1	Determine if memory address should be PC register data or ALU result
o_STORE_FROM_MEM	Output	1	Store ALU Result or memory data to GPR Bank
o_PC_EN	Output	1	Determine if the PC should store PC+4
o_ALU_SRC_A	Output	2	Select Source A Input
o_ALU_SRC_B	Output	2	Select Source B Input
o_ALU_INV_B	Output	1	Invert the B Input
o_COMP_SEL	Output	3	Select the Comparator Output
o_OVER_CTRL	Output	2	Overflow Control Signal
o_ALU_OUTPUT	Output	4	Select the ALU Output
o_REG_SEL	Output	2	Determine which GPR is enabled and whether the PC register is enabled
o_UNSIGNED	Output	1	Comparator Unsigned Control Signal
o_overflow	Output	1	Comparator Overflow Control Signal
o_imm_extend	Input	1	Determine whether immediate is zero or sign extended
o_RS_SEL	Output	5	Select RS Register
o_RT_SEL	Output	5	Select RT Register
o_immediate	Output	16	Immediate - also contains the shift amount for

Table 59 – *Continued on next page*

Table 59 – *Continued from previous page*

Port Name	Type	Width	Description
			the left shift and right shift modules (bits 10 downto 6), RS Register select signal (bits 15downto 11), and bit 0 determines the difference between a logical right shift and an arithmetic right shift

2.23 Top Level of Basic MIPS Architecture

The top level of the Basic MIPS Architecture connects the Datapath and Controller so they function together as a cohesive Basic MIPS unit. The only connections to the Basic MIPS module are the clock input, reset input, memory data inputs and outputs, the memory read and write enable signals, and the memory ready signal. The following tables show the Basic MIPS connections and the inputs and outputs.

Table 60. Basic MIPS Connections

Controller FSM			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_MEM_OUT	Input	32	i_MEM_OUT
i_MEM_READY	Input	1	i_MEM_READY
o_MEM_READ	Output	1	o_MEM_READ
o_MEM_WRITE	Output	1	o_MEM_WRITE
o_MEM_ADDRESS_SEL	Output	1	w_MEM_ADDRESS_SEL
o_STORE_FROM_MEM	Output	1	w_STORE_FROM_MEM
o_PC_EN	Output	1	w_PC_EN

Table 60 – *Continued on next page*

Table 60 – *Continued from previous page*

o_ALU_SRC_A	Output	2	w_ALU_SRC_A
o_ALU_SRC_B	Output	2	w_ALU_SRC_B
o_ALU_INV_B	Output	1	w_ALU_INV_B
o_COMP_SEL	Output	3	w_COMP_SEL
o_OVER_CTRL	Output	2	w_OVER_CTRL
o_ALU_OUTPUT	Output	4	w_ALU_OUTPUT
o_REG_SEL	Output	2	w_REG_SEL
o_UNSIGNED	Output	1	w_UNSIGNED
o_overflow	Output	1	w_overflow
o_imm_extend	Input	1	w_imm_extend
o_RS_SEL	Output	5	w_RS_SEL
o_RT_SEL	Output	5	w_RT_SEL
o_immediate	Output	16	w_immediate
Datapath			
I/O	Type	Width	Connection
i_clk	Input	1	i_clk
i_reset	Input	1	i_reset
i_ALU_SRC_A	Input	2	w_ALU_SRC_A
i_ALU_SRC_B	Input	2	w_ALU_SRC_B
i_ALU_INV_B	Input	1	w_ALU_INV_B
i_COMP_SEL	Input	3	w_COMP_SEL
i_OVER_CTRL	Input	2	w_OVER_CTRL
i_ALU_OUTPUT	Input	4	w_ALU_OUTPUT
i_RS_SEL	Input	5	w_RS_SEL
i_RT_SEL	Input	5	w_RT_SEL
i_imm	Input	16	w_immediate
i_unsigned	Input	1	w_UNSIGNED
i_overflow	Input	1	w_overflow
i_imm_extend	Input	1	w_imm_extend
i_STORE_FROM_MEM	Input	1	w_STORE_FROM_MEM
i_REG_SEL	Input	2	w_REG_SEL
i_PC_EN	Input	1	w_PC_EN
i_MEM_ADDRESS_SEL	Input	1	w_MEM_ADDRESS_SEL
i_MEM_OUT	Input	32	i_MEM_OUT
o_MEM_ADDRESS	Output	32	o_MEM_ADDRESS
o_RT_DATA	Output	32	o_MEM_IN

Table 61. Basic MIPS Inputs and Outputs

Port Name	Type	Width	Description
i_clk	Input	1	Clock
i_reset	Input	1	Reset
i_MEM_OUT	Input	32	Output from memory
i_MEM_READY	Input	1	Memory Completed read/write operation
o_MEM_READ	Output	1	Enable memory read operation
o_MEM_WRITE	Output	1	Enable memory Write Operation
o_MEM_IN	Output	32	Data to store to memory
o_MEM_Address	Output	32	Address used to access memory

III. Simulation Testing

Throughout the design of the Basic MIPS architecture, a simulate-as-you-go test approach was used to ensure that each submodule functioned as expected before integrating that module into higher level modules. VHDL testbench code was run against the VHDL code modules in a simulation tool called ModelSIM. Full factorial simulation testing was performed for most of the submodules. Full factorial simulation testing was impossible for higher level modules because the number of possible input vectors grew exponentially (i.e. 1 or more 32-bit inputs compounded with multiple control signals). This test approach was followed rigorously with few exceptions. The appendix contains all of the test code used to test the VHDL modules in the Basic MIPS architecture. Software simulation verification was purposefully not performed for some simple modules; however, their performance was evaluated once integrated into higher level modules. Additionally, the ALU Core and Datapath were not verified independently of the entire Basic MIPS architecture because it would have proved difficult to emulate the behavior of all the control signals correctly. Instead, Controller verification was performed to ensure it was producing the correct control signals before connecting the Controller to the Datapath and testing the function of the Datapath via. the Basic MIPS simulation test. Through simulation, Basic MIPS architecture was verified to work exactly as designed.

Appendix A. Basic MIPS VHDL Code

1.1 NAND

```
1  --| myNAND2
   --| Perfomrs the operation Z = NAND(A,B)
3  --|
   --| INPUTS:
5  --| A - Input A
   --| B - Input B
7  --|
   --| OUTPUTS:
9  --| Z - NAND(A,B)
   library IEEE;
11 use IEEE.std_logic_1164.all;

13 entity myNAND2 is
     port (i_A : in  std_logic;
15         i_B : in  std_logic;
           o_Z : out std_logic
17         );
   end myNAND2;
19
   architecture a_myNAND2 of myNAND2 is
21 begin
     o_Z <= i_A nand i_B;
23 end a_myNAND2;
```

Listing A.1. myNAND2.vhd Code

1.2 Inverter

1.2.1 1-Bit Inverter

```
--| myINV
--| Perfomrs the invert operation Z = NOT(A)
--|
--| INPUTS:
--| A - Input A
--|
--| OUTPUTS:
--| Z - NOT(A)
library IEEE;
use IEEE.std_logic_1164.all;

entity myINV is
  port (i_A : in  std_logic;
        o_Z : out std_logic);
end myINV;

architecture a_myINV of myINV is

  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;
begin
  u_myNAND2: myNAND2
  port map (i_A => i_A,
            i_B => i_A,
            o_Z => o_Z);
end a_myINV;
```

Listing A.2. myINV.vhd Code

1.2.2 N-Bit Inverter

```
--| myINV_N
--| N-bit inverter.  Each bit is inverted idependently of the
    others
--|
--| INPUTS:
--| i_A - Input to invert
--|
--| OUTPUTS:
--| o_Z - i_0 if i_S = 0 and i_1 if i_S = 1
library IEEE;
use IEEE.std_logic_1164.all;

entity myINV_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_A : in  std_logic_vector(m_width-1 downto 0);
        o_Z : out std_logic_vector(m_width-1 downto 0));
end myINV_N;

architecture a_myINV_N of myINV_N is
--| Declare components
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic);
  end component;
begin
  myINV_array: for bit_index in 0 to m_width-1 generate
    u_myINV : myINV
    port map (i_A => i_A(bit_index),
              o_Z => o_Z(bit_index));
  end generate myINV_array;
end a_myINV_N;
```

Listing A.3. myINV_N.vhd Code

1.3 Multiplexers

1.3.1 2 Input, 1-Bit Multiplexer

```
--| myMUX2_1
--| 2 input, single bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_s - Select Input
--|
--| OUTPUTS:
--| o_Z - Output i_0 if i_S = 0 and i_1 if i_S = 1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX2_1 is
  port (i_0 : in  std_logic;
        i_1 : in  std_logic;
        i_S : in  std_logic;
        o_Z : out std_logic);
end myMUX2_1;

architecture a_myMUX2_1 of myMUX2_1 is
--| Declare Components
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic);
  end component;

  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic);
  end component;
--| Declare signals
  signal w_S_n    : std_logic; -- INV(i_S)
  signal w_S_ni_0 : std_logic; -- NAND(INV(i_S),i_0)
  signal w_Si_1   : std_logic; -- NAND(i_S,i_1)
begin
  u_myINV_S: myINV
  port map (i_A => i_S,
            o_Z => w_S_n);
```

```
    u_myNAND2_S_ni_0: myNAND2
42  port map (i_A => w_S_n,
              i_B => i_0,
44            o_Z => w_S_ni_0);
    u_myNAND2_Si_1: myNAND2
46  port map (i_A => i_S,
              i_B => i_1,
48            o_Z => w_Si_1);
    u_myNAND2_MUX_OUT: myNAND2
50  port map (i_A => w_S_ni_0,
              i_B => w_Si_1,
52            o_Z => o_Z);
end a_myMUX2_1;
```

Listing A.4. myMUX2_1.vhd Code

1.3.2 2 Input, N-Bit Multiplexer

```
--| MUX2_N
--| 2 input, N-bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_S - Select input
--|
--| OUTPUTS:
--| o_Z - Output i_0 if i_S = 0 and i_1 if i_S = 1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX2_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_0 : in  std_logic_vector(m_width-1 downto 0);
        i_1 : in  std_logic_vector(m_width-1 downto 0);
        i_S : in  std_logic;
        o_Z : out std_logic_vector(m_width-1 downto 0));
end myMUX2_N;

architecture a_myMUX2_N of myMUX2_N is
--| Declare components
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;
begin
  mux2_1_array: for bit_index in 0 to m_width-1 generate
    u_mux2_1 : myMUX2_1
    port map (i_0 => i_0(bit_index),
              i_1 => i_1(bit_index),
              i_S => i_S,
              o_Z => o_Z(bit_index));
  end generate mux2_1_array;
end a_myMUX2_N;
```

Listing A.5. myMUX2_N.vhd Code

1.3.3 4 Input, 1-Bit Multiplexer

```
--| myMUX4_1
--| 4 input, single bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_S - Select Input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
   representation of i_S
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX4_1 is
  port (i_0 : in  std_logic;
        i_1 : in  std_logic;
        i_2 : in  std_logic;
        i_3 : in  std_logic;
        i_S : in  std_logic_vector (1 downto 0);
        o_Z : out std_logic);
end myMUX4_1;

architecture a_myMUX4_1 of myMUX4_1 is
--| Declare Components
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;
--| Declare signals
  signal w_01  : std_logic; -- MUX(0,1,i_S(0))
  signal w_23 : std_logic; -- MUX(2,3,i_S(0))
begin
  u_myMUX2_1_01: myMUX2_1
  port map (i_0 => i_0,
            i_1 => i_1,
```

```
41              i_S => i_S(0),
                o_Z => w_01);
43    u_myMUX2_1_23: myMUX2_1
      port map (i_0 => i_2,
45              i_1 => i_3,
                i_S => i_S(0),
47              o_Z => w_23);
      u_myMUX2_1_0123: myMUX2_1
49    port map (i_0 => w_01,
                i_1 => w_23,
51              i_S => i_S(1),
                o_Z => o_Z);
53  end a_myMUX4_1;
```

Listing A.6. myMUX4_1.vhd Code

1.3.4 4 Input, N-Bit Multiplexer

```
--| MUX4_N
--| 4 input, N-bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_S - Select input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
    representation of i_S
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX4_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_0 : in  std_logic_vector(m_width-1 downto 0);
        i_1 : in  std_logic_vector(m_width-1 downto 0);
        i_2 : in  std_logic_vector(m_width-1 downto 0);
        i_3 : in  std_logic_vector(m_width-1 downto 0);
        i_S : in  std_logic_vector(1 downto 0);
        o_Z : out std_logic_vector(m_width-1 downto 0));
end myMUX4_N;

architecture a_myMUX4_N of myMUX4_N is
--| Declare components
  component myMUX4_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_S : in  std_logic_vector(1 downto 0);
          o_Z : out std_logic
          );
  end component;
begin
  mux4_1_array: for bit_index in 0 to m_width-1 generate
    u_mux4_1 : myMUX4_1
```

```
    port map (i_0 => i_0(bit_index),
41          i_1 => i_1(bit_index),
            i_2 => i_2(bit_index),
43          i_3 => i_3(bit_index),
            i_S => i_S,
45          o_Z => o_Z(bit_index));
  end generate mux4_1_array;
47 end a_myMUX4_N;
```

Listing A.7. myMUX4_N.vhd Code

1.3.5 8 Input, 1-Bit Multiplexer

```
--| myMUX8_1
--| 8 input, single bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_4 - Input 4
--| i_5 - Input 5
--| i_6 - Input 6
--| i_7 - Input 7
--| i_S - Select Input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
   representation of i_S
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX8_1 is
  port (i_0 : in  std_logic;
        i_1 : in  std_logic;
        i_2 : in  std_logic;
        i_3 : in  std_logic;
        i_4 : in  std_logic;
        i_5 : in  std_logic;
        i_6 : in  std_logic;
        i_7 : in  std_logic;
        i_S : in  std_logic_vector (2 downto 0);
        o_Z : out std_logic);
end myMUX8_1;

architecture a_myMUX8_1 of myMUX8_1 is
--| Declare Components
  component myMUX4_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_S : in  std_logic_vector (1 downto 0);
```

```
41          o_Z : out std_logic
            );
43    end component;
      component myMUX2_1 is
45      port (i_0 : in  std_logic;
            i_1 : in  std_logic;
47          i_S : in  std_logic;
            o_Z : out std_logic
49          );
      end component;
51  --| Declare signals
      signal w_0123 : std_logic; -- MUX(0,1,2,3,i_S(1:0))
53    signal w_4567 : std_logic; -- MUX(4,5,6,7,i_S(1:0))
  begin
55    u_myMUX4_1_0123: myMUX4_1
      port map (i_0 => i_0,
57            i_1 => i_1,
              i_2 => i_2,
59            i_3 => i_3,
              i_S => i_S(1 downto 0),
61            o_Z => w_0123);
      u_myMUX4_1_4567: myMUX4_1
63    port map (i_0 => i_4,
              i_1 => i_5,
65            i_2 => i_6,
              i_3 => i_7,
67            i_S => i_S(1 downto 0),
              o_Z => w_4567);
69    u_myMUX2_1_OUT: myMUX2_1
      port map (i_0 => w_0123,
71            i_1 => w_4567,
              i_S => i_S(2),
73            o_Z => o_Z);
  end a_myMUX8_1;
```

Listing A.8. myMUX8_1.vhd Code

1.3.6 8 Input, N-Bit Multiplexer

```
--| MUX8_N
--| 8 input, N-bit multiplexer
--|
--| INPUTS:
--| i_0 - Input 0
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_4 - Input 4
--| i_5 - Input 5
--| i_6 - Input 6
--| i_7 - Input 7
--| i_S - Select input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
    representation of i_S
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX8_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_0 : in  std_logic_vector(m_width-1 downto 0);
        i_1 : in  std_logic_vector(m_width-1 downto 0);
        i_2 : in  std_logic_vector(m_width-1 downto 0);
        i_3 : in  std_logic_vector(m_width-1 downto 0);
        i_4 : in  std_logic_vector(m_width-1 downto 0);
        i_5 : in  std_logic_vector(m_width-1 downto 0);
        i_6 : in  std_logic_vector(m_width-1 downto 0);
        i_7 : in  std_logic_vector(m_width-1 downto 0);
        i_S : in  std_logic_vector(2 downto 0);
        o_Z : out std_logic_vector(m_width-1 downto 0));
end myMUX8_N;

architecture a_myMUX8_N of myMUX8_N is
--| Declare components
  component myMUX8_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
```

```
40          i_3 : in  std_logic;
            i_4 : in  std_logic;
42          i_5 : in  std_logic;
            i_6 : in  std_logic;
44          i_7 : in  std_logic;
            i_S : in  std_logic_vector(2 downto 0);
46          o_Z : out std_logic
            );
48    end component;
  begin
50    MUX8_1_array: for bit_index in 0 to m_width-1 generate
        u_MUX8_1 : myMUX8_1
52      port map (i_0 => i_0(bit_index),
                  i_1 => i_1(bit_index),
54                i_2 => i_2(bit_index),
                  i_3 => i_3(bit_index),
56                i_4 => i_4(bit_index),
                  i_5 => i_5(bit_index),
58                i_6 => i_6(bit_index),
                  i_7 => i_7(bit_index),
60                i_S => i_S,
                  o_Z => o_Z(bit_index));
62    end generate MUX8_1_array;
  end a_myMUX8_N;
```

Listing A.9. myMUX8_N.vhd Code

1.3.7 16 Input, 1-Bit Multiplexer

```
1  --| myMUX16_1
   --| 16 input, single bit multiplexer
3  --|
   --| INPUTS:
5  --| i_0  - Input 0
   --| i_1  - Input 1
7  --| i_2  - Input 2
   --| i_3  - Input 3
9  --| i_4  - Input 4
   --| i_5  - Input 5
11 --| i_6  - Input 6
   --| i_7  - Input 7
13 --| i_8  - Input 8
   --| i_9  - Input 9
15 --| i_10 - Input 10
   --| i_11 - Input 11
17 --| i_12 - Input 12
   --| i_13 - Input 13
19 --| i_14 - Input 14
   --| i_15 - Input 15
21 --| i_S - Select input
   --|
23 --| OUTPUTS:
   --| o_Z - Output the input corresponding the the decimal
      representation of i_S
25 library IEEE;
   use IEEE.std_logic_1164.all;
27
   entity myMUX16_1 is
29   port (i_0 : in  std_logic;
           i_1 : in  std_logic;
31         i_2 : in  std_logic;
           i_3 : in  std_logic;
33         i_4 : in  std_logic;
           i_5 : in  std_logic;
35         i_6 : in  std_logic;
           i_7 : in  std_logic;
37         i_8 : in  std_logic;
           i_9 : in  std_logic;
39         i_10  : in  std_logic;
           i_11  : in  std_logic;
```

```
41        i_12  : in  std_logic;
          i_13  : in  std_logic;
43        i_14  : in  std_logic;
          i_15  : in  std_logic;
45        i_S : in  std_logic_vector (3 downto 0);
          o_Z : out std_logic);
47  end myMUX16_1;

49  architecture a_myMUX16_1 of myMUX16_1 is
    --| Declare Components
51    component myMUX8_1 is
        port (i_0 : in  std_logic;
53            i_1 : in  std_logic;
              i_2 : in  std_logic;
55            i_3 : in  std_logic;
              i_4 : in  std_logic;
57            i_5 : in  std_logic;
              i_6 : in  std_logic;
59            i_7 : in  std_logic;
              i_S : in  std_logic_vector (2 downto 0);
61            o_Z : out std_logic
              );
63    end component;
      component myMUX2_1 is
65      port (i_0 : in  std_logic;
              i_1 : in  std_logic;
67            i_S : in  std_logic;
              o_Z : out std_logic
69            );
      end component;
71  --| Declare signals
      signal w_0_7  : std_logic; -- MUX(0,1, 2, 3, 4, 5, 6, 7,i_S
       (2:0))
73    signal w_8_15 : std_logic; -- MUX(8,9,10,11,12,13,14,15,i_S
       (2:0))
    begin
75    u_myMUX8_1_0_7: myMUX8_1
      port map (i_0 => i_0,
77              i_1 => i_1,
                i_2 => i_2,
79              i_3 => i_3,
                i_4 => i_4,
81              i_5 => i_5,
```

```
            i_6 => i_6,
            i_7 => i_7,
            i_S => i_S(2 downto 0),
            o_Z => w_0_7);
    u_myMUX8_1_8_15: myMUX8_1
    port map (i_0 => i_8,
            i_1 => i_9,
            i_2 => i_10,
            i_3 => i_11,
            i_4 => i_12,
            i_5 => i_13,
            i_6 => i_14,
            i_7 => i_15,
            i_S => i_S(2 downto 0),
            o_Z => w_8_15);
    u_myMUX2_1_OUT: myMUX2_1
    port map (i_0 => w_0_7,
            i_1 => w_8_15,
            i_S => i_S(3),
            o_Z => o_Z);
end a_myMUX16_1;
```

Listing A.10. myMUX16_1.vhd Code

1.3.8 16 Input, N-Bit Multiplexer

```
--| MUX16_N
--| 16 input, N-bit multiplexer
--|
--| INPUTS:
--| i_0  - Input 0
--| i_1  - Input 1
--| i_2  - Input 2
--| i_3  - Input 3
--| i_4  - Input 4
--| i_5  - Input 5
--| i_6  - Input 6
--| i_7  - Input 7
--| i_8  - Input 8
--| i_9  - Input 9
--| i_10 - Input 10
--| i_11 - Input 11
--| i_12 - Input 12
--| i_13 - Input 13
--| i_14 - Input 14
--| i_15 - Input 15
--| i_S - Select input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
   representation of i_S
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX16_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_0 : in  std_logic_vector(m_width-1 downto 0);
        i_1 : in  std_logic_vector(m_width-1 downto 0);
        i_2 : in  std_logic_vector(m_width-1 downto 0);
        i_3 : in  std_logic_vector(m_width-1 downto 0);
        i_4 : in  std_logic_vector(m_width-1 downto 0);
        i_5 : in  std_logic_vector(m_width-1 downto 0);
        i_6 : in  std_logic_vector(m_width-1 downto 0);
        i_7 : in  std_logic_vector(m_width-1 downto 0);
        i_8 : in  std_logic_vector(m_width-1 downto 0);
        i_9 : in  std_logic_vector(m_width-1 downto 0);
```

```
40        i_10  : in  std_logic_vector(m_width-1 downto 0);
          i_11  : in  std_logic_vector(m_width-1 downto 0);
42        i_12  : in  std_logic_vector(m_width-1 downto 0);
          i_13  : in  std_logic_vector(m_width-1 downto 0);
44        i_14  : in  std_logic_vector(m_width-1 downto 0);
          i_15  : in  std_logic_vector(m_width-1 downto 0);
46        i_S : in  std_logic_vector(3 downto 0);
          o_Z : out std_logic_vector(m_width-1 downto 0));
48  end myMUX16_N;

50  architecture a_myMUX16_N of myMUX16_N is
  --| Declare components
52    component myMUX16_1 is
        port (i_0 : in  std_logic;
54            i_1 : in  std_logic;
              i_2 : in  std_logic;
56            i_3 : in  std_logic;
              i_4 : in  std_logic;
58            i_5 : in  std_logic;
              i_6 : in  std_logic;
60            i_7 : in  std_logic;
              i_8 : in  std_logic;
62            i_9 : in  std_logic;
              i_10 : in  std_logic;
64            i_11 : in  std_logic;
              i_12 : in  std_logic;
66            i_13 : in  std_logic;
              i_14 : in  std_logic;
68            i_15 : in  std_logic;
              i_S : in  std_logic_vector(3 downto 0);
70            o_Z : out std_logic
              );
72    end component;
  begin
74    MUX16_1_array: for bit_index in 0 to m_width-1 generate
        u_MUX16_1 : myMUX16_1
76      port map (i_0 => i_0(bit_index),
                  i_1 => i_1(bit_index),
78                i_2 => i_2(bit_index),
                  i_3 => i_3(bit_index),
80                i_4 => i_4(bit_index),
                  i_5 => i_5(bit_index),
82                i_6 => i_6(bit_index),
```

```
            i_7 => i_7(bit_index),
            i_8 => i_8(bit_index),
            i_9 => i_9(bit_index),
            i_10 => i_10(bit_index),
            i_11 => i_11(bit_index),
            i_12 => i_12(bit_index),
            i_13 => i_13(bit_index),
            i_14 => i_14(bit_index),
            i_15 => i_15(bit_index),
            i_S => i_S,
            o_Z => o_Z(bit_index));
  end generate MUX16_1_array;
end a_myMUX16_N;
```

Listing A.11. myMUX16_N.vhd Code

1.3.9 32 Input, 1-Bit Multiplexer

```
--| myMUX32_1
--| 32 input, single bit multiplexer
--|
--| INPUTS:
--| i_0  - Input 0
--| i_1  - Input 1
--| i_2  - Input 2
--| i_3  - Input 3
--| i_4  - Input 4
--| i_5  - Input 5
--| i_6  - Input 6
--| i_7  - Input 7
--| i_8  - Input 8
--| i_9  - Input 9
--| i_10 - Input 10
--| i_11 - Input 11
--| i_12 - Input 12
--| i_13 - Input 13
--| i_14 - Input 14
--| i_15 - Input 15
--| i_16 - Input 16
--| i_17 - Input 17
--| i_18 - Input 18
--| i_19 - Input 19
--| i_20 - Input 20
--| i_21 - Input 21
--| i_22 - Input 22
--| i_23 - Input 23
--| i_24 - Input 24
--| i_25 - Input 25
--| i_26 - Input 26
--| i_27 - Input 27
--| i_28 - Input 28
--| i_29 - Input 29
--| i_30 - Input 30
--| i_31 - Input 31
--| i_S  - Select Input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
   representation of i_S
```

```
41 library IEEE;
   use IEEE.std_logic_1164.all;
43
   entity myMUX32_1 is
45   port (i_0 : in  std_logic;
         i_1 : in  std_logic;
47       i_2 : in  std_logic;
         i_3 : in  std_logic;
49       i_4 : in  std_logic;
         i_5 : in  std_logic;
51       i_6 : in  std_logic;
         i_7 : in  std_logic;
53       i_8 : in  std_logic;
         i_9 : in  std_logic;
55       i_10  : in  std_logic;
         i_11  : in  std_logic;
57       i_12  : in  std_logic;
         i_13  : in  std_logic;
59       i_14  : in  std_logic;
         i_15  : in  std_logic;
61       i_16  : in  std_logic;
         i_17  : in  std_logic;
63       i_18  : in  std_logic;
         i_19  : in  std_logic;
65       i_20  : in  std_logic;
         i_21  : in  std_logic;
67       i_22  : in  std_logic;
         i_23  : in  std_logic;
69       i_24  : in  std_logic;
         i_25  : in  std_logic;
71       i_26  : in  std_logic;
         i_27  : in  std_logic;
73       i_28  : in  std_logic;
         i_29  : in  std_logic;
75       i_30  : in  std_logic;
         i_31  : in  std_logic;
77       i_S : in  std_logic_vector (4 downto 0);
         o_Z : out std_logic);
79 end myMUX32_1;

81 architecture a_myMUX32_1 of myMUX32_1 is
   --| Declare Components
83   component myMUX16_1 is
```

```
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          i_9 : in  std_logic;
          i_10  : in  std_logic;
          i_11  : in  std_logic;
          i_12  : in  std_logic;
          i_13  : in  std_logic;
          i_14  : in  std_logic;
          i_15  : in  std_logic;
          i_S   : in  std_logic_vector (3 downto 0);
          o_Z   : out std_logic
          );
  end component;
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;
--| Declare signals
  signal w_0_15 : std_logic; -- MUX(0, ..., 15,i_S(3:0))
  signal w_16_31  : std_logic; -- MUX(16, ..., 31,i_S(3:0))
begin
  u_myMUX16_1_0_15: myMUX16_1
  port map (i_0 => i_0,
            i_1 => i_1,
            i_2 => i_2,
            i_3 => i_3,
            i_4 => i_4,
            i_5 => i_5,
            i_6 => i_6,
            i_7 => i_7,
            i_8 => i_8,
            i_9 => i_9,
            i_10 => i_10,
```

```
127             i_11 => i_11,
                i_12 => i_12,
129             i_13 => i_13,
                i_14 => i_14,
131             i_15 => i_15,
                i_S => i_S(3 downto 0),
133             o_Z => w_0_15);
    u_myMUX16_1_16_31: myMUX16_1
135 port map (i_0 => i_16,
                i_1 => i_17,
137             i_2 => i_18,
                i_3 => i_19,
139             i_4 => i_20,
                i_5 => i_21,
141             i_6 => i_22,
                i_7 => i_23,
143             i_8 => i_24,
                i_9 => i_25,
145             i_10 => i_26,
                i_11 => i_27,
147             i_12 => i_28,
                i_13 => i_29,
149             i_14 => i_30,
                i_15 => i_31,
151             i_S => i_S(3 downto 0),
                o_Z => w_16_31);
153 u_myMUX2_1_OUT: myMUX2_1
    port map (i_0 => w_0_15,
155             i_1 => w_16_31,
                i_S => i_S(4),
157             o_Z => o_Z);
end a_myMUX32_1;
```

Listing A.12. myMUX32_1.vhd Code

1.3.10 32 Input, N-Bit Multiplexer

```
--| MUX32_N
--| 32 input, N-bit multiplexer
--|
--| INPUTS:
--| i_0  - Input 0
--| i_1  - Input 1
--| i_2  - Input 2
--| i_3  - Input 3
--| i_4  - Input 4
--| i_5  - Input 5
--| i_6  - Input 6
--| i_7  - Input 7
--| i_8  - Input 8
--| i_9  - Input 9
--| i_10 - Input 10
--| i_11 - Input 11
--| i_12 - Input 12
--| i_13 - Input 13
--| i_14 - Input 14
--| i_15 - Input 15
--| i_16 - Input 16
--| i_17 - Input 17
--| i_18 - Input 18
--| i_19 - Input 19
--| i_20 - Input 20
--| i_21 - Input 21
--| i_22 - Input 22
--| i_23 - Input 23
--| i_24 - Input 24
--| i_25 - Input 25
--| i_26 - Input 26
--| i_27 - Input 27
--| i_28 - Input 28
--| i_29 - Input 29
--| i_30 - Input 30
--| i_31 - Input 31
--| i_S - Select input
--|
--| OUTPUTS:
--| o_Z - Output the input corresponding the the decimal
   representation of i_S
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX32_N is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_0 : in  std_logic_vector(m_width-1 downto 0);
        i_1 : in  std_logic_vector(m_width-1 downto 0);
        i_2 : in  std_logic_vector(m_width-1 downto 0);
        i_3 : in  std_logic_vector(m_width-1 downto 0);
        i_4 : in  std_logic_vector(m_width-1 downto 0);
        i_5 : in  std_logic_vector(m_width-1 downto 0);
        i_6 : in  std_logic_vector(m_width-1 downto 0);
        i_7 : in  std_logic_vector(m_width-1 downto 0);
        i_8 : in  std_logic_vector(m_width-1 downto 0);
        i_9 : in  std_logic_vector(m_width-1 downto 0);
        i_10  : in  std_logic_vector(m_width-1 downto 0);
        i_11  : in  std_logic_vector(m_width-1 downto 0);
        i_12  : in  std_logic_vector(m_width-1 downto 0);
        i_13  : in  std_logic_vector(m_width-1 downto 0);
        i_14  : in  std_logic_vector(m_width-1 downto 0);
        i_15  : in  std_logic_vector(m_width-1 downto 0);
        i_16  : in  std_logic_vector(m_width-1 downto 0);
        i_17  : in  std_logic_vector(m_width-1 downto 0);
        i_18  : in  std_logic_vector(m_width-1 downto 0);
        i_19  : in  std_logic_vector(m_width-1 downto 0);
        i_20  : in  std_logic_vector(m_width-1 downto 0);
        i_21  : in  std_logic_vector(m_width-1 downto 0);
        i_22  : in  std_logic_vector(m_width-1 downto 0);
        i_23  : in  std_logic_vector(m_width-1 downto 0);
        i_24  : in  std_logic_vector(m_width-1 downto 0);
        i_25  : in  std_logic_vector(m_width-1 downto 0);
        i_26  : in  std_logic_vector(m_width-1 downto 0);
        i_27  : in  std_logic_vector(m_width-1 downto 0);
        i_28  : in  std_logic_vector(m_width-1 downto 0);
        i_29  : in  std_logic_vector(m_width-1 downto 0);
        i_30  : in  std_logic_vector(m_width-1 downto 0);
        i_31  : in  std_logic_vector(m_width-1 downto 0);
        i_S : in  std_logic_vector(4 downto 0);
        o_Z : out std_logic_vector(m_width-1 downto 0));
end myMUX32_N;

architecture a_myMUX32_N of myMUX32_N is
```

```
--| Declare components
  component myMUX32_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          i_9 : in  std_logic;
          i_10 : in  std_logic;
          i_11 : in  std_logic;
          i_12 : in  std_logic;
          i_13 : in  std_logic;
          i_14 : in  std_logic;
          i_15 : in  std_logic;
          i_16 : in  std_logic;
          i_17 : in  std_logic;
          i_18 : in  std_logic;
          i_19 : in  std_logic;
          i_20 : in  std_logic;
          i_21 : in  std_logic;
          i_22 : in  std_logic;
          i_23 : in  std_logic;
          i_24 : in  std_logic;
          i_25 : in  std_logic;
          i_26 : in  std_logic;
          i_27 : in  std_logic;
          i_28 : in  std_logic;
          i_29 : in  std_logic;
          i_30 : in  std_logic;
          i_31 : in  std_logic;
          i_S : in  std_logic_vector(4 downto 0);
          o_Z : out std_logic
          );
  end component;
begin
  MUX32_1_array: for bit_index in 0 to m_width-1 generate
    u_MUX32_1 : myMUX32_1
    port map (i_0 => i_0(bit_index),
              i_1 => i_1(bit_index),
```

```
126             i_2 => i_2(bit_index),
                i_3 => i_3(bit_index),
128             i_4 => i_4(bit_index),
                i_5 => i_5(bit_index),
130             i_6 => i_6(bit_index),
                i_7 => i_7(bit_index),
132             i_8 => i_8(bit_index),
                i_9 => i_9(bit_index),
134             i_10 => i_10(bit_index),
                i_11 => i_11(bit_index),
136             i_12 => i_12(bit_index),
                i_13 => i_13(bit_index),
138             i_14 => i_14(bit_index),
                i_15 => i_15(bit_index),
140             i_16 => i_16(bit_index),
                i_17 => i_17(bit_index),
142             i_18 => i_18(bit_index),
                i_19 => i_19(bit_index),
144             i_20 => i_20(bit_index),
                i_21 => i_21(bit_index),
146             i_22 => i_22(bit_index),
                i_23 => i_23(bit_index),
148             i_24 => i_24(bit_index),
                i_25 => i_25(bit_index),
150             i_26 => i_26(bit_index),
                i_27 => i_27(bit_index),
152             i_28 => i_28(bit_index),
                i_29 => i_29(bit_index),
154             i_30 => i_30(bit_index),
                i_31 => i_31(bit_index),
156             i_S => i_S,
                o_Z => o_Z(bit_index));
158   end generate MUX32_1_array;
  end a_myMUX32_N;
```

Listing A.13. myMUX32_N.vhd Code

1.4 Left Shift

```
--| LEFT_SHIFT.vhd
--| Implements the left shift function.  Shifts input A by
   the amount specified by SA.
--|
--| INPUTS:
--| i_A - Data Input
--| i_SA - Shift Amount
--|
--| OUTPUTS:
--| o_Z - Shifted input data (o_Z = i_A << i_SA)
library IEEE;
use IEEE.std_logic_1164.all;

entity LEFT_SHIFT is
  port (i_A : in  std_logic_vector(31 downto 0);
        i_SA  : in  std_logic_vector(4 downto 0);
        o_Z : out std_logic_vector(31 downto 0));
end LEFT_SHIFT;

architecture a_LEFT_SHIFT of LEFT_SHIFT is
  -- Declare Components
  component myMUX32_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          i_9 : in  std_logic;
          i_10  : in  std_logic;
          i_11  : in  std_logic;
          i_12  : in  std_logic;
          i_13  : in  std_logic;
          i_14  : in  std_logic;
          i_15  : in  std_logic;
          i_16  : in  std_logic;
          i_17  : in  std_logic;
          i_18  : in  std_logic;
```

```
41          i_19   : in  std_logic;
            i_20   : in  std_logic;
43          i_21   : in  std_logic;
            i_22   : in  std_logic;
45          i_23   : in  std_logic;
            i_24   : in  std_logic;
47          i_25   : in  std_logic;
            i_26   : in  std_logic;
49          i_27   : in  std_logic;
            i_28   : in  std_logic;
51          i_29   : in  std_logic;
            i_30   : in  std_logic;
53          i_31   : in  std_logic;
            i_S : in  std_logic_vector (4 downto 0);
55          o_Z : out std_logic
            );
57    end component;

59    -- Declare constants
      constant k_zero : std_logic := '0';
61
  begin
63    u_myMUX_bit31: myMUX32_1
      port map (i_0 => i_A(31),
65            i_1 => i_A(30),
              i_2 => i_A(29),
67            i_3 => i_A(28),
              i_4 => i_A(27),
69            i_5 => i_A(26),
              i_6 => i_A(25),
71            i_7 => i_A(24),
              i_8 => i_A(23),
73            i_9 => i_A(22),
              i_10 => i_A(21),
75            i_11 => i_A(20),
              i_12 => i_A(19),
77            i_13 => i_A(18),
              i_14 => i_A(17),
79            i_15 => i_A(16),
              i_16 => i_A(15),
81            i_17 => i_A(14),
              i_18 => i_A(13),
83            i_19 => i_A(12),
```

```
            i_20 => i_A(11),
            i_21 => i_A(10),
            i_22 => i_A(9),
            i_23 => i_A(8),
            i_24 => i_A(7),
            i_25 => i_A(6),
            i_26 => i_A(5),
            i_27 => i_A(4),
            i_28 => i_A(3),
            i_29 => i_A(2),
            i_30 => i_A(1),
            i_31 => i_A(0),
            i_S => i_SA,
            o_Z => o_Z(31));
    u_myMUX_bit30: myMUX32_1
    port map (i_0 => i_A(30),
            i_1 => i_A(29),
            i_2 => i_A(28),
            i_3 => i_A(27),
            i_4 => i_A(26),
            i_5 => i_A(25),
            i_6 => i_A(24),
            i_7 => i_A(23),
            i_8 => i_A(22),
            i_9 => i_A(21),
            i_10 => i_A(20),
            i_11 => i_A(19),
            i_12 => i_A(18),
            i_13 => i_A(17),
            i_14 => i_A(16),
            i_15 => i_A(15),
            i_16 => i_A(14),
            i_17 => i_A(13),
            i_18 => i_A(12),
            i_19 => i_A(11),
            i_20 => i_A(10),
            i_21 => i_A(9),
            i_22 => i_A(8),
            i_23 => i_A(7),
            i_24 => i_A(6),
            i_25 => i_A(5),
            i_26 => i_A(4),
            i_27 => i_A(3),
```

```
127             i_28 => i_A(2),
                i_29 => i_A(1),
129             i_30 => i_A(0),
                i_31 => k_zero,
131             i_S => i_SA,
                o_Z => o_Z(30));
133
    u_myMUX_bit29: myMUX32_1
135 port map (i_0 => i_A(29),
                i_1 => i_A(28),
137             i_2 => i_A(27),
                i_3 => i_A(26),
139             i_4 => i_A(25),
                i_5 => i_A(24),
141             i_6 => i_A(23),
                i_7 => i_A(22),
143             i_8 => i_A(21),
                i_9 => i_A(20),
145             i_10 => i_A(19),
                i_11 => i_A(18),
147             i_12 => i_A(17),
                i_13 => i_A(16),
149             i_14 => i_A(15),
                i_15 => i_A(14),
151             i_16 => i_A(13),
                i_17 => i_A(12),
153             i_18 => i_A(11),
                i_19 => i_A(10),
155             i_20 => i_A(9),
                i_21 => i_A(8),
157             i_22 => i_A(7),
                i_23 => i_A(6),
159             i_24 => i_A(5),
                i_25 => i_A(4),
161             i_26 => i_A(3),
                i_27 => i_A(2),
163             i_28 => i_A(1),
                i_29 => i_A(0),
165             i_30 => k_zero,
                i_31 => k_zero,
167             i_S => i_SA,
                o_Z => o_Z(29));
169 u_myMUX_bit28: myMUX32_1
```

```
    port map (i_0 => i_A(28),
171           i_1 => i_A(27),
              i_2 => i_A(26),
173           i_3 => i_A(25),
              i_4 => i_A(24),
175           i_5 => i_A(23),
              i_6 => i_A(22),
177           i_7 => i_A(21),
              i_8 => i_A(20),
179           i_9 => i_A(19),
              i_10 => i_A(18),
181           i_11 => i_A(17),
              i_12 => i_A(16),
183           i_13 => i_A(15),
              i_14 => i_A(14),
185           i_15 => i_A(13),
              i_16 => i_A(12),
187           i_17 => i_A(11),
              i_18 => i_A(10),
189           i_19 => i_A(9),
              i_20 => i_A(8),
191           i_21 => i_A(7),
              i_22 => i_A(6),
193           i_23 => i_A(5),
              i_24 => i_A(4),
195           i_25 => i_A(3),
              i_26 => i_A(2),
197           i_27 => i_A(1),
              i_28 => i_A(0),
199           i_29 => k_zero,
              i_30 => k_zero,
201           i_31 => k_zero,
              i_S => i_SA,
203           o_Z => o_Z(28));

205 u_myMUX_bit27: myMUX32_1
    port map (i_0 => i_A(27),
207           i_1 => i_A(26),
              i_2 => i_A(25),
209           i_3 => i_A(24),
              i_4 => i_A(23),
211           i_5 => i_A(22),
              i_6 => i_A(21),
```

```
213             i_7 => i_A(20),
                i_8 => i_A(19),
215             i_9 => i_A(18),
                i_10 => i_A(17),
217             i_11 => i_A(16),
                i_12 => i_A(15),
219             i_13 => i_A(14),
                i_14 => i_A(13),
221             i_15 => i_A(12),
                i_16 => i_A(11),
223             i_17 => i_A(10),
                i_18 => i_A(9),
225             i_19 => i_A(8),
                i_20 => i_A(7),
227             i_21 => i_A(6),
                i_22 => i_A(5),
229             i_23 => i_A(4),
                i_24 => i_A(3),
231             i_25 => i_A(2),
                i_26 => i_A(1),
233             i_27 => i_A(0),
                i_28 => k_zero,
235             i_29 => k_zero,
                i_30 => k_zero,
237             i_31 => k_zero,
                i_S => i_SA,
239             o_Z => o_Z(27));

241     u_myMUX_bit26: myMUX32_1
        port map (i_0 => i_A(26),
243             i_1 => i_A(25),
                i_2 => i_A(24),
245             i_3 => i_A(23),
                i_4 => i_A(22),
247             i_5 => i_A(21),
                i_6 => i_A(20),
249             i_7 => i_A(19),
                i_8 => i_A(18),
251             i_9 => i_A(17),
                i_10 => i_A(16),
253             i_11 => i_A(15),
                i_12 => i_A(14),
255             i_13 => i_A(13),
```

```
            i_14 => i_A(12),
            i_15 => i_A(11),
            i_16 => i_A(10),
            i_17 => i_A(9),
            i_18 => i_A(8),
            i_19 => i_A(7),
            i_20 => i_A(6),
            i_21 => i_A(5),
            i_22 => i_A(4),
            i_23 => i_A(3),
            i_24 => i_A(2),
            i_25 => i_A(1),
            i_26 => i_A(0),
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(26));

    u_myMUX_bit25: myMUX32_1
    port map (i_0 => i_A(25),
            i_1 => i_A(24),
            i_2 => i_A(23),
            i_3 => i_A(22),
            i_4 => i_A(21),
            i_5 => i_A(20),
            i_6 => i_A(19),
            i_7 => i_A(18),
            i_8 => i_A(17),
            i_9 => i_A(16),
            i_10 => i_A(15),
            i_11 => i_A(14),
            i_12 => i_A(13),
            i_13 => i_A(12),
            i_14 => i_A(11),
            i_15 => i_A(10),
            i_16 => i_A(9),
            i_17 => i_A(8),
            i_18 => i_A(7),
            i_19 => i_A(6),
            i_20 => i_A(5),
```

```
299             i_21 => i_A(4),
                i_22 => i_A(3),
301             i_23 => i_A(2),
                i_24 => i_A(1),
303             i_25 => i_A(0),
                i_26 => k_zero,
305             i_27 => k_zero,
                i_28 => k_zero,
307             i_29 => k_zero,
                i_30 => k_zero,
309             i_31 => k_zero,
                i_S => i_SA,
311             o_Z => o_Z(25));

313     u_myMUX_bit24: myMUX32_1
        port map (i_0 => i_A(24),
315             i_1 => i_A(23),
                i_2 => i_A(22),
317             i_3 => i_A(21),
                i_4 => i_A(20),
319             i_5 => i_A(19),
                i_6 => i_A(18),
321             i_7 => i_A(17),
                i_8 => i_A(16),
323             i_9 => i_A(15),
                i_10 => i_A(14),
325             i_11 => i_A(13),
                i_12 => i_A(12),
327             i_13 => i_A(11),
                i_14 => i_A(10),
329             i_15 => i_A(9),
                i_16 => i_A(8),
331             i_17 => i_A(7),
                i_18 => i_A(6),
333             i_19 => i_A(5),
                i_20 => i_A(4),
335             i_21 => i_A(3),
                i_22 => i_A(2),
337             i_23 => i_A(1),
                i_24 => i_A(0),
339             i_25 => k_zero,
                i_26 => k_zero,
341             i_27 => k_zero,
```

```
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(24));

    u_myMUX_bit23: myMUX32_1
    port map (i_0 => i_A(23),
            i_1 => i_A(22),
            i_2 => i_A(21),
            i_3 => i_A(20),
            i_4 => i_A(19),
            i_5 => i_A(18),
            i_6 => i_A(17),
            i_7 => i_A(16),
            i_8 => i_A(15),
            i_9 => i_A(14),
            i_10 => i_A(13),
            i_11 => i_A(12),
            i_12 => i_A(11),
            i_13 => i_A(10),
            i_14 => i_A(9),
            i_15 => i_A(8),
            i_16 => i_A(7),
            i_17 => i_A(6),
            i_18 => i_A(5),
            i_19 => i_A(4),
            i_20 => i_A(3),
            i_21 => i_A(2),
            i_22 => i_A(1),
            i_23 => i_A(0),
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(23));
```

```
385   u_myMUX_bit22: myMUX32_1
      port map (i_0 => i_A(22),
387             i_1 => i_A(21),
                i_2 => i_A(20),
389             i_3 => i_A(19),
                i_4 => i_A(18),
391             i_5 => i_A(17),
                i_6 => i_A(16),
393             i_7 => i_A(15),
                i_8 => i_A(14),
395             i_9 => i_A(13),
                i_10 => i_A(12),
397             i_11 => i_A(11),
                i_12 => i_A(10),
399             i_13 => i_A(9),
                i_14 => i_A(8),
401             i_15 => i_A(7),
                i_16 => i_A(6),
403             i_17 => i_A(5),
                i_18 => i_A(4),
405             i_19 => i_A(3),
                i_20 => i_A(2),
407             i_21 => i_A(1),
                i_22 => i_A(0),
409             i_23 => k_zero,
                i_24 => k_zero,
411             i_25 => k_zero,
                i_26 => k_zero,
413             i_27 => k_zero,
                i_28 => k_zero,
415             i_29 => k_zero,
                i_30 => k_zero,
417             i_31 => k_zero,
                i_S => i_SA,
419             o_Z => o_Z(22));

421   u_myMUX_bit21: myMUX32_1
      port map (i_0 => i_A(21),
423             i_1 => i_A(20),
                i_2 => i_A(19),
425             i_3 => i_A(18),
                i_4 => i_A(17),
427             i_5 => i_A(16),
```

```
            i_6 => i_A(15),
            i_7 => i_A(14),
            i_8 => i_A(13),
            i_9 => i_A(12),
            i_10 => i_A(11),
            i_11 => i_A(10),
            i_12 => i_A(9),
            i_13 => i_A(8),
            i_14 => i_A(7),
            i_15 => i_A(6),
            i_16 => i_A(5),
            i_17 => i_A(4),
            i_18 => i_A(3),
            i_19 => i_A(2),
            i_20 => i_A(1),
            i_21 => i_A(0),
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(21));

    u_myMUX_bit20: myMUX32_1
    port map (i_0 => i_A(20),
            i_1 => i_A(19),
            i_2 => i_A(18),
            i_3 => i_A(17),
            i_4 => i_A(16),
            i_5 => i_A(15),
            i_6 => i_A(14),
            i_7 => i_A(13),
            i_8 => i_A(12),
            i_9 => i_A(11),
            i_10 => i_A(10),
            i_11 => i_A(9),
            i_12 => i_A(8),
```

```
471             i_13 => i_A(7),
                i_14 => i_A(6),
473             i_15 => i_A(5),
                i_16 => i_A(4),
475             i_17 => i_A(3),
                i_18 => i_A(2),
477             i_19 => i_A(1),
                i_20 => i_A(0),
479             i_21 => k_zero,
                i_22 => k_zero,
481             i_23 => k_zero,
                i_24 => k_zero,
483             i_25 => k_zero,
                i_26 => k_zero,
485             i_27 => k_zero,
                i_28 => k_zero,
487             i_29 => k_zero,
                i_30 => k_zero,
489             i_31 => k_zero,
                i_S => i_SA,
491             o_Z => o_Z(20));

493     u_myMUX_bit19: myMUX32_1
        port map (i_0 => i_A(19),
495             i_1 => i_A(18),
                i_2 => i_A(17),
497             i_3 => i_A(16),
                i_4 => i_A(15),
499             i_5 => i_A(14),
                i_6 => i_A(13),
501             i_7 => i_A(12),
                i_8 => i_A(11),
503             i_9 => i_A(10),
                i_10 => i_A(9),
505             i_11 => i_A(8),
                i_12 => i_A(7),
507             i_13 => i_A(6),
                i_14 => i_A(5),
509             i_15 => i_A(4),
                i_16 => i_A(3),
511             i_17 => i_A(2),
                i_18 => i_A(1),
513             i_19 => i_A(0),
```

```
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(19));

    u_myMUX_bit18: myMUX32_1
    port map (i_0 => i_A(18),
            i_1 => i_A(17),
            i_2 => i_A(16),
            i_3 => i_A(15),
            i_4 => i_A(14),
            i_5 => i_A(13),
            i_6 => i_A(12),
            i_7 => i_A(11),
            i_8 => i_A(10),
            i_9 => i_A(9),
            i_10 => i_A(8),
            i_11 => i_A(7),
            i_12 => i_A(6),
            i_13 => i_A(5),
            i_14 => i_A(4),
            i_15 => i_A(3),
            i_16 => i_A(2),
            i_17 => i_A(1),
            i_18 => i_A(0),
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
```

```
557             i_27 => k_zero,
                i_28 => k_zero,
559             i_29 => k_zero,
                i_30 => k_zero,
561             i_31 => k_zero,
                i_S => i_SA,
563             o_Z => o_Z(18));

565     u_myMUX_bit17: myMUX32_1
        port map (i_0 => i_A(17),
567             i_1 => i_A(16),
                i_2 => i_A(15),
569             i_3 => i_A(14),
                i_4 => i_A(13),
571             i_5 => i_A(12),
                i_6 => i_A(11),
573             i_7 => i_A(10),
                i_8 => i_A(9),
575             i_9 => i_A(8),
                i_10 => i_A(7),
577             i_11 => i_A(6),
                i_12 => i_A(5),
579             i_13 => i_A(4),
                i_14 => i_A(3),
581             i_15 => i_A(2),
                i_16 => i_A(1),
583             i_17 => i_A(0),
                i_18 => k_zero,
585             i_19 => k_zero,
                i_20 => k_zero,
587             i_21 => k_zero,
                i_22 => k_zero,
589             i_23 => k_zero,
                i_24 => k_zero,
591             i_25 => k_zero,
                i_26 => k_zero,
593             i_27 => k_zero,
                i_28 => k_zero,
595             i_29 => k_zero,
                i_30 => k_zero,
597             i_31 => k_zero,
                i_S => i_SA,
599             o_Z => o_Z(17));
```

```
601   u_myMUX_bit16: myMUX32_1
      port map (i_0 => i_A(16),
603             i_1 => i_A(15),
                i_2 => i_A(14),
605             i_3 => i_A(13),
                i_4 => i_A(12),
607             i_5 => i_A(11),
                i_6 => i_A(10),
609             i_7 => i_A(9),
                i_8 => i_A(8),
611             i_9 => i_A(7),
                i_10 => i_A(6),
613             i_11 => i_A(5),
                i_12 => i_A(4),
615             i_13 => i_A(3),
                i_14 => i_A(2),
617             i_15 => i_A(1),
                i_16 => i_A(0),
619             i_17 => k_zero,
                i_18 => k_zero,
621             i_19 => k_zero,
                i_20 => k_zero,
623             i_21 => k_zero,
                i_22 => k_zero,
625             i_23 => k_zero,
                i_24 => k_zero,
627             i_25 => k_zero,
                i_26 => k_zero,
629             i_27 => k_zero,
                i_28 => k_zero,
631             i_29 => k_zero,
                i_30 => k_zero,
633             i_31 => k_zero,
                i_S => i_SA,
635             o_Z => o_Z(16));

637   u_myMUX_bit15: myMUX32_1
      port map (i_0 => i_A(15),
639             i_1 => i_A(14),
                i_2 => i_A(13),
641             i_3 => i_A(12),
                i_4 => i_A(11),
```

```
643             i_5 => i_A(10),
                i_6 => i_A(9),
645             i_7 => i_A(8),
                i_8 => i_A(7),
647             i_9 => i_A(6),
                i_10 => i_A(5),
649             i_11 => i_A(4),
                i_12 => i_A(3),
651             i_13 => i_A(2),
                i_14 => i_A(1),
653             i_15 => i_A(0),
                i_16 => k_zero,
655             i_17 => k_zero,
                i_18 => k_zero,
657             i_19 => k_zero,
                i_20 => k_zero,
659             i_21 => k_zero,
                i_22 => k_zero,
661             i_23 => k_zero,
                i_24 => k_zero,
663             i_25 => k_zero,
                i_26 => k_zero,
665             i_27 => k_zero,
                i_28 => k_zero,
667             i_29 => k_zero,
                i_30 => k_zero,
669             i_31 => k_zero,
                i_S => i_SA,
671             o_Z => o_Z(15));

673     u_myMUX_bit14: myMUX32_1
        port map (i_0 => i_A(14),
675             i_1 => i_A(13),
                i_2 => i_A(12),
677             i_3 => i_A(11),
                i_4 => i_A(10),
679             i_5 => i_A(9),
                i_6 => i_A(8),
681             i_7 => i_A(7),
                i_8 => i_A(6),
683             i_9 => i_A(5),
                i_10 => i_A(4),
685             i_11 => i_A(3),
```

```
            i_12 => i_A(2),
            i_13 => i_A(1),
            i_14 => i_A(0),
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(14));

    u_myMUX_bit13: myMUX32_1
    port map (i_0 => i_A(13),
            i_1 => i_A(12),
            i_2 => i_A(11),
            i_3 => i_A(10),
            i_4 => i_A(9),
            i_5 => i_A(8),
            i_6 => i_A(7),
            i_7 => i_A(6),
            i_8 => i_A(5),
            i_9 => i_A(4),
            i_10 => i_A(3),
            i_11 => i_A(2),
            i_12 => i_A(1),
            i_13 => i_A(0),
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
```

```
729             i_19 => k_zero,
                i_20 => k_zero,
731             i_21 => k_zero,
                i_22 => k_zero,
733             i_23 => k_zero,
                i_24 => k_zero,
735             i_25 => k_zero,
                i_26 => k_zero,
737             i_27 => k_zero,
                i_28 => k_zero,
739             i_29 => k_zero,
                i_30 => k_zero,
741             i_31 => k_zero,
                i_S => i_SA,
743             o_Z => o_Z(13));

745     u_myMUX_bit12: myMUX32_1
        port map (i_0 => i_A(12),
747             i_1 => i_A(11),
                i_2 => i_A(10),
749             i_3 => i_A(9),
                i_4 => i_A(8),
751             i_5 => i_A(7),
                i_6 => i_A(6),
753             i_7 => i_A(5),
                i_8 => i_A(4),
755             i_9 => i_A(3),
                i_10 => i_A(2),
757             i_11 => i_A(1),
                i_12 => i_A(0),
759             i_13 => k_zero,
                i_14 => k_zero,
761             i_15 => k_zero,
                i_16 => k_zero,
763             i_17 => k_zero,
                i_18 => k_zero,
765             i_19 => k_zero,
                i_20 => k_zero,
767             i_21 => k_zero,
                i_22 => k_zero,
769             i_23 => k_zero,
                i_24 => k_zero,
771             i_25 => k_zero,
```

```
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(12));

    u_myMUX_bit11: myMUX32_1
    port map (i_0 => i_A(11),
            i_1 => i_A(10),
            i_2 => i_A(9),
            i_3 => i_A(8),
            i_4 => i_A(7),
            i_5 => i_A(6),
            i_6 => i_A(5),
            i_7 => i_A(4),
            i_8 => i_A(3),
            i_9 => i_A(2),
            i_10 => i_A(1),
            i_11 => i_A(0),
            i_12 => k_zero,
            i_13 => k_zero,
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
```

```
815             o_Z => o_Z(11));

817     u_myMUX_bit10: myMUX32_1
        port map (i_0 => i_A(10),
819             i_1 => i_A(9),
                i_2 => i_A(8),
821             i_3 => i_A(7),
                i_4 => i_A(6),
823             i_5 => i_A(5),
                i_6 => i_A(4),
825             i_7 => i_A(3),
                i_8 => i_A(2),
827             i_9 => i_A(1),
                i_10 => i_A(0),
829             i_11 => k_zero,
                i_12 => k_zero,
831             i_13 => k_zero,
                i_14 => k_zero,
833             i_15 => k_zero,
                i_16 => k_zero,
835             i_17 => k_zero,
                i_18 => k_zero,
837             i_19 => k_zero,
                i_20 => k_zero,
839             i_21 => k_zero,
                i_22 => k_zero,
841             i_23 => k_zero,
                i_24 => k_zero,
843             i_25 => k_zero,
                i_26 => k_zero,
845             i_27 => k_zero,
                i_28 => k_zero,
847             i_29 => k_zero,
                i_30 => k_zero,
849             i_31 => k_zero,
                i_S => i_SA,
851             o_Z => o_Z(10));

853     u_myMUX_bit9: myMUX32_1
        port map (i_0 => i_A(9),
855             i_1 => i_A(8),
                i_2 => i_A(7),
857             i_3 => i_A(6),
```

```
            i_4 => i_A(5),
            i_5 => i_A(4),
            i_6 => i_A(3),
            i_7 => i_A(2),
            i_8 => i_A(1),
            i_9 => i_A(0),
            i_10 => k_zero,
            i_11 => k_zero,
            i_12 => k_zero,
            i_13 => k_zero,
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(9));

    u_myMUX_bit8: myMUX32_1
    port map (i_0 => i_A(8),
            i_1 => i_A(7),
            i_2 => i_A(6),
            i_3 => i_A(5),
            i_4 => i_A(4),
            i_5 => i_A(3),
            i_6 => i_A(2),
            i_7 => i_A(1),
            i_8 => i_A(0),
            i_9 => k_zero,
            i_10 => k_zero,
```

```
901             i_11 => k_zero,
                i_12 => k_zero,
903             i_13 => k_zero,
                i_14 => k_zero,
905             i_15 => k_zero,
                i_16 => k_zero,
907             i_17 => k_zero,
                i_18 => k_zero,
909             i_19 => k_zero,
                i_20 => k_zero,
911             i_21 => k_zero,
                i_22 => k_zero,
913             i_23 => k_zero,
                i_24 => k_zero,
915             i_25 => k_zero,
                i_26 => k_zero,
917             i_27 => k_zero,
                i_28 => k_zero,
919             i_29 => k_zero,
                i_30 => k_zero,
921             i_31 => k_zero,
                i_S => i_SA,
923             o_Z => o_Z(8));

925     u_myMUX_bit7: myMUX32_1
        port map (i_0 => i_A(7),
927             i_1 => i_A(6),
                i_2 => i_A(5),
929             i_3 => i_A(4),
                i_4 => i_A(3),
931             i_5 => i_A(2),
                i_6 => i_A(1),
933             i_7 => i_A(0),
                i_8 => k_zero,
935             i_9 => k_zero,
                i_10 => k_zero,
937             i_11 => k_zero,
                i_12 => k_zero,
939             i_13 => k_zero,
                i_14 => k_zero,
941             i_15 => k_zero,
                i_16 => k_zero,
943             i_17 => k_zero,
```

```
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(7));

    u_myMUX_bit6: myMUX32_1
    port map (i_0 => i_A(6),
            i_1 => i_A(5),
            i_2 => i_A(4),
            i_3 => i_A(3),
            i_4 => i_A(2),
            i_5 => i_A(1),
            i_6 => i_A(0),
            i_7 => k_zero,
            i_8 => k_zero,
            i_9 => k_zero,
            i_10 => k_zero,
            i_11 => k_zero,
            i_12 => k_zero,
            i_13 => k_zero,
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
```

```
987             i_25 => k_zero,
                i_26 => k_zero,
989             i_27 => k_zero,
                i_28 => k_zero,
991             i_29 => k_zero,
                i_30 => k_zero,
993             i_31 => k_zero,
                i_S => i_SA,
995             o_Z => o_Z(6));

997     u_myMUX_bit5: myMUX32_1
        port map (i_0 => i_A(5),
999             i_1 => i_A(4),
                i_2 => i_A(3),
1001            i_3 => i_A(2),
                i_4 => i_A(1),
1003            i_5 => i_A(0),
                i_6 => k_zero,
1005            i_7 => k_zero,
                i_8 => k_zero,
1007            i_9 => k_zero,
                i_10 => k_zero,
1009            i_11 => k_zero,
                i_12 => k_zero,
1011            i_13 => k_zero,
                i_14 => k_zero,
1013            i_15 => k_zero,
                i_16 => k_zero,
1015            i_17 => k_zero,
                i_18 => k_zero,
1017            i_19 => k_zero,
                i_20 => k_zero,
1019            i_21 => k_zero,
                i_22 => k_zero,
1021            i_23 => k_zero,
                i_24 => k_zero,
1023            i_25 => k_zero,
                i_26 => k_zero,
1025            i_27 => k_zero,
                i_28 => k_zero,
1027            i_29 => k_zero,
                i_30 => k_zero,
1029            i_31 => k_zero,
```

```
            i_S => i_SA,
1031        o_Z => o_Z(5));

1033    u_myMUX_bit4: myMUX32_1
        port map (i_0 => i_A(4),
1035            i_1 => i_A(3),
            i_2 => i_A(2),
1037        i_3 => i_A(1),
            i_4 => i_A(0),
1039        i_5 => k_zero,
            i_6 => k_zero,
1041        i_7 => k_zero,
            i_8 => k_zero,
1043        i_9 => k_zero,
            i_10 => k_zero,
1045        i_11 => k_zero,
            i_12 => k_zero,
1047        i_13 => k_zero,
            i_14 => k_zero,
1049        i_15 => k_zero,
            i_16 => k_zero,
1051        i_17 => k_zero,
            i_18 => k_zero,
1053        i_19 => k_zero,
            i_20 => k_zero,
1055        i_21 => k_zero,
            i_22 => k_zero,
1057        i_23 => k_zero,
            i_24 => k_zero,
1059        i_25 => k_zero,
            i_26 => k_zero,
1061        i_27 => k_zero,
            i_28 => k_zero,
1063        i_29 => k_zero,
            i_30 => k_zero,
1065        i_31 => k_zero,
            i_S => i_SA,
1067        o_Z => o_Z(4));

1069    u_myMUX_bit3: myMUX32_1
        port map (i_0 => i_A(3),
1071            i_1 => i_A(2),
            i_2 => i_A(1),
```

```
1073            i_3 => i_A(0),
                i_4 => k_zero,
1075            i_5 => k_zero,
                i_6 => k_zero,
1077            i_7 => k_zero,
                i_8 => k_zero,
1079            i_9 => k_zero,
                i_10 => k_zero,
1081            i_11 => k_zero,
                i_12 => k_zero,
1083            i_13 => k_zero,
                i_14 => k_zero,
1085            i_15 => k_zero,
                i_16 => k_zero,
1087            i_17 => k_zero,
                i_18 => k_zero,
1089            i_19 => k_zero,
                i_20 => k_zero,
1091            i_21 => k_zero,
                i_22 => k_zero,
1093            i_23 => k_zero,
                i_24 => k_zero,
1095            i_25 => k_zero,
                i_26 => k_zero,
1097            i_27 => k_zero,
                i_28 => k_zero,
1099            i_29 => k_zero,
                i_30 => k_zero,
1101            i_31 => k_zero,
                i_S => i_SA,
1103            o_Z => o_Z(3));

1105    u_myMUX_bit2: myMUX32_1
        port map (i_0 => i_A(2),
1107            i_1 => i_A(1),
                i_2 => i_A(0),
1109            i_3 => k_zero,
                i_4 => k_zero,
1111            i_5 => k_zero,
                i_6 => k_zero,
1113            i_7 => k_zero,
                i_8 => k_zero,
1115            i_9 => k_zero,
```

```
            i_10 => k_zero,
            i_11 => k_zero,
            i_12 => k_zero,
            i_13 => k_zero,
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
            i_17 => k_zero,
            i_18 => k_zero,
            i_19 => k_zero,
            i_20 => k_zero,
            i_21 => k_zero,
            i_22 => k_zero,
            i_23 => k_zero,
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(2));

    u_myMUX_bit1: myMUX32_1
    port map (i_0 => i_A(1),
            i_1 => i_A(0),
            i_2 => k_zero,
            i_3 => k_zero,
            i_4 => k_zero,
            i_5 => k_zero,
            i_6 => k_zero,
            i_7 => k_zero,
            i_8 => k_zero,
            i_9 => k_zero,
            i_10 => k_zero,
            i_11 => k_zero,
            i_12 => k_zero,
            i_13 => k_zero,
            i_14 => k_zero,
            i_15 => k_zero,
            i_16 => k_zero,
```

```
1159            i_17 => k_zero,
                i_18 => k_zero,
1161            i_19 => k_zero,
                i_20 => k_zero,
1163            i_21 => k_zero,
                i_22 => k_zero,
1165            i_23 => k_zero,
                i_24 => k_zero,
1167            i_25 => k_zero,
                i_26 => k_zero,
1169            i_27 => k_zero,
                i_28 => k_zero,
1171            i_29 => k_zero,
                i_30 => k_zero,
1173            i_31 => k_zero,
                i_S => i_SA,
1175            o_Z => o_Z(1));

1177    u_myMUX_bit0: myMUX32_1
        port map (i_0 => i_A(0),
1179            i_1 => k_zero,
                i_2 => k_zero,
1181            i_3 => k_zero,
                i_4 => k_zero,
1183            i_5 => k_zero,
                i_6 => k_zero,
1185            i_7 => k_zero,
                i_8 => k_zero,
1187            i_9 => k_zero,
                i_10 => k_zero,
1189            i_11 => k_zero,
                i_12 => k_zero,
1191            i_13 => k_zero,
                i_14 => k_zero,
1193            i_15 => k_zero,
                i_16 => k_zero,
1195            i_17 => k_zero,
                i_18 => k_zero,
1197            i_19 => k_zero,
                i_20 => k_zero,
1199            i_21 => k_zero,
                i_22 => k_zero,
1201            i_23 => k_zero,
```

```
            i_24 => k_zero,
            i_25 => k_zero,
            i_26 => k_zero,
            i_27 => k_zero,
            i_28 => k_zero,
            i_29 => k_zero,
            i_30 => k_zero,
            i_31 => k_zero,
            i_S => i_SA,
            o_Z => o_Z(0));
end a_LEFT_SHIFT;
```

Listing A.14. LEFT_SHIFT.vhd Code

1.5 Left Shift 2

```
--| SLL2_32.vhd
--| Shift a 32-bit input left by 2 bits
--|
--| INPUTS:
--| i_A - Input A
--|
--| OUTPUTS:
--| o_Z - Shifted input
library IEEE;
use IEEE.std_logic_1164.all;

entity SLL2_32 is
  port (i_A : in  std_logic_vector(31 downto 0);
        o_Z : out std_logic_vector(31 downto 0));
end SLL2_32;

architecture a_SLL2_32 of SLL2_32 is
begin
  o_Z <= i_A(29 downto 0) & B"00";
end a_SLL2_32;
```

Listing A.15. SLL2_32.vhd Code

1.6 Right Shift

```
--| RIGHT_SHIFT.vhd
--| Implements the right shift function.  Shifts input A by the amount specified by SA.
--| Performs a logical right shift if w_fill is 0 and an arithmetic right shift if w_fill is 1.
--|
--| INPUTS:
--| i_A - Data Input
--| i_SA - Shift Amount
--| i_ARITH - 0 for logical shift, 1 for arithmetic shift
--|
--| OUTPUTS:
--| o_Z - Shifted input data (o_Z = i_A >> i_SA)
library IEEE;
use IEEE.std_logic_1164.all;

entity RIGHT_SHIFT is
  port (i_A    : in  std_logic_vector(31 downto 0);
        i_SA   : in  std_logic_vector(4 downto 0);
        i_ARITH : in std_logic;
        o_Z    : out std_logic_vector(31 downto 0));
end RIGHT_SHIFT;

architecture a_RIGHT_SHIFT of RIGHT_SHIFT is
  -- Declare Components
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  component myMUX32_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
```

```
40          i_7 : in  std_logic;
            i_8 : in  std_logic;
42          i_9 : in  std_logic;
            i_10  : in  std_logic;
44          i_11  : in  std_logic;
            i_12  : in  std_logic;
46          i_13  : in  std_logic;
            i_14  : in  std_logic;
48          i_15  : in  std_logic;
            i_16  : in  std_logic;
50          i_17  : in  std_logic;
            i_18  : in  std_logic;
52          i_19  : in  std_logic;
            i_20  : in  std_logic;
54          i_21  : in  std_logic;
            i_22  : in  std_logic;
56          i_23  : in  std_logic;
            i_24  : in  std_logic;
58          i_25  : in  std_logic;
            i_26  : in  std_logic;
60          i_27  : in  std_logic;
            i_28  : in  std_logic;
62          i_29  : in  std_logic;
            i_30  : in  std_logic;
64          i_31  : in  std_logic;
            i_S : in  std_logic_vector (4 downto 0);
66          o_Z : out std_logic
            );
68    end component;

70    signal w_fill : std_logic;
      constant k_zero : std_logic := '0';
72
  begin
74
      u_mux2_1 : myMUX2_1
76    port map (i_0 => k_zero,
              i_1 => i_A(31),
78            i_S => i_ARITH,
              o_Z => w_fill);
80
      u_myMUX_bit31: myMUX32_1
82    port map (i_0 => i_A(31),
```

```
            i_1 => w_fill,
84          i_2 => w_fill,
            i_3 => w_fill,
86          i_4 => w_fill,
            i_5 => w_fill,
88          i_6 => w_fill,
            i_7 => w_fill,
90          i_8 => w_fill,
            i_9 => w_fill,
92          i_10 => w_fill,
            i_11 => w_fill,
94          i_12 => w_fill,
            i_13 => w_fill,
96          i_14 => w_fill,
            i_15 => w_fill,
98          i_16 => w_fill,
            i_17 => w_fill,
100         i_18 => w_fill,
            i_19 => w_fill,
102         i_20 => w_fill,
            i_21 => w_fill,
104         i_22 => w_fill,
            i_23 => w_fill,
106         i_24 => w_fill,
            i_25 => w_fill,
108         i_26 => w_fill,
            i_27 => w_fill,
110         i_28 => w_fill,
            i_29 => w_fill,
112         i_30 => w_fill,
            i_31 => w_fill,
114         i_S => i_SA,
            o_Z => o_Z(31));
116
    u_myMUX_bit30: myMUX32_1
118 port map (i_0 => i_A(30),
            i_1 => i_A(31),
120         i_2 => w_fill,
            i_3 => w_fill,
122         i_4 => w_fill,
            i_5 => w_fill,
124         i_6 => w_fill,
            i_7 => w_fill,
```

```
126             i_8 => w_fill,
                i_9 => w_fill,
128             i_10 => w_fill,
                i_11 => w_fill,
130             i_12 => w_fill,
                i_13 => w_fill,
132             i_14 => w_fill,
                i_15 => w_fill,
134             i_16 => w_fill,
                i_17 => w_fill,
136             i_18 => w_fill,
                i_19 => w_fill,
138             i_20 => w_fill,
                i_21 => w_fill,
140             i_22 => w_fill,
                i_23 => w_fill,
142             i_24 => w_fill,
                i_25 => w_fill,
144             i_26 => w_fill,
                i_27 => w_fill,
146             i_28 => w_fill,
                i_29 => w_fill,
148             i_30 => w_fill,
                i_31 => w_fill,
150             i_S => i_SA,
                o_Z => o_Z(30));
152
    u_myMUX_bit29: myMUX32_1
154 port map (i_0 => i_A(29),
                i_1 => i_A(30),
156             i_2 => i_A(31),
                i_3 => w_fill,
158             i_4 => w_fill,
                i_5 => w_fill,
160             i_6 => w_fill,
                i_7 => w_fill,
162             i_8 => w_fill,
                i_9 => w_fill,
164             i_10 => w_fill,
                i_11 => w_fill,
166             i_12 => w_fill,
                i_13 => w_fill,
168             i_14 => w_fill,
```

```
            i_15 => w_fill,
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(29));

    u_myMUX_bit28: myMUX32_1
    port map (i_0 => i_A(28),
            i_1 => i_A(29),
            i_2 => i_A(30),
            i_3 => i_A(31),
            i_4 => w_fill,
            i_5 => w_fill,
            i_6 => w_fill,
            i_7 => w_fill,
            i_8 => w_fill,
            i_9 => w_fill,
            i_10 => w_fill,
            i_11 => w_fill,
            i_12 => w_fill,
            i_13 => w_fill,
            i_14 => w_fill,
            i_15 => w_fill,
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
```

```
212             i_22 => w_fill,
                i_23 => w_fill,
214             i_24 => w_fill,
                i_25 => w_fill,
216             i_26 => w_fill,
                i_27 => w_fill,
218             i_28 => w_fill,
                i_29 => w_fill,
220             i_30 => w_fill,
                i_31 => w_fill,
222             i_S => i_SA,
                o_Z => o_Z(28));
224
    u_myMUX_bit27: myMUX32_1
226 port map (i_0 => i_A(27),
                i_1 => i_A(28),
228             i_2 => i_A(29),
                i_3 => i_A(30),
230             i_4 => i_A(31),
                i_5 => w_fill,
232             i_6 => w_fill,
                i_7 => w_fill,
234             i_8 => w_fill,
                i_9 => w_fill,
236             i_10 => w_fill,
                i_11 => w_fill,
238             i_12 => w_fill,
                i_13 => w_fill,
240             i_14 => w_fill,
                i_15 => w_fill,
242             i_16 => w_fill,
                i_17 => w_fill,
244             i_18 => w_fill,
                i_19 => w_fill,
246             i_20 => w_fill,
                i_21 => w_fill,
248             i_22 => w_fill,
                i_23 => w_fill,
250             i_24 => w_fill,
                i_25 => w_fill,
252             i_26 => w_fill,
                i_27 => w_fill,
254             i_28 => w_fill,
```

```
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(27));

    u_myMUX_bit26: myMUX32_1
    port map (i_0 => i_A(26),
            i_1 => i_A(27),
            i_2 => i_A(28),
            i_3 => i_A(29),
            i_4 => i_A(30),
            i_5 => i_A(31),
            i_6 => w_fill,
            i_7 => w_fill,
            i_8 => w_fill,
            i_9 => w_fill,
            i_10 => w_fill,
            i_11 => w_fill,
            i_12 => w_fill,
            i_13 => w_fill,
            i_14 => w_fill,
            i_15 => w_fill,
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(26));

    u_myMUX_bit25: myMUX32_1
```

```
298     port map (i_0 => i_A(25),
                  i_1 => i_A(26),
300               i_2 => i_A(27),
                  i_3 => i_A(28),
302               i_4 => i_A(29),
                  i_5 => i_A(30),
304               i_6 => i_A(31),
                  i_7 => w_fill,
306               i_8 => w_fill,
                  i_9 => w_fill,
308               i_10 => w_fill,
                  i_11 => w_fill,
310               i_12 => w_fill,
                  i_13 => w_fill,
312               i_14 => w_fill,
                  i_15 => w_fill,
314               i_16 => w_fill,
                  i_17 => w_fill,
316               i_18 => w_fill,
                  i_19 => w_fill,
318               i_20 => w_fill,
                  i_21 => w_fill,
320               i_22 => w_fill,
                  i_23 => w_fill,
322               i_24 => w_fill,
                  i_25 => w_fill,
324               i_26 => w_fill,
                  i_27 => w_fill,
326               i_28 => w_fill,
                  i_29 => w_fill,
328               i_30 => w_fill,
                  i_31 => w_fill,
330               i_S => i_SA,
                  o_Z => o_Z(25));
332
    u_myMUX_bit24: myMUX32_1
334 port map (i_0 => i_A(24),
                  i_1 => i_A(25),
336               i_2 => i_A(26),
                  i_3 => i_A(27),
338               i_4 => i_A(28),
                  i_5 => i_A(29),
340               i_6 => i_A(30),
```

```
            i_7 => i_A(31),
342         i_8 => w_fill,
            i_9 => w_fill,
344         i_10 => w_fill,
            i_11 => w_fill,
346         i_12 => w_fill,
            i_13 => w_fill,
348         i_14 => w_fill,
            i_15 => w_fill,
350         i_16 => w_fill,
            i_17 => w_fill,
352         i_18 => w_fill,
            i_19 => w_fill,
354         i_20 => w_fill,
            i_21 => w_fill,
356         i_22 => w_fill,
            i_23 => w_fill,
358         i_24 => w_fill,
            i_25 => w_fill,
360         i_26 => w_fill,
            i_27 => w_fill,
362         i_28 => w_fill,
            i_29 => w_fill,
364         i_30 => w_fill,
            i_31 => w_fill,
366         i_S => i_SA,
            o_Z => o_Z(24));
368
    u_myMUX_bit23: myMUX32_1
370 port map (i_0 => i_A(23),
            i_1 => i_A(24),
372         i_2 => i_A(25),
            i_3 => i_A(26),
374         i_4 => i_A(27),
            i_5 => i_A(28),
376         i_6 => i_A(29),
            i_7 => i_A(30),
378         i_8 => i_A(31),
            i_9 => w_fill,
380         i_10 => w_fill,
            i_11 => w_fill,
382         i_12 => w_fill,
            i_13 => w_fill,
```

```
384             i_14 => w_fill,
                i_15 => w_fill,
386             i_16 => w_fill,
                i_17 => w_fill,
388             i_18 => w_fill,
                i_19 => w_fill,
390             i_20 => w_fill,
                i_21 => w_fill,
392             i_22 => w_fill,
                i_23 => w_fill,
394             i_24 => w_fill,
                i_25 => w_fill,
396             i_26 => w_fill,
                i_27 => w_fill,
398             i_28 => w_fill,
                i_29 => w_fill,
400             i_30 => w_fill,
                i_31 => w_fill,
402             i_S => i_SA,
                o_Z => o_Z(23));
404
    u_myMUX_bit22: myMUX32_1
406 port map (i_0 => i_A(22),
                i_1 => i_A(23),
408             i_2 => i_A(24),
                i_3 => i_A(25),
410             i_4 => i_A(26),
                i_5 => i_A(27),
412             i_6 => i_A(28),
                i_7 => i_A(29),
414             i_8 => i_A(30),
                i_9 => i_A(31),
416             i_10 => w_fill,
                i_11 => w_fill,
418             i_12 => w_fill,
                i_13 => w_fill,
420             i_14 => w_fill,
                i_15 => w_fill,
422             i_16 => w_fill,
                i_17 => w_fill,
424             i_18 => w_fill,
                i_19 => w_fill,
426             i_20 => w_fill,
```

```
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(22));

    u_myMUX_bit21: myMUX32_1
    port map (i_0 => i_A(21),
            i_1 => i_A(22),
            i_2 => i_A(23),
            i_3 => i_A(24),
            i_4 => i_A(25),
            i_5 => i_A(26),
            i_6 => i_A(27),
            i_7 => i_A(28),
            i_8 => i_A(29),
            i_9 => i_A(30),
            i_10 => i_A(31),
            i_11 => w_fill,
            i_12 => w_fill,
            i_13 => w_fill,
            i_14 => w_fill,
            i_15 => w_fill,
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
```

```
470             i_28 => w_fill,
                i_29 => w_fill,
472             i_30 => w_fill,
                i_31 => w_fill,
474             i_S => i_SA,
                o_Z => o_Z(21));
476
    u_myMUX_bit20: myMUX32_1
478 port map (i_0 => i_A(20),
                i_1 => i_A(21),
480             i_2 => i_A(22),
                i_3 => i_A(23),
482             i_4 => i_A(24),
                i_5 => i_A(25),
484             i_6 => i_A(26),
                i_7 => i_A(27),
486             i_8 => i_A(28),
                i_9 => i_A(29),
488             i_10 => i_A(30),
                i_11 => i_A(31),
490             i_12 => w_fill,
                i_13 => w_fill,
492             i_14 => w_fill,
                i_15 => w_fill,
494             i_16 => w_fill,
                i_17 => w_fill,
496             i_18 => w_fill,
                i_19 => w_fill,
498             i_20 => w_fill,
                i_21 => w_fill,
500             i_22 => w_fill,
                i_23 => w_fill,
502             i_24 => w_fill,
                i_25 => w_fill,
504             i_26 => w_fill,
                i_27 => w_fill,
506             i_28 => w_fill,
                i_29 => w_fill,
508             i_30 => w_fill,
                i_31 => w_fill,
510             i_S => i_SA,
                o_Z => o_Z(20));
512
```

```
    u_myMUX_bit19: myMUX32_1
514 port map (i_0 => i_A(19),
              i_1 => i_A(20),
516           i_2 => i_A(21),
              i_3 => i_A(22),
518           i_4 => i_A(23),
              i_5 => i_A(24),
520           i_6 => i_A(25),
              i_7 => i_A(26),
522           i_8 => i_A(27),
              i_9 => i_A(28),
524           i_10 => i_A(29),
              i_11 => i_A(30),
526           i_12 => i_A(31),
              i_13 => w_fill,
528           i_14 => w_fill,
              i_15 => w_fill,
530           i_16 => w_fill,
              i_17 => w_fill,
532           i_18 => w_fill,
              i_19 => w_fill,
534           i_20 => w_fill,
              i_21 => w_fill,
536           i_22 => w_fill,
              i_23 => w_fill,
538           i_24 => w_fill,
              i_25 => w_fill,
540           i_26 => w_fill,
              i_27 => w_fill,
542           i_28 => w_fill,
              i_29 => w_fill,
544           i_30 => w_fill,
              i_31 => w_fill,
546           i_S => i_SA,
              o_Z => o_Z(19));
548
    u_myMUX_bit18: myMUX32_1
550 port map (i_0 => i_A(18),
              i_1 => i_A(19),
552           i_2 => i_A(20),
              i_3 => i_A(21),
554           i_4 => i_A(22),
              i_5 => i_A(23),
```

```
556             i_6 => i_A(24),
                i_7 => i_A(25),
558             i_8 => i_A(26),
                i_9 => i_A(27),
560             i_10 => i_A(28),
                i_11 => i_A(29),
562             i_12 => i_A(30),
                i_13 => i_A(31),
564             i_14 => w_fill,
                i_15 => w_fill,
566             i_16 => w_fill,
                i_17 => w_fill,
568             i_18 => w_fill,
                i_19 => w_fill,
570             i_20 => w_fill,
                i_21 => w_fill,
572             i_22 => w_fill,
                i_23 => w_fill,
574             i_24 => w_fill,
                i_25 => w_fill,
576             i_26 => w_fill,
                i_27 => w_fill,
578             i_28 => w_fill,
                i_29 => w_fill,
580             i_30 => w_fill,
                i_31 => w_fill,
582             i_S => i_SA,
                o_Z => o_Z(18));
584
    u_myMUX_bit17: myMUX32_1
586 port map (i_0 => i_A(17),
                i_1 => i_A(18),
588             i_2 => i_A(19),
                i_3 => i_A(20),
590             i_4 => i_A(21),
                i_5 => i_A(22),
592             i_6 => i_A(23),
                i_7 => i_A(24),
594             i_8 => i_A(25),
                i_9 => i_A(26),
596             i_10 => i_A(27),
                i_11 => i_A(28),
598             i_12 => i_A(29),
```

```
            i_13 => i_A(30),
            i_14 => i_A(31),
            i_15 => w_fill,
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(17));

    u_myMUX_bit16: myMUX32_1
    port map (i_0 => i_A(16),
            i_1 => i_A(17),
            i_2 => i_A(18),
            i_3 => i_A(19),
            i_4 => i_A(20),
            i_5 => i_A(21),
            i_6 => i_A(22),
            i_7 => i_A(23),
            i_8 => i_A(24),
            i_9 => i_A(25),
            i_10 => i_A(26),
            i_11 => i_A(27),
            i_12 => i_A(28),
            i_13 => i_A(29),
            i_14 => i_A(30),
            i_15 => i_A(31),
            i_16 => w_fill,
            i_17 => w_fill,
            i_18 => w_fill,
            i_19 => w_fill,
```

```
642             i_20 => w_fill,
                i_21 => w_fill,
644             i_22 => w_fill,
                i_23 => w_fill,
646             i_24 => w_fill,
                i_25 => w_fill,
648             i_26 => w_fill,
                i_27 => w_fill,
650             i_28 => w_fill,
                i_29 => w_fill,
652             i_30 => w_fill,
                i_31 => w_fill,
654             i_S => i_SA,
                o_Z => o_Z(16));
656
    u_myMUX_bit15: myMUX32_1
658 port map (i_0 => i_A(15),
                i_1 => i_A(16),
660             i_2 => i_A(17),
                i_3 => i_A(18),
662             i_4 => i_A(19),
                i_5 => i_A(20),
664             i_6 => i_A(21),
                i_7 => i_A(22),
666             i_8 => i_A(23),
                i_9 => i_A(24),
668             i_10 => i_A(25),
                i_11 => i_A(26),
670             i_12 => i_A(27),
                i_13 => i_A(28),
672             i_14 => i_A(29),
                i_15 => i_A(30),
674             i_16 => i_A(31),
                i_17 => w_fill,
676             i_18 => w_fill,
                i_19 => w_fill,
678             i_20 => w_fill,
                i_21 => w_fill,
680             i_22 => w_fill,
                i_23 => w_fill,
682             i_24 => w_fill,
                i_25 => w_fill,
684             i_26 => w_fill,
```

```
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(15));

    u_myMUX_bit14: myMUX32_1
    port map (i_0 => i_A(14),
            i_1 => i_A(15),
            i_2 => i_A(16),
            i_3 => i_A(17),
            i_4 => i_A(18),
            i_5 => i_A(19),
            i_6 => i_A(20),
            i_7 => i_A(21),
            i_8 => i_A(22),
            i_9 => i_A(23),
            i_10 => i_A(24),
            i_11 => i_A(25),
            i_12 => i_A(26),
            i_13 => i_A(27),
            i_14 => i_A(28),
            i_15 => i_A(29),
            i_16 => i_A(30),
            i_17 => i_A(31),
            i_18 => w_fill,
            i_19 => w_fill,
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(14));
```

```
728
    u_myMUX_bit13: myMUX32_1
730 port map (i_0 => i_A(13),
              i_1 => i_A(14),
732           i_2 => i_A(15),
              i_3 => i_A(16),
734           i_4 => i_A(17),
              i_5 => i_A(18),
736           i_6 => i_A(19),
              i_7 => i_A(20),
738           i_8 => i_A(21),
              i_9 => i_A(22),
740           i_10 => i_A(23),
              i_11 => i_A(24),
742           i_12 => i_A(25),
              i_13 => i_A(26),
744           i_14 => i_A(27),
              i_15 => i_A(28),
746           i_16 => i_A(29),
              i_17 => i_A(30),
748           i_18 => i_A(31),
              i_19 => w_fill,
750           i_20 => w_fill,
              i_21 => w_fill,
752           i_22 => w_fill,
              i_23 => w_fill,
754           i_24 => w_fill,
              i_25 => w_fill,
756           i_26 => w_fill,
              i_27 => w_fill,
758           i_28 => w_fill,
              i_29 => w_fill,
760           i_30 => w_fill,
              i_31 => w_fill,
762           i_S => i_SA,
              o_Z => o_Z(13));
764
    u_myMUX_bit12: myMUX32_1
766 port map (i_0 => i_A(12),
              i_1 => i_A(13),
768           i_2 => i_A(14),
              i_3 => i_A(15),
770           i_4 => i_A(16),
```

```
            i_5 => i_A(17),
            i_6 => i_A(18),
            i_7 => i_A(19),
            i_8 => i_A(20),
            i_9 => i_A(21),
            i_10 => i_A(22),
            i_11 => i_A(23),
            i_12 => i_A(24),
            i_13 => i_A(25),
            i_14 => i_A(26),
            i_15 => i_A(27),
            i_16 => i_A(28),
            i_17 => i_A(29),
            i_18 => i_A(30),
            i_19 => i_A(31),
            i_20 => w_fill,
            i_21 => w_fill,
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(12));

    u_myMUX_bit11: myMUX32_1
    port map (i_0 => i_A(11),
            i_1 => i_A(12),
            i_2 => i_A(13),
            i_3 => i_A(14),
            i_4 => i_A(15),
            i_5 => i_A(16),
            i_6 => i_A(17),
            i_7 => i_A(18),
            i_8 => i_A(19),
            i_9 => i_A(20),
            i_10 => i_A(21),
            i_11 => i_A(22),
```

```
814             i_12 => i_A(23),
                i_13 => i_A(24),
816             i_14 => i_A(25),
                i_15 => i_A(26),
818             i_16 => i_A(27),
                i_17 => i_A(28),
820             i_18 => i_A(29),
                i_19 => i_A(30),
822             i_20 => i_A(31),
                i_21 => w_fill,
824             i_22 => w_fill,
                i_23 => w_fill,
826             i_24 => w_fill,
                i_25 => w_fill,
828             i_26 => w_fill,
                i_27 => w_fill,
830             i_28 => w_fill,
                i_29 => w_fill,
832             i_30 => w_fill,
                i_31 => w_fill,
834             i_S => i_SA,
                o_Z => o_Z(11));
836
    u_myMUX_bit10: myMUX32_1
838 port map (i_0 => i_A(10),
                i_1 => i_A(11),
840             i_2 => i_A(12),
                i_3 => i_A(13),
842             i_4 => i_A(14),
                i_5 => i_A(15),
844             i_6 => i_A(16),
                i_7 => i_A(17),
846             i_8 => i_A(18),
                i_9 => i_A(19),
848             i_10 => i_A(20),
                i_11 => i_A(21),
850             i_12 => i_A(22),
                i_13 => i_A(23),
852             i_14 => i_A(24),
                i_15 => i_A(25),
854             i_16 => i_A(26),
                i_17 => i_A(27),
856             i_18 => i_A(28),
```

```
            i_19 => i_A(29),
            i_20 => i_A(30),
            i_21 => i_A(31),
            i_22 => w_fill,
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
            i_26 => w_fill,
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(10));

    u_myMUX_bit9: myMUX32_1
    port map (i_0 => i_A(9),
            i_1 => i_A(10),
            i_2 => i_A(11),
            i_3 => i_A(12),
            i_4 => i_A(13),
            i_5 => i_A(14),
            i_6 => i_A(15),
            i_7 => i_A(16),
            i_8 => i_A(17),
            i_9 => i_A(18),
            i_10 => i_A(19),
            i_11 => i_A(20),
            i_12 => i_A(21),
            i_13 => i_A(22),
            i_14 => i_A(23),
            i_15 => i_A(24),
            i_16 => i_A(25),
            i_17 => i_A(26),
            i_18 => i_A(27),
            i_19 => i_A(28),
            i_20 => i_A(29),
            i_21 => i_A(30),
            i_22 => i_A(31),
            i_23 => w_fill,
            i_24 => w_fill,
            i_25 => w_fill,
```

```
900             i_26 => w_fill,
                i_27 => w_fill,
902             i_28 => w_fill,
                i_29 => w_fill,
904             i_30 => w_fill,
                i_31 => w_fill,
906             i_S => i_SA,
                o_Z => o_Z(9));
908
    u_myMUX_bit8: myMUX32_1
910 port map (i_0 => i_A(8),
                i_1 => i_A(9),
912             i_2 => i_A(10),
                i_3 => i_A(11),
914             i_4 => i_A(12),
                i_5 => i_A(13),
916             i_6 => i_A(14),
                i_7 => i_A(15),
918             i_8 => i_A(16),
                i_9 => i_A(17),
920             i_10 => i_A(18),
                i_11 => i_A(19),
922             i_12 => i_A(20),
                i_13 => i_A(21),
924             i_14 => i_A(22),
                i_15 => i_A(23),
926             i_16 => i_A(24),
                i_17 => i_A(25),
928             i_18 => i_A(26),
                i_19 => i_A(27),
930             i_20 => i_A(28),
                i_21 => i_A(29),
932             i_22 => i_A(30),
                i_23 => i_A(31),
934             i_24 => w_fill,
                i_25 => w_fill,
936             i_26 => w_fill,
                i_27 => w_fill,
938             i_28 => w_fill,
                i_29 => w_fill,
940             i_30 => w_fill,
                i_31 => w_fill,
942             i_S => i_SA,
```

```
        o_Z => o_Z(8));

u_myMUX_bit7: myMUX32_1
port map (i_0 => i_A(7),
        i_1 => i_A(8),
        i_2 => i_A(9),
        i_3 => i_A(10),
        i_4 => i_A(11),
        i_5 => i_A(12),
        i_6 => i_A(13),
        i_7 => i_A(14),
        i_8 => i_A(15),
        i_9 => i_A(16),
        i_10 => i_A(17),
        i_11 => i_A(18),
        i_12 => i_A(19),
        i_13 => i_A(20),
        i_14 => i_A(21),
        i_15 => i_A(22),
        i_16 => i_A(23),
        i_17 => i_A(24),
        i_18 => i_A(25),
        i_19 => i_A(26),
        i_20 => i_A(27),
        i_21 => i_A(28),
        i_22 => i_A(29),
        i_23 => i_A(30),
        i_24 => i_A(31),
        i_25 => w_fill,
        i_26 => w_fill,
        i_27 => w_fill,
        i_28 => w_fill,
        i_29 => w_fill,
        i_30 => w_fill,
        i_31 => w_fill,
        i_S => i_SA,
        o_Z => o_Z(7));

u_myMUX_bit6: myMUX32_1
port map (i_0 => i_A(6),
        i_1 => i_A(7),
        i_2 => i_A(8),
        i_3 => i_A(9),
```

```
986             i_4 => i_A(10),
                i_5 => i_A(11),
988             i_6 => i_A(12),
                i_7 => i_A(13),
990             i_8 => i_A(14),
                i_9 => i_A(15),
992             i_10 => i_A(16),
                i_11 => i_A(17),
994             i_12 => i_A(18),
                i_13 => i_A(19),
996             i_14 => i_A(20),
                i_15 => i_A(21),
998             i_16 => i_A(22),
                i_17 => i_A(23),
1000            i_18 => i_A(24),
                i_19 => i_A(25),
1002            i_20 => i_A(26),
                i_21 => i_A(27),
1004            i_22 => i_A(28),
                i_23 => i_A(29),
1006            i_24 => i_A(30),
                i_25 => i_A(31),
1008            i_26 => w_fill,
                i_27 => w_fill,
1010            i_28 => w_fill,
                i_29 => w_fill,
1012            i_30 => w_fill,
                i_31 => w_fill,
1014            i_S => i_SA,
                o_Z => o_Z(6));
1016
    u_myMUX_bit5: myMUX32_1
1018    port map (i_0 => i_A(5),
                i_1 => i_A(6),
1020            i_2 => i_A(7),
                i_3 => i_A(8),
1022            i_4 => i_A(9),
                i_5 => i_A(10),
1024            i_6 => i_A(11),
                i_7 => i_A(12),
1026            i_8 => i_A(13),
                i_9 => i_A(14),
1028            i_10 => i_A(15),
```

```
            i_11 => i_A(16),
            i_12 => i_A(17),
            i_13 => i_A(18),
            i_14 => i_A(19),
            i_15 => i_A(20),
            i_16 => i_A(21),
            i_17 => i_A(22),
            i_18 => i_A(23),
            i_19 => i_A(24),
            i_20 => i_A(25),
            i_21 => i_A(26),
            i_22 => i_A(27),
            i_23 => i_A(28),
            i_24 => i_A(29),
            i_25 => i_A(30),
            i_26 => i_A(31),
            i_27 => w_fill,
            i_28 => w_fill,
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(5));

    u_myMUX_bit4: myMUX32_1
    port map (i_0 => i_A(4),
            i_1 => i_A(5),
            i_2 => i_A(6),
            i_3 => i_A(7),
            i_4 => i_A(8),
            i_5 => i_A(9),
            i_6 => i_A(10),
            i_7 => i_A(11),
            i_8 => i_A(12),
            i_9 => i_A(13),
            i_10 => i_A(14),
            i_11 => i_A(15),
            i_12 => i_A(16),
            i_13 => i_A(17),
            i_14 => i_A(18),
            i_15 => i_A(19),
            i_16 => i_A(20),
            i_17 => i_A(21),
```

```
1072            i_18 => i_A(22),
                i_19 => i_A(23),
1074            i_20 => i_A(24),
                i_21 => i_A(25),
1076            i_22 => i_A(26),
                i_23 => i_A(27),
1078            i_24 => i_A(28),
                i_25 => i_A(29),
1080            i_26 => i_A(30),
                i_27 => i_A(31),
1082            i_28 => w_fill,
                i_29 => w_fill,
1084            i_30 => w_fill,
                i_31 => w_fill,
1086            i_S => i_SA,
                o_Z => o_Z(4));
1088
    u_myMUX_bit3: myMUX32_1
1090    port map (i_0 => i_A(3),
                i_1 => i_A(4),
1092            i_2 => i_A(5),
                i_3 => i_A(6),
1094            i_4 => i_A(7),
                i_5 => i_A(8),
1096            i_6 => i_A(9),
                i_7 => i_A(10),
1098            i_8 => i_A(11),
                i_9 => i_A(12),
1100            i_10 => i_A(13),
                i_11 => i_A(14),
1102            i_12 => i_A(15),
                i_13 => i_A(16),
1104            i_14 => i_A(17),
                i_15 => i_A(18),
1106            i_16 => i_A(19),
                i_17 => i_A(20),
1108            i_18 => i_A(21),
                i_19 => i_A(22),
1110            i_20 => i_A(23),
                i_21 => i_A(24),
1112            i_22 => i_A(25),
                i_23 => i_A(26),
1114            i_24 => i_A(27),
```

```
            i_25 => i_A(28),
            i_26 => i_A(29),
            i_27 => i_A(30),
            i_28 => i_A(31),
            i_29 => w_fill,
            i_30 => w_fill,
            i_31 => w_fill,
            i_S => i_SA,
            o_Z => o_Z(3));

    u_myMUX_bit2: myMUX32_1
    port map (i_0 => i_A(2),
            i_1 => i_A(3),
            i_2 => i_A(4),
            i_3 => i_A(5),
            i_4 => i_A(6),
            i_5 => i_A(7),
            i_6 => i_A(8),
            i_7 => i_A(9),
            i_8 => i_A(10),
            i_9 => i_A(11),
            i_10 => i_A(12),
            i_11 => i_A(13),
            i_12 => i_A(14),
            i_13 => i_A(15),
            i_14 => i_A(16),
            i_15 => i_A(17),
            i_16 => i_A(18),
            i_17 => i_A(19),
            i_18 => i_A(20),
            i_19 => i_A(21),
            i_20 => i_A(22),
            i_21 => i_A(23),
            i_22 => i_A(24),
            i_23 => i_A(25),
            i_24 => i_A(26),
            i_25 => i_A(27),
            i_26 => i_A(28),
            i_27 => i_A(29),
            i_28 => i_A(30),
            i_29 => i_A(31),
            i_30 => w_fill,
            i_31 => w_fill,
```

```
1158            i_S => i_SA,
                o_Z => o_Z(2));
1160
    u_myMUX_bit1: myMUX32_1
1162    port map (i_0 => i_A(1),
                i_1 => i_A(2),
1164            i_2 => i_A(3),
                i_3 => i_A(4),
1166            i_4 => i_A(5),
                i_5 => i_A(6),
1168            i_6 => i_A(7),
                i_7 => i_A(8),
1170            i_8 => i_A(9),
                i_9 => i_A(10),
1172            i_10 => i_A(11),
                i_11 => i_A(12),
1174            i_12 => i_A(13),
                i_13 => i_A(14),
1176            i_14 => i_A(15),
                i_15 => i_A(16),
1178            i_16 => i_A(17),
                i_17 => i_A(18),
1180            i_18 => i_A(19),
                i_19 => i_A(20),
1182            i_20 => i_A(21),
                i_21 => i_A(22),
1184            i_22 => i_A(23),
                i_23 => i_A(24),
1186            i_24 => i_A(25),
                i_25 => i_A(26),
1188            i_26 => i_A(27),
                i_27 => i_A(28),
1190            i_28 => i_A(29),
                i_29 => i_A(30),
1192            i_30 => i_A(31),
                i_31 => w_fill,
1194            i_S => i_SA,
                o_Z => o_Z(1));
1196
    u_myMUX_bit0: myMUX32_1
1198    port map (i_0 => i_A(0),
                i_1 => i_A(1),
1200            i_2 => i_A(2),
```

```
            i_3 => i_A(3),
            i_4 => i_A(4),
            i_5 => i_A(5),
            i_6 => i_A(6),
            i_7 => i_A(7),
            i_8 => i_A(8),
            i_9 => i_A(9),
            i_10 => i_A(10),
            i_11 => i_A(11),
            i_12 => i_A(12),
            i_13 => i_A(13),
            i_14 => i_A(14),
            i_15 => i_A(15),
            i_16 => i_A(16),
            i_17 => i_A(17),
            i_18 => i_A(18),
            i_19 => i_A(19),
            i_20 => i_A(20),
            i_21 => i_A(21),
            i_22 => i_A(22),
            i_23 => i_A(23),
            i_24 => i_A(24),
            i_25 => i_A(25),
            i_26 => i_A(26),
            i_27 => i_A(27),
            i_28 => i_A(28),
            i_29 => i_A(29),
            i_30 => i_A(30),
            i_31 => i_A(31),
            i_S => i_SA,
            o_Z => o_Z(0));
end a_RIGHT_SHIFT;
```

Listing A.16. RIGHT_SHIFT.vhd Code

1.7 Bitwise Operations

1.7.1 2-bit Bitwise Operations

```
--| Bitwise_Ops2
--| Performs the AND, OR, XOR, and NOR operations on inputs A
    and B
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--|
--| OUTPUTS:
--| o_A_n - Inverted Input A
--| o_B_n - Inverted Input B
--| o_AND - AND(A,B)
--| o_OR  - OR(A,B)
--| o_XOR - XOR(A,B)
--| o_XNOR - XNOR(A,B) - Inverted XOR output
--| o_NOR - NOR(A,B)
library IEEE;
use IEEE.std_logic_1164.all;

entity Bitwise_Ops2 is
  port (i_A : in  std_logic;
        i_B : in  std_logic;
        o_A_n : out std_logic;
        o_B_n : out std_logic;
        o_AND : out std_logic;
        o_NAND: out std_logic;
        o_OR  : out std_logic;
        o_XOR : out std_logic;
        o_XNOR  : out std_logic;
        o_NOR : out std_logic);
end Bitwise_Ops2;

architecture a_Bitwise_Ops2 of Bitwise_Ops2 is
--| Declare components
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;
```

```
40
   component myINV is
42     port (i_A : in  std_logic;
           o_Z : out std_logic);
44   end component;
--| Declare signals
46   signal w_A_n    : std_logic; -- INV(A)
   signal w_B_n    : std_logic; -- INV(B)
48   signal w_AB_n   : std_logic; -- INV(AB) = NAND(A,B)
   signal w_A_B    : std_logic; -- OR(A,B)
50   signal w_xorAB_n  : std_logic; -- INV(XOR(A,B))
 begin
52   o_A_n <= w_A_n;
   o_B_n <= w_B_n;
54   o_OR <= w_A_B;
   o_NAND <= w_AB_n;
56   o_XNOR <= w_xorAB_n;
   --| INV(A)
58   u_myINV_A: myINV
   port map (i_A => i_A,
60           o_Z => w_A_n);
   --| INV(B)
62   u_myINV_B: myINV
   port map (i_A => i_B,
64           o_Z => w_B_n);
   --| NAND(A,B)
66   u_myNAND2_AB_n: myNAND2
   port map (i_A => i_A,
68           i_B => i_B,
           o_Z => w_AB_n);
70   --| INV(NAND(A,B)) = AND(A,B)
   u_myINV_AB: myINV
72   port map (i_A => w_AB_n,
           o_Z => o_AND);
74   --| NAND(INV(A),INV(B)) = OR(A,B)
   u_myNAND2_A_B: myNAND2
76   port map (i_A => w_A_n,
           i_B => w_B_n,
78           o_Z => w_A_B);
   --| INV(NAND(INV(A),INV(B))) = NOR(A,B)
80   u_myINV_A_B_n: myINV
   port map (i_A => w_A_B,
82           o_Z => o_NOR);
```

```
  --| NAND(NAND(A,B),NOR(A,B)) = INV(XOR(A,B))
84  u_myNAND2_XORAB_n: myNAND2
    port map (i_A => w_AB_n,
86            i_B => w_A_B,
              o_Z => w_xorAB_n);
88  --| XOR(A,B)
    u_myINV_XORAB: myINV
90  port map (i_A => w_xorAB_n,
              o_Z => o_XOR);
92 end a_Bitwise_Ops2;
```

Listing A.17. Bitwise_Ops2.vhd Code

1.7.2 2-bit Bitwise Operations

```
--| Bitwise_Ops2_32
--| Performs the AND, OR, XOR, and NOR operations on inputs A and B
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--|
--| OUTPUTS:
--| o_A_n - Inverted Input A
--| o_B_n - Inverted Input B
--| o_AND - AND(A,B)
--| o_OR  - OR(A,B)
--| o_XOR - XOR(A,B)
--| o_XNOR - XNOR(A,B) - Inverted XOR output
--| o_NOR - NOR(A,B)
library IEEE;
use IEEE.std_logic_1164.all;

entity Bitwise_Ops2_32 is
  port (i_A    : in  std_logic_vector(31 downto 0);
        i_B    : in  std_logic_vector(31 downto 0);
        o_A_n  : out std_logic_vector(31 downto 0);
        o_B_n  : out std_logic_vector(31 downto 0);
        o_AND  : out std_logic_vector(31 downto 0);
        o_NAND : out std_logic_vector(31 downto 0);
        o_OR   : out std_logic_vector(31 downto 0);
        o_XOR  : out std_logic_vector(31 downto 0);
        o_XNOR : out std_logic_vector(31 downto 0);
        o_NOR  : out std_logic_vector(31 downto 0));
end Bitwise_Ops2_32;

architecture a_Bitwise_Ops2_32 of Bitwise_Ops2_32 is
--| Declare components
  component Bitwise_Ops2 is
    port (i_A    : in  std_logic;
          i_B    : in  std_logic;
          o_A_n  : out std_logic;
          o_B_n  : out std_logic;
          o_AND  : out std_logic;
          o_NAND : out std_logic;
```

```
        o_OR     : out std_logic;
        o_XOR    : out std_logic;
        o_XNOR   : out std_logic;
        o_NOR    : out std_logic);
  end component;
begin
  bit_wise_array: for bit_index in 0 to 31 generate
    u_bit_ops : Bitwise_Ops2
    port map (i_A => i_A(bit_index),
              i_B => i_B(bit_index),
              o_A_n => o_A_n(bit_index),
              o_B_n => o_B_n(bit_index),
              o_AND => o_AND(bit_index),
              o_NAND => o_NAND(bit_index),
              o_OR => o_OR(bit_index),
              o_XOR => o_XOR(bit_index),
              o_XNOR => o_XNOR(bit_index),
              o_NOR => o_NOR(bit_index));
  end generate bit_wise_array;
end a_Bitwise_Ops2_32;
```

Listing A.18. Bitwise_Ops2_32.vhd Code

1.8 Adder

1.8.1 1-bit Full Adder

```
--| FULL_ADDER_1.vhd
--| Single bit full adder
--| Add two 1-bit numbers along with a carry in bit.  Produce
--| a 1-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A      - Input A
--| i_A_n    - Inverted input A
--| i_B      - Input B
--| i_B_n    - Inverted input B
--| i_AB_AND  - AND(A,B)
--| i_AB_NAND - NAND(A,B)
--| i_AB_NOR  - NOR(A,B)
--| i_C      - Carry In
--|
--| OUTPUTS:
--| o_S      - i_A+i_B+i_C (sum)
--| o_C      - 1 if (i_A)(i_B) OR (i_A)(i_C) OR (i_B)(i_C) and
    0 otherwise
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_1 is
  port (i_A      : in  std_logic;
        i_A_n    : in  std_logic;
        i_B      : in  std_logic;
        i_B_n    : in  std_logic;
        i_AB_AND    : in  std_logic;
        i_AB_NAND : in  std_logic;
        i_AB_NOR    : in  std_logic;
        i_C      : in  std_logic;
        o_S      : out std_logic;
        o_C      : out std_logic);
end FULL_ADDER_1;

architecture a_FULL_ADDER_1 of FULL_ADDER_1 is
--| Declare components
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
```

```
40          o_Z : out std_logic
            );
42    end component;

44    component myINV is
        port (i_A : in  std_logic;
46          o_Z : out std_logic);
      end component;
48  --| Declare signals

50    -- Inverted Inputs
      signal w_C_n : std_logic;
52
      -- Intermediate Signals
54    signal w_add_01 : std_logic;
      signal w_add_02 : std_logic;
56    signal w_add_03, w_add_03_n : std_logic;
      signal w_add_05, w_add_05_n : std_logic;
58    signal w_add_06, w_add_06_n : std_logic;
      signal w_add_07 : std_logic;
60    signal w_add_08 : std_logic;
      signal w_add_09 : std_logic;
62    signal w_add_10 : std_logic;
      signal w_add_11, w_add_11_n : std_logic;
64    signal w_add_12, w_add_12_n : std_logic;
  begin
66
      --| INV(i_C)
68    u_myINV_C: myINV
      port map (i_A => i_C,
70            o_Z => w_C_n);
      --| Create Inverted Intermediate Signals
72    u_myINV_03: myINV
      port map (i_A => w_add_03,
74            o_Z => w_add_03_n);

76    u_myINV_05: myINV
      port map (i_A => w_add_05,
78            o_Z => w_add_05_n);

80    u_myINV_06: myINV
      port map (i_A => w_add_06,
82            o_Z => w_add_06_n);
```

```
84  u_myINV_11: myINV
    port map (i_A => w_add_11,
86          o_Z => w_add_11_n);

88  u_myINV_12: myINV
    port map (i_A => w_add_12,
90          o_Z => w_add_12_n);

92  u_myNAND2_01: myNAND2
    port map (i_A => i_A,
94          i_B => i_C,
            o_Z => w_add_01);
96
    u_myNAND2_02: myNAND2
98  port map (i_A => i_B,
            i_B => i_C,
100         o_Z => w_add_02);

102 u_myNAND2_03: myNAND2
    port map (i_A => i_AB_NAND,
104         i_B => w_add_01,
            o_Z => w_add_03);
106
    u_myNAND2_05: myNAND2
108 port map (i_A => i_A_n,
            i_B => i_B,
110         o_Z => w_add_05);

112 u_myNAND2_06: myNAND2
    port map (i_A => i_A,
114         i_B => i_B_n,
            o_Z => w_add_06);
116
    u_myNAND2_07: myNAND2
118 port map (i_A => i_AB_NOR,
            i_B => i_C,
120         o_Z => w_add_07);

122 u_myNAND2_08: myNAND2
    port map (i_A => w_add_05_n,
124         i_B => w_C_n,
            o_Z => w_add_08);
```

```
126
    u_myNAND2_09: myNAND2
128 port map (i_A => w_add_06_n,
              i_B => w_C_n,
130           o_Z => w_add_09);

132 u_myNAND2_10: myNAND2
    port map (i_A => i_AB_AND,
134           i_B => i_C,
              o_Z => w_add_10);
136
    u_myNAND2_11: myNAND2
138 port map (i_A => w_add_07,
              i_B => w_add_08,
140           o_Z => w_add_11);

142 u_myNAND2_12: myNAND2
    port map (i_A => w_add_09,
144           i_B => w_add_10,
              o_Z => w_add_12);
146
    --| Generate Outputs
148 u_myNAND2_S: myNAND2
    port map (i_A => w_add_11_n,
150           i_B => w_add_12_n,
              o_Z => o_S);
152
    u_myNAND2_CO: myNAND2
154 port map (i_A => w_add_03_n,
              i_B => w_add_02,
156           o_Z => o_C);

158 end a_FULL_ADDER_1;
```

Listing A.19. FULL_ADDER_1.vhd Code

1.8.2 4-bit Full Adder

```
--| FULL_ADDER_4.vhd
--| 4-bit full adder
--| Add two 4-bit numbers along with a carry in bit. Produce
--| a 4-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_A_n - Inverted input A
--| i_B_n - Inverted input B
--| i_AB_AND - AND(A,B)
--| i_AB_NAND - NOT(AND(A,B))
--| i_AB_NOR - NOT(OR(A,B))
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_4 is
  port (i_A      : in  std_logic_vector(3 downto 0);
        i_A_n    : in  std_logic_vector(3 downto 0);
        i_B      : in  std_logic_vector(3 downto 0);
        i_B_n    : in  std_logic_vector(3 downto 0);
        i_AB_AND    : in  std_logic_vector(3 downto 0);
        i_AB_NAND : in  std_logic_vector(3 downto 0);
        i_AB_NOR    : in  std_logic_vector(3 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(3 downto 0);
        o_C      : out std_logic);
end FULL_ADDER_4;

architecture a_FULL_ADDER_4 of FULL_ADDER_4 is
--| Declare components
  component FULL_ADDER_1 is
    port (i_A      : in  std_logic;
          i_A_n    : in  std_logic;
          i_B      : in  std_logic;
          i_B_n    : in  std_logic;
```

```
42          i_AB_AND    : in  std_logic;
            i_AB_NAND : in  std_logic;
44          i_AB_NOR    : in  std_logic;
            i_C     : in  std_logic;
46          o_S     : out std_logic;
            o_C     : out std_logic);
48    end component;
  --| Declare signals
50    signal w_C1 : std_logic; -- Carry out from bit 0 addition
       to bit 1 addition
      signal w_C2 : std_logic; -- Carry out from bit 1 addition
       to bit 2 addition
52    signal w_C3 : std_logic; -- Carry out from bit 3 addition
       to bit 4 addition

54  begin
    -- Bit 0 Full Adder
56    u_FULL_ADD_0: FULL_ADDER_1
      port map (i_A => i_A(0),
58            i_A_n => i_A_n(0),
              i_B => i_B(0),
60            i_B_n => i_B_n(0),
              i_AB_AND => i_AB_AND(0),
62            i_AB_NAND => i_AB_NAND(0),
              i_AB_NOR => i_AB_NOR(0),
64            i_C => i_C,
              o_S => o_S(0),
66            o_C => w_C1);

68    -- Bit 1 Full Adder
      u_FULL_ADD_1: FULL_ADDER_1
70    port map (i_A => i_A(1),
              i_A_n => i_A_n(1),
72            i_B => i_B(1),
              i_B_n => i_B_n(1),
74            i_AB_AND => i_AB_AND(1),
              i_AB_NAND => i_AB_NAND(1),
76            i_AB_NOR => i_AB_NOR(1),
              i_C => w_C1,
78            o_S => o_S(1),
              o_C => w_C2);
80
    -- Bit 2 Full Adder
```

```
82    u_FULL_ADD_2: FULL_ADDER_1
      port map (i_A => i_A(2),
84            i_A_n => i_A_n(2),
              i_B => i_B(2),
86            i_B_n => i_B_n(2),
              i_AB_AND => i_AB_AND(2),
88            i_AB_NAND => i_AB_NAND(2),
              i_AB_NOR => i_AB_NOR(2),
90            i_C => w_C2,
              o_S => o_S(2),
92            o_C => w_C3);

94    -- Bit 3 Full Adder
      u_FULL_ADD_3: FULL_ADDER_1
96    port map (i_A => i_A(3),
              i_A_n => i_A_n(3),
98            i_B => i_B(3),
              i_B_n => i_B_n(3),
100           i_AB_AND => i_AB_AND(3),
              i_AB_NAND => i_AB_NAND(3),
102           i_AB_NOR => i_AB_NOR(3),
              i_C => w_C3,
104           o_S => o_S(3),
              o_C => o_C);
106
  end a_FULL_ADDER_4;
```

Listing A.20. FULL_ADDER_4.vhd Code

1.8.3 4-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_4
--| Add two 4-bit numbers along with a carry in bit.  Produce
--| a 4-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_A_n - Inverted input A
--| i_B_n - Inverted input B
--| i_AB_AND - AND(A,B)
--| i_AB_NAND - NOT(AND(A,B))
--| i_AB_NOR - NOT(OR(A,B))
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_4 is
  port (i_A      : in  std_logic_vector(3 downto 0);
        i_A_n    : in  std_logic_vector(3 downto 0);
        i_B      : in  std_logic_vector(3 downto 0);
        i_B_n    : in  std_logic_vector(3 downto 0);
        i_AB_AND    : in  std_logic_vector(3 downto 0);
        i_AB_NAND : in  std_logic_vector(3 downto 0);
        i_AB_NOR    : in  std_logic_vector(3 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(3 downto 0);
        o_C      : out std_logic);
end CARRY_SELECT_ADDER_4;

architecture a_CARRY_SELECT_ADDER_4 of CARRY_SELECT_ADDER_4
   is
--| Declare components
  component FULL_ADDER_4 is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_A_n    : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_B_n    : in  std_logic_vector(3 downto 0);
```

```
41          i_AB_AND    : in  std_logic_vector(3 downto 0);
            i_AB_NAND : in  std_logic_vector(3 downto 0);
43          i_AB_NOR    : in  std_logic_vector(3 downto 0);
            i_C     : in  std_logic;
45          o_S     : out std_logic_vector(3 downto 0);
            o_C     : out std_logic);
47    end component;

49    component myMUX2_N is
        generic (m_width : integer := 4);
51      port (i_0 : in  std_logic_vector(3 downto 0);
              i_1 : in  std_logic_vector(3 downto 0);
53            i_S : in  std_logic;
              o_Z : out std_logic_vector(3 downto 0)
55            );
    end component;
57
    component myNAND2 is
59    port (i_A : in  std_logic;
              i_B : in  std_logic;
61            o_Z : out std_logic
              );
63  end component;

65  component myINV is
      port (i_A : in  std_logic;
67          o_Z : out std_logic
            );
69  end component;
--| Declare Constants
71  constant k_zero : std_logic := '0';
    constant k_one : std_logic := '1';
73
--| Declare signals
75  signal w_S0 : std_logic_vector(3 downto 0); -- Sum from
     FULL_ADDER with 0 carry in
    signal w_S1 : std_logic_vector(3 downto 0); -- Sum from
     FULL_ADDER with 1 carry in
77  signal w_C0, w_C0_n : std_logic; -- Carry out from
     FULL_ADDER with 0 carry in
    signal w_C1 : std_logic; -- Carry out from FULL_ADDER with
     1 carry in
79  signal w_C11 : std_logic; -- NAND(C1,i_C)
```

```
81 begin
   -- Connect FULL_ADDER_4
83   u_FULL_ADDER_0: FULL_ADDER_4
     port map (i_A => i_A,
85             i_A_n => i_A_n,
               i_B => i_B,
87             i_B_n => i_B_n,
               i_AB_AND => i_AB_AND,
89             i_AB_NAND => i_AB_NAND,
               i_AB_NOR => i_AB_NOR,
91             i_C => k_zero,
               o_S => w_S0,
93             o_C => w_C0);

95   u_FULL_ADDER_1: FULL_ADDER_4
     port map (i_A => i_A,
97             i_A_n => i_A_n,
               i_B => i_B,
99             i_B_n => i_B_n,
               i_AB_AND => i_AB_AND,
101            i_AB_NAND => i_AB_NAND,
               i_AB_NOR => i_AB_NOR,
103            i_C => k_one,
               o_S => w_S1,
105            o_C => w_C1);

107  u_myMUX2_4_S: myMUX2_N
     generic map (m_width => 4)
109  port map (i_0 => w_S0,
               i_1 => w_S1,
111            i_S => i_C,
               o_Z => o_S);
113
     u_myINV_C0 : myINV
115  port map (
       i_A => w_C0,
117    o_Z => w_C0_n);

119  u_myNAND_C11 : myNAND2
     port map (
121    i_A => w_C1,
       i_B => i_C,
```

```
123       o_Z => w_C11);

125     u_myNAND_Cout : myNAND2
        port map (
127       i_A => w_C11,
          i_B => w_C0_n,
129       o_Z => o_C);
    end a_CARRY_SELECT_ADDER_4;
```

Listing A.21. CARRY_SELECT_ADDER_4.vhd Code

1.8.4 32-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_32
--| Add two 32-bit numbers along with a carry in bit.
   Produce
--| a 32-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_A_n - Inverted input A
--| i_B_n - Inverted input B
--| i_AB_AND - AND(A,B)
--| i_AB_NAND - NOT(AND(A,B))
--| i_AB_NOR - NOT(OR(A,B))
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_32 is
  port (i_A      : in  std_logic_vector(31 downto 0);
        i_A_n    : in  std_logic_vector(31 downto 0);
        i_B      : in  std_logic_vector(31 downto 0);
        i_B_n    : in  std_logic_vector(31 downto 0);
        i_AB_AND    : in  std_logic_vector(31 downto 0);
        i_AB_NAND : in  std_logic_vector(31 downto 0);
        i_AB_NOR    : in  std_logic_vector(31 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(31 downto 0);
        o_C      : out std_logic);
end CARRY_SELECT_ADDER_32;

architecture a_CARRY_SELECT_ADDER_32 of CARRY_SELECT_ADDER_32
    is
--| Declare components
  component CARRY_SELECT_ADDER_4 is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_A_n    : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
```

```
40          i_B_n      : in  std_logic_vector(3 downto 0);
            i_AB_AND   : in  std_logic_vector(3 downto 0);
42          i_AB_NAND  : in  std_logic_vector(3 downto 0);
            i_AB_NOR   : in  std_logic_vector(3 downto 0);
44          i_C        : in  std_logic;
            o_S        : out std_logic_vector(3 downto 0);
46          o_C        : out std_logic);
    end component;
48
--| Declare signals
50    signal w_C1 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 0 to CARRY_SELECT_ADDER 1
      signal w_C2 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 1 to CARRY_SELECT_ADDER 2
52    signal w_C3 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 2 to CARRY_SELECT_ADDER 3
      signal w_C4 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 3 to CARRY_SELECT_ADDER 4
54    signal w_C5 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 4 to CARRY_SELECT_ADDER 5
      signal w_C6 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 5 to CARRY_SELECT_ADDER 6
56    signal w_C7 : std_logic; -- Carry out from
      CARRY_SELECT_ADDER 6 to CARRY_SELECT_ADDER 7
  begin
58    -- Connect CARRY_SELECT_ADDERs
    u_CARRY_SELECT_ADDER_0: CARRY_SELECT_ADDER_4
60    port map (i_A => i_A(3 downto 0),
              i_A_n => i_A_n(3 downto 0),
62            i_B => i_B(3 downto 0),
              i_B_n => i_B_n(3 downto 0),
64            i_AB_AND => i_AB_AND(3 downto 0),
              i_AB_NAND => i_AB_NAND(3 downto 0),
66            i_AB_NOR => i_AB_NOR(3 downto 0),
              i_C => i_C,
68            o_S => o_S(3 downto 0),
              o_C => w_C1);
70
    u_CARRY_SELECT_ADDER_1: CARRY_SELECT_ADDER_4
72    port map (i_A => i_A(7 downto 4),
              i_A_n => i_A_n(7 downto 4),
74            i_B => i_B(7 downto 4),
              i_B_n => i_B_n(7 downto 4),
```

```
76          i_AB_AND => i_AB_AND(7 downto 4),
            i_AB_NAND => i_AB_NAND(7 downto 4),
78          i_AB_NOR => i_AB_NOR(7 downto 4),
            i_C => w_C1,
80          o_S => o_S(7 downto 4),
            o_C => w_C2);
82
    u_CARRY_SELECT_ADDER_2: CARRY_SELECT_ADDER_4
84  port map (i_A => i_A(11 downto 8),
            i_A_n => i_A_n(11 downto 8),
86          i_B => i_B(11 downto 8),
            i_B_n => i_B_n(11 downto 8),
88          i_AB_AND => i_AB_AND(11 downto 8),
            i_AB_NAND => i_AB_NAND(11 downto 8),
90          i_AB_NOR => i_AB_NOR(11 downto 8),
            i_C => w_C2,
92          o_S => o_S(11 downto 8),
            o_C => w_C3);
94
    u_CARRY_SELECT_ADDER_3: CARRY_SELECT_ADDER_4
96  port map (i_A => i_A(15 downto 12),
            i_A_n => i_A_n(15 downto 12),
98          i_B => i_B(15 downto 12),
            i_B_n => i_B_n(15 downto 12),
100         i_AB_AND => i_AB_AND(15 downto 12),
            i_AB_NAND => i_AB_NAND(15 downto 12),
102         i_AB_NOR => i_AB_NOR(15 downto 12),
            i_C => w_C3,
104         o_S => o_S(15 downto 12),
            o_C => w_C4);
106
    u_CARRY_SELECT_ADDER_4: CARRY_SELECT_ADDER_4
108 port map (i_A => i_A(19 downto 16),
            i_A_n => i_A_n(19 downto 16),
110         i_B => i_B(19 downto 16),
            i_B_n => i_B_n(19 downto 16),
112         i_AB_AND => i_AB_AND(19 downto 16),
            i_AB_NAND => i_AB_NAND(19 downto 16),
114         i_AB_NOR => i_AB_NOR(19 downto 16),
            i_C => w_C4,
116         o_S => o_S(19 downto 16),
            o_C => w_C5);
118
```

```
    u_CARRY_SELECT_ADDER_5: CARRY_SELECT_ADDER_4
120  port map (i_A => i_A(23 downto 20),
              i_A_n => i_A_n(23 downto 20),
122           i_B => i_B(23 downto 20),
              i_B_n => i_B_n(23 downto 20),
124           i_AB_AND => i_AB_AND(23 downto 20),
              i_AB_NAND => i_AB_NAND(23 downto 20),
126           i_AB_NOR => i_AB_NOR(23 downto 20),
              i_C => w_C5,
128           o_S => o_S(23 downto 20),
              o_C => w_C6);
130
    u_CARRY_SELECT_ADDER_6: CARRY_SELECT_ADDER_4
132  port map (i_A => i_A(27 downto 24),
              i_A_n => i_A_n(27 downto 24),
134           i_B => i_B(27 downto 24),
              i_B_n => i_B_n(27 downto 24),
136           i_AB_AND => i_AB_AND(27 downto 24),
              i_AB_NAND => i_AB_NAND(27 downto 24),
138           i_AB_NOR => i_AB_NOR(27 downto 24),
              i_C => w_C6,
140           o_S => o_S(27 downto 24),
              o_C => w_C7);
142
    u_CARRY_SELECT_ADDER_7: CARRY_SELECT_ADDER_4
144  port map (i_A => i_A(31 downto 28),
              i_A_n => i_A_n(31 downto 28),
146           i_B => i_B(31 downto 28),
              i_B_n => i_B_n(31 downto 28),
148           i_AB_AND => i_AB_AND(31 downto 28),
              i_AB_NAND => i_AB_NAND(31 downto 28),
150           i_AB_NOR => i_AB_NOR(31 downto 28),
              i_C => w_C7,
152           o_S => o_S(31 downto 28),
              o_C => o_C);
154 end a_CARRY_SELECT_ADDER_32;
```

Listing A.22. CARRY_SELECT_ADDER_32.vhd Code

1.8.5 Self-Contained 1-bit Full Adder

```
--| FULL_ADDER_1_SC.vhd
--| Single bit full adder that is self contained.  Does not receive any
--| intermediate signals to compute the sum from external sources and
--| does not supply any intermediate signals to external modules for
--| use in computing other values.  Unlike the FULL_ADDER_1.vhd.
--| Add two 1-bit numbers along with a carry in bit.  Produce
--| a 1-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - 1 if (i_A)(i_B) OR (i_A)(i_C) OR (i_B)(i_C) and 0 otherwise
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_1_SC is
  port (i_A      : in  std_logic;
        i_B      : in  std_logic;
        i_C      : in  std_logic;
        o_S      : out std_logic;
        o_C      : out std_logic);
end FULL_ADDER_1_SC;

architecture a_FULL_ADDER_1_SC of FULL_ADDER_1_SC is
--| Declare components
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;

```

```
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic);
  end component;
--| Declare signals

  -- Inverted Inputs
  signal w_A_n : std_logic;
  signal w_B_n : std_logic;
  signal w_C_n : std_logic;

  -- Intermediate Signals
  signal w_add_01 : std_logic;
  signal w_add_02 : std_logic;
  signal w_add_03, w_add_03_n : std_logic;
  signal w_add_05, w_add_05_n : std_logic;
  signal w_add_06, w_add_06_n : std_logic;
  signal w_add_07 : std_logic;
  signal w_add_08 : std_logic;
  signal w_add_09 : std_logic;
  signal w_add_10 : std_logic;
  signal w_add_11, w_add_11_n : std_logic;
  signal w_add_12, w_add_12_n : std_logic;

  signal w_AB_NAND  : std_logic;
  signal w_AB_AND : std_logic;
  signal w_AB_OR    : std_logic;
  signal w_AB_NOR : std_logic;
begin
  --| INV(i_A)
  u_myINV_A: myINV
  port map (i_A => i_A,
            o_Z => w_A_n);

  --| INV(i_B)
  u_myINV_B: myINV
  port map (i_A => i_B,
            o_Z => w_B_n);

  --| INV(i_C)
  u_myINV_C: myINV
  port map (i_A => i_C,
            o_Z => w_C_n);
```

```
80
   --| Create Inverted Intermediate Signals
82 u_myINV_03: myINV
   port map (i_A => w_add_03,
84         o_Z => w_add_03_n);

86 u_myINV_05: myINV
   port map (i_A => w_add_05,
88         o_Z => w_add_05_n);

90 u_myINV_06: myINV
   port map (i_A => w_add_06,
92         o_Z => w_add_06_n);

94 u_myINV_11: myINV
   port map (i_A => w_add_11,
96         o_Z => w_add_11_n);

98 u_myINV_12: myINV
   port map (i_A => w_add_12,
100        o_Z => w_add_12_n);

102 u_myINV_AB_AND: myINV
   port map (i_A => w_AB_NAND,
104        o_Z => w_AB_AND);

106 u_myINV_AB_NOR: myINV
   port map (i_A => w_AB_OR,
108        o_Z => w_AB_NOR);

110 --| Create Intermediate Signals
   u_myNAND2_01: myNAND2
112 port map (i_A => i_A,
           i_B => i_C,
114        o_Z => w_add_01);

116 u_myNAND2_02: myNAND2
   port map (i_A => i_B,
118        i_B => i_C,
           o_Z => w_add_02);
120
   u_myNAND2_03: myNAND2
122 port map (i_A => w_AB_NAND,
```

```
        i_B => w_add_01,
        o_Z => w_add_03);

u_myNAND2_05: myNAND2
port map (i_A => w_A_n,
        i_B => i_B,
        o_Z => w_add_05);

u_myNAND2_06: myNAND2
port map (i_A => i_A,
        i_B => w_B_n,
        o_Z => w_add_06);

u_myNAND2_07: myNAND2
port map (i_A => w_AB_NOR,
        i_B => i_C,
        o_Z => w_add_07);

u_myNAND2_08: myNAND2
port map (i_A => w_add_05_n,
        i_B => w_C_n,
        o_Z => w_add_08);

u_myNAND2_09: myNAND2
port map (i_A => w_add_06_n,
        i_B => w_C_n,
        o_Z => w_add_09);

u_myNAND2_10: myNAND2
port map (i_A => w_AB_AND,
        i_B => i_C,
        o_Z => w_add_10);

u_myNAND2_11: myNAND2
port map (i_A => w_add_07,
        i_B => w_add_08,
        o_Z => w_add_11);

u_myNAND2_12: myNAND2
port map (i_A => w_add_09,
        i_B => w_add_10,
        o_Z => w_add_12);
```

```
166   u_myNAND2_AB_NAND: myNAND2
    port map (i_A => i_A,
168           i_B => i_B,
              o_Z => w_AB_NAND);
170
    u_myNAND2_AB_OR: myNAND2
172   port map (i_A => w_A_n,
              i_B => w_B_n,
174           o_Z => w_AB_OR);

176   --| Generate Outputs
    u_myNAND2_S: myNAND2
178   port map (i_A => w_add_11_n,
              i_B => w_add_12_n,
180           o_Z => o_S);

182   u_myNAND2_CO: myNAND2
    port map (i_A => w_add_03_n,
184           i_B => w_add_02,
              o_Z => o_C);
186
  end a_FULL_ADDER_1_SC;
```

Listing A.23. FULL_ADDER_1_SC.vhd Code

1.8.6 Self-Contained 2-bit Full Adder

```
--| FULL_ADDER_2_SC.vhd
--| Self contained 2-bit full adder
--| Add two 2-bit numbers along with a carry in bit.  Produce
--| a 2-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_2_SC is
  port (i_A      : in  std_logic_vector(1 downto 0);
        i_B      : in  std_logic_vector(1 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(1 downto 0);
        o_C      : out std_logic);
end FULL_ADDER_2_SC;

architecture a_FULL_ADDER_2_SC of FULL_ADDER_2_SC is
--| Declare components
  component FULL_ADDER_1_SC is
    port (i_A      : in  std_logic;
          i_B      : in  std_logic;
          i_C      : in  std_logic;
          o_S      : out std_logic;
          o_C      : out std_logic);
  end component;
--| Declare signals
  signal w_C1 : std_logic; -- Carry out from bit 0 addition
   to bit 1 addition
  signal w_C2 : std_logic; -- Carry out from bit 1 addition
   to bit 2 addition
  signal w_C3 : std_logic; -- Carry out from bit 3 addition
   to bit 4 addition
```

```
39 begin
   -- Bit 0 Full Adder
41  u_FULL_ADD_0: FULL_ADDER_1_SC
   port map (i_A => i_A(0),
43           i_B => i_B(0),
           i_C => i_C,
45           o_S => o_S(0),
           o_C => w_C1);
47
   -- Bit 1 Full Adder
49  u_FULL_ADD_1: FULL_ADDER_1_SC
   port map (i_A => i_A(1),
51           i_B => i_B(1),
           i_C => w_C1,
53           o_S => o_S(1),
           o_C => o_C);
55 end a_FULL_ADDER_2_SC;
```

Listing A.24. FULL_ADDER_2_SC.vhd Code

1.8.7 Self-Contained 4-bit Full Adder

```
--| FULL_ADDER_4_SC.vhd
--| Self Contained 4-bit full adder
--| Add two 4-bit numbers along with a carry in bit. Produce
--| a 4-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_4_SC is
  port (i_A      : in  std_logic_vector(3 downto 0);
        i_B      : in  std_logic_vector(3 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(3 downto 0);
        o_C      : out std_logic);
end FULL_ADDER_4_SC;

architecture a_FULL_ADDER_4_SC of FULL_ADDER_4_SC is
--| Declare components
  component FULL_ADDER_1_SC is
    port (i_A      : in  std_logic;
          i_B      : in  std_logic;
          i_C      : in  std_logic;
          o_S      : out std_logic;
          o_C      : out std_logic);
  end component;
--| Declare signals
  signal w_C1 : std_logic; -- Carry out from bit 0 addition
   to bit 1 addition
  signal w_C2 : std_logic; -- Carry out from bit 1 addition
   to bit 2 addition
  signal w_C3 : std_logic; -- Carry out from bit 3 addition
   to bit 4 addition
```

```
39 begin
   -- Bit 0 Full Adder
41 u_FULL_ADD_0: FULL_ADDER_1_SC
   port map (i_A => i_A(0),
43           i_B => i_B(0),
             i_C => i_C,
45           o_S => o_S(0),
             o_C => w_C1);
47
   -- Bit 1 Full Adder
49 u_FULL_ADD_1: FULL_ADDER_1_SC
   port map (i_A => i_A(1),
51           i_B => i_B(1),
             i_C => w_C1,
53           o_S => o_S(1),
             o_C => w_C2);
55
   -- Bit 2 Full Adder
57 u_FULL_ADD_2: FULL_ADDER_1_SC
   port map (i_A => i_A(2),
59           i_B => i_B(2),
             i_C => w_C2,
61           o_S => o_S(2),
             o_C => w_C3);
63
   -- Bit 3 Full Adder
65 u_FULL_ADD_3: FULL_ADDER_1_SC
   port map (i_A => i_A(3),
67           i_B => i_B(3),
             i_C => w_C3,
69           o_S => o_S(3),
             o_C => o_C);
71
   end a_FULL_ADDER_4_SC;
```

Listing A.25. FULL_ADDER_4_SC.vhd Code

1.8.8 Self-Contained 2-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_2_SC
--| Self Contained 2-bit version of the 4-bit carry select adder
--| Add two 2-bit numbers along with a carry in bit. Produce
--| a 2-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_2_SC is
  port (i_A      : in  std_logic_vector(1 downto 0);
        i_B      : in  std_logic_vector(1 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(1 downto 0);
        o_C      : out std_logic);
end CARRY_SELECT_ADDER_2_SC;

architecture a_CARRY_SELECT_ADDER_2_SC of CARRY_SELECT_ADDER_2_SC is
--| Declare components
  component FULL_ADDER_2_SC is
    port (i_A      : in  std_logic_vector(1 downto 0);
          i_B      : in  std_logic_vector(1 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(1 downto 0);
          o_C      : out std_logic);
  end component;

  component myMUX2_N is
    generic (m_width : integer := 4);
    port (i_0 : in  std_logic_vector(1 downto 0);
          i_1 : in  std_logic_vector(1 downto 0);
          i_S : in  std_logic;
```

```
40          o_Z : out std_logic_vector(1 downto 0)
            );
42    end component;

44    component myNAND2 is
        port (i_A : in  std_logic;
46          i_B : in  std_logic;
            o_Z : out std_logic
48          );
    end component;
50
    component myINV is
52    port (i_A : in  std_logic;
            o_Z : out std_logic
54          );
    end component;
56--| Declare Constants
    constant k_zero : std_logic := '0';
58  constant k_one : std_logic := '1';

60--| Declare signals
    signal w_S0 : std_logic_vector(1 downto 0); -- Sum from
     FULL_ADDER with 0 carry in
62  signal w_S1 : std_logic_vector(1 downto 0); -- Sum from
     FULL_ADDER with 1 carry in
    signal w_C0, w_C0_n : std_logic; -- Carry out from
     FULL_ADDER with 0 carry in
64  signal w_C1 : std_logic; -- Carry out from FULL_ADDER with
     1 carry in
    signal w_C11 : std_logic; -- NAND(C1,i_C)
66
  begin
68  -- Connect FULL_ADDER_2_SC
    u_FULL_ADDER_0: FULL_ADDER_2_SC
70  port map (i_A => i_A,
            i_B => i_B,
72          i_C => k_zero,
            o_S => w_S0,
74          o_C => w_C0);

76  u_FULL_ADDER_1: FULL_ADDER_2_SC
    port map (i_A => i_A,
78          i_B => i_B,
```

```
            i_C => k_one,
            o_S => w_S1,
            o_C => w_C1);

  u_myMUX2_2_S: myMUX2_N
  generic map (m_width => 2)
  port map (i_0 => w_S0,
            i_1 => w_S1,
            i_S => i_C,
            o_Z => o_S);

  u_myINV_C0 : myINV
  port map (
    i_A => w_C0,
    o_Z => w_C0_n);

  u_myNAND_C11 : myNAND2
  port map (
    i_A => w_C1,
    i_B => i_C,
    o_Z => w_C11);

  u_myNAND_Cout : myNAND2
  port map (
    i_A => w_C11,
    i_B => w_C0_n,
    o_Z => o_C);
end a_CARRY_SELECT_ADDER_2_SC;
```

Listing A.26. CARRY_SELECT_ADDER_2_SC.vhd Code

1.8.9 Self-Contained 4-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_4_SC
--| Self Contained version of CARRY_SELECT_ADDER_4_SC
--| Add two 4-bit numbers along with a carry in bit. Produce
--| a 4-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_4_SC is
  port (i_A      : in  std_logic_vector(3 downto 0);
        i_B      : in  std_logic_vector(3 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(3 downto 0);
        o_C      : out std_logic);
end CARRY_SELECT_ADDER_4_SC;

architecture a_CARRY_SELECT_ADDER_4_SC of
   CARRY_SELECT_ADDER_4_SC is
--| Declare components
  component FULL_ADDER_4_SC is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;

  component myMUX2_N is
    generic (m_width : integer := 4);
    port (i_0 : in  std_logic_vector(3 downto 0);
          i_1 : in  std_logic_vector(3 downto 0);
          i_S : in  std_logic;
          o_Z : out std_logic_vector(3 downto 0)
```

```
        );
  end component;

  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic
          );
  end component;
--| Declare Constants
  constant k_zero : std_logic := '0';
  constant k_one : std_logic := '1';

--| Declare signals
  signal w_S0 : std_logic_vector(3 downto 0); -- Sum from
   FULL_ADDER with 0 carry in
  signal w_S1 : std_logic_vector(3 downto 0); -- Sum from
   FULL_ADDER with 1 carry in
  signal w_C0, w_C0_n : std_logic; -- Carry out from
   FULL_ADDER with 0 carry in
  signal w_C1 : std_logic; -- Carry out from FULL_ADDER with
   1 carry in
  signal w_C11 : std_logic; -- NAND(C1,i_C)

begin
  -- Connect FULL_ADDER_4_SC
  u_FULL_ADDER_0: FULL_ADDER_4_SC
  port map (i_A => i_A,
            i_B => i_B,
            i_C => k_zero,
            o_S => w_S0,
            o_C => w_C0);

  u_FULL_ADDER_1: FULL_ADDER_4_SC
  port map (i_A => i_A,
            i_B => i_B,
            i_C => k_one,
```

```
80              o_S => w_S1,
                o_C => w_C1);
82
    u_myMUX2_4_S: myMUX2_N
84  port map (i_0 => w_S0,
                i_1 => w_S1,
86              i_S => i_C,
                o_Z => o_S);
88
    u_myINV_C0 : myINV
90  port map (
      i_A => w_C0,
92    o_Z => w_C0_n);

94  u_myNAND_C11 : myNAND2
    port map (
96    i_A => w_C1,
      i_B => i_C,
98    o_Z => w_C11);

100 u_myNAND_Cout : myNAND2
    port map (
102   i_A => w_C11,
      i_B => w_C0_n,
104   o_Z => o_C);
  end a_CARRY_SELECT_ADDER_4_SC;
```

Listing A.27. CARRY_SELECT_ADDER_4_SC.vhd Code

1.8.10 Self-Contained 30-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_30_SC
--| Add two 30-bit numbers along with a carry in bit.
   Produce
--| a 30-bit sum and a carry out bit.
--|
--| INPUTS:
--| i_A - Input A
--| i_B - Input B
--| i_A_n - Inverted input A
--| i_B_n - Inverted input B
--| i_AB_AND - AND(A,B)
--| i_AB_NAND - NOT(AND(A,B))
--| i_AB_NOR - NOT(OR(A,B))
--| i_C - Carry In
--|
--| OUTPUTS:
--| o_S - i_A+i_B+i_C (sum)
--| o_C - Carry Out
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_30_SC is
  port (i_A      : in  std_logic_vector(29 downto 0);
        i_B      : in  std_logic_vector(29 downto 0);
        i_C      : in  std_logic;
        o_S      : out std_logic_vector(29 downto 0);
        o_C      : out std_logic);
end CARRY_SELECT_ADDER_30_SC;

architecture a_CARRY_SELECT_ADDER_30_SC of
   CARRY_SELECT_ADDER_30_SC is
--| Declare components
  component CARRY_SELECT_ADDER_4_SC is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;

  component CARRY_SELECT_ADDER_2_SC is
```

```
    port (i_A      : in  std_logic_vector(1 downto 0);
41        i_B      : in  std_logic_vector(1 downto 0);
          i_C      : in  std_logic;
43        o_S      : out std_logic_vector(1 downto 0);
          o_C      : out std_logic);
45  end component;

47--| Declare signals
  signal w_C1 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 0 to CARRY_SELECT_ADDER 1
49  signal w_C2 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 1 to CARRY_SELECT_ADDER 2
  signal w_C3 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 2 to CARRY_SELECT_ADDER 3
51  signal w_C4 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 3 to CARRY_SELECT_ADDER 4
  signal w_C5 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 4 to CARRY_SELECT_ADDER 5
53  signal w_C6 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 5 to CARRY_SELECT_ADDER 6
  signal w_C7 : std_logic; -- Carry out from
   CARRY_SELECT_ADDER 6 to CARRY_SELECT_ADDER 7
55 begin
  -- Connect CARRY_SELECT_ADDERs
57  u_CARRY_SELECT_ADDER_0: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(3 downto 0),
59          i_B => i_B(3 downto 0),
          i_C => i_C,
61          o_S => o_S(3 downto 0),
          o_C => w_C1);
63
  u_CARRY_SELECT_ADDER_1: CARRY_SELECT_ADDER_4_SC
65  port map (i_A => i_A(7 downto 4),
          i_B => i_B(7 downto 4),
67          i_C => w_C1,
          o_S => o_S(7 downto 4),
69          o_C => w_C2);

71  u_CARRY_SELECT_ADDER_2: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(11 downto 8),
73          i_B => i_B(11 downto 8),
          i_C => w_C2,
75          o_S => o_S(11 downto 8),
```

```
            o_C => w_C3);

  u_CARRY_SELECT_ADDER_3: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(15 downto 12),
            i_B => i_B(15 downto 12),
            i_C => w_C3,
            o_S => o_S(15 downto 12),
            o_C => w_C4);

  u_CARRY_SELECT_ADDER_4_SC: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(19 downto 16),
            i_B => i_B(19 downto 16),
            i_C => w_C4,
            o_S => o_S(19 downto 16),
            o_C => w_C5);

  u_CARRY_SELECT_ADDER_5: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(23 downto 20),
            i_B => i_B(23 downto 20),
            i_C => w_C5,
            o_S => o_S(23 downto 20),
            o_C => w_C6);

  u_CARRY_SELECT_ADDER_6: CARRY_SELECT_ADDER_4_SC
  port map (i_A => i_A(27 downto 24),
            i_B => i_B(27 downto 24),
            i_C => w_C6,
            o_S => o_S(27 downto 24),
            o_C => w_C7);

  u_CARRY_SELECT_ADDER_7: CARRY_SELECT_ADDER_2_SC
  port map (i_A => i_A(29 downto 28),
            i_B => i_B(29 downto 28),
            i_C => w_C7,
            o_S => o_S(29 downto 28),
            o_C => o_C);
end a_CARRY_SELECT_ADDER_30_SC;
```

Listing A.28. CARRY_SELECT_ADDER_30_SC.vhd Code

1.9 Load Upper Immediate

```
--| LUI - Load Upper Immediate
--| Perfomrs the operation Z = [Immediate B"0000
    _0000_0000_0000"]
--|
--| INPUTS:
--| A - Input A
--| B - Input B
--|
--| OUTPUTS:
--| Z - NAND(A,B)
library IEEE;
use IEEE.std_logic_1164.all;

entity LUI is
  port (i_A : in  std_logic_vector(15 downto 0);
        o_Z : out std_logic_vector(31 downto 0)
        );
end LUI;

architecture a_LUI of LUI is
  --| Declare constants
  constant k_zero16 : std_logic_vector(15 downto 0) := (
   others => '0');
begin
  o_Z <= i_A & k_zero16;
end a_LUI;
```

Listing A.29. LUI.vhd Code

1.10 Overflow Detection

```
--| Over_Detect.vhd
--| Detect overflow for ADD, ADDI, and SUB instructions. If an overflow occurs
--| for one of these instructions, the destination register should retain its
--| original contents. To accomplish this, the value of the destination register
--| (RD for ADD/SUB and RT for ADDI) is passed to the output of the ALU. This
--| value is then stored back to the destination register. If no overflow
--| occurs or no overflow detection is required, the output of the adder may
--| proceed to the ALU output and be stored to the destination register (assuming
--| that an operation making use of the adder has been selected for output).
--|
--| INPUTS:
--| i_AB_NAND  - NAND(A,B) from bitwise operations (only MSB, bit 31)
--| i_AB_OR    - OR(A,B) from bitwise operations (only MSB, bit 31)
--| i_AB_XNOR - XNOR(A,B) from bitwise operations (only MSB, bit 31)
--| i_S - Result of the addition
--| i_RD - Data from register indicated by RD
--| i_RT - Data from register indicated by RT
--| i_OVER_CTRL - Determine which of the signals to output.
--| 0 - i_S - no overflow control needed for the selected operation
--| 1 - i_RD - data from register indicated by RD should be passed through
--| 2 - i_RT - data from register indicated by RD should be passed through
--| 3 - i_S - no overflow control needed for the selected operation
--|
--| OUTPUTS:
--| o_Z - depends on the value of i_OVER_CTRL. If i_OVER_CTRL is 1 or 2, and
```

```
--|       an overflow is detected, then output either i_RD (1) or i_RT (2) so
--|       so that the result of the addition (subtraction) is not written to
--|    the destination register.  If no overflow is detected, output the
--|    result of the addition (subtraction)
library IEEE;
use IEEE.std_logic_1164.all;

entity Over_Detect is
  port (i_AB_NAND : in  std_logic;
        i_AB_OR   : in  std_logic;
        i_AB_XNOR : in  std_logic;
        i_S       : in  std_logic_vector(31 downto 0);
        i_RD      : in  std_logic_vector(31 downto 0);
        i_RT      : in  std_logic_vector(31 downto 0);
        i_OVER_CTRL : in  std_logic_vector(1 downto 0);
        o_Z       : out std_logic_vector(31 downto 0)
        );
end Over_Detect;

architecture a_Over_Detect of Over_Detect is
--| Declare Components
  -- Declare NAND_2
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;
  -- Declare Inverter
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare 4 input, 32-bit mux
  component myMUX4_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
```

```
66          i_2 : in  std_logic_vector(31 downto 0);
            i_3 : in  std_logic_vector(31 downto 0);
68          i_S : in  std_logic_vector(1 downto 0);
            o_Z : out std_logic_vector(31 downto 0)
70          );
    end component;
72
    -- Declare 2 input, 32-bit mux
74  component myMUX2_N is
      generic (m_width : integer := 32);
76    port (i_0 : in  std_logic_vector(31 downto 0);
            i_1 : in  std_logic_vector(31 downto 0);
78          i_S : in  std_logic;
            o_Z : out std_logic_vector(31 downto 0)
80          );
    end component;
82
    --| Declare Signals
84  signal w_S31_n : std_logic;  -- INV(i_S(31))
    signal w_OV_0 : std_logic;
86  signal w_OV_1 : std_logic;
    signal w_OV_2, w_OV_2_n : std_logic;
88  signal w_OV_3, w_OV_3_n : std_logic;
    signal w_data : std_logic_vector(31 downto 0); -- Output
     from MUX4_32 and input to MUX2_32
90 begin
    -- Create Inverets for signals that must be inverted
92  u_myINV_S31_n : myINV
    port map (
94    i_A => i_S(31),
      o_Z => w_S31_n);
96
    u_myINV_OV_2_n : myINV
98  port map (
      i_A => w_OV_2,
100   o_Z => w_OV_2_n);

102 u_myINV_OV_3_n : myINV
    port map (
104   i_A => w_OV_3,
      o_Z => w_OV_3_n); -- w_OV_3_n is 1 if an overflow occured
      and if the instruction is ADD, SUB, or ADDI, 0 otherwise
106
```

```
  --Create intermediate signals
108 u_myNAND_OV_0 : myNAND2
  port map (
110   i_A => i_S(31),
    i_B => i_AB_OR,
112   o_Z => w_OV_0);

114 u_myNAND_OV_1 : myNAND2
  port map (
116   i_A => w_S31_n,
    i_B => i_AB_NAND,
118   o_Z => w_OV_1);

120 u_myNAND_OV_2 : myNAND2
  port map (
122   i_A => w_OV_0,
    i_B => w_OV_1,
124   o_Z => w_OV_2);

126 u_myNAND_OV_3 : myNAND2
  port map (
128   i_A => i_AB_XNOR,
    i_B => w_OV_2_n,
130   o_Z => w_OV_3);

132 -- Create MUXes to select output.
  -- If no overflow detection is required for the given
   instruction,
134 -- pass the output of the adder.  If this is an ADD or SUB
  -- instruction, pass the value of the register pointed to
   by RD.
136 -- If this is an ADDI instruction, pass the value of the
   register
  -- pointed to by RT.
138 u_myMUX4_32: myMUX4_N
  generic map (m_width => 32)
140 port map (i_0 => i_S,  -- No Overflow Detection
        i_1 => i_RD, -- ADD
142         i_2 => i_RD, -- SUB
        i_3 => i_RT, -- ADDI
144         i_S => i_OVER_CTRL,
        o_Z => w_data);
146
```

```
  -- If an overflow occured, pass the selected data from the
   output
148 -- of the MUX4_32.  If no overflow occured, pass the result
   from the
  -- adder.
150 u_myMUX2_32: myMUX2_N
  generic map (m_width => 32)
152 port map (i_0 => i_S,
        i_1 => w_data,
154         i_S => w_OV_3_n,
        o_Z => o_Z);
156
end a_Over_Detect;
```

Listing A.30. Over_Detect.vhd Code

1.11 Comparator

```
1  --| Comparator.vhd
   --| Compare the input to zero and create equal, not equal,
   less than, less than or equal
3  --| to, greater than, and greater than or equal to signals.
   Additionally, accept a control
   --| signal that determines which of these signals should be
   output. In the event that an
5  --| overflow occurs in the adder, the MSB of inputs A and B
   to the adder must be utilized
   --| to determine if an overflow has occured. This also
   depends on whether the operation
7  --| was a signed or unsigned operation. The branch
   operations and set-on-less-than
   --| (immediate) instructions all make use of signed values
   while the set-on-less-than
9  --| (immediate) unsigned operations make use of unsigned
   values.
   --|
11 --| INPUTS:
   --| i_S - Data to compare to zero
13 --| i_NAND_AB - AND(A(31),B(31))
   --| i_OR_AB   - OR(A(31),B(31))
15 --| i_XNOR_AB - XNOR(A(31),B(31))
   --| i_unsigned - 1 if the adder operands are unsigned and 0
   if they are signed
17 --| i_overflow  - Determine if the comparator is performing a
    Less Than comparison that will
   --|              or will not produce an overflow. No
   overflow can occur in BLTZ instructions,
19 --|         but overflow can occur in SLT, SLTU, SLTI, and
   SLTIU instructions. 0 - no
   --|         overflow can occur, 1 - overflow can occur
21 --| i_COMP_SEL - Determine which of the signals to output.
   --| 0 - Greater than or equal to zero
23 --| 1 - Less than zero
   --| 2 - Equal to zero
25 --| 3 - Not equal to zero
   --| 4 - Less than or equal to zero
27 --| 5 - Greater than zero
   --| 6,7 - Greater than or equal to zero
29 --|
```

```
--| OUTPUTS:
--| o_Z - Selected comparison
library IEEE;
use IEEE.std_logic_1164.all;

entity Comparator is
  port (i_S      : in  std_logic_vector(31 downto 0);
        i_NAND_AB : in  std_logic;
        i_OR_AB   : in  std_logic;
        i_XNOR_AB : in  std_logic;
        i_unsigned  : in  std_logic;
        i_overflow  : in  std_logic;
        i_COMP_SEL  : in  std_logic_vector(2 downto 0);
        o_Z       : out std_logic);
end Comparator;

architecture a_Comparator of Comparator is
--| Declare Components
  -- Declare NAND_2
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare Inverter
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare 2 input, 1-bit mux
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare 8 input, 1-bit mux
```

```
73  component myMUX8_1 is
      port (i_0 : in  std_logic;
75          i_1 : in  std_logic;
            i_2 : in  std_logic;
77          i_3 : in  std_logic;
            i_4 : in  std_logic;
79          i_5 : in  std_logic;
            i_6 : in  std_logic;
81          i_7 : in  std_logic;
            i_S : in  std_logic_vector(2 downto 0);
83          o_Z : out std_logic
            );
85  end component;

87  --| Declare Signals
    signal w_GEZ            : std_logic; --w_GEZ0, w_GEZ1,
     w_GEZ  : std_logic; -- Greater than or equal to zero
89  signal w_LTZ0, w_LTZ1, w_LTZ  : std_logic; -- Less than
     zero
    signal w_EQ             : std_logic; --w_EQ0,   w_EQ
       : std_logic; -- Equal to zero
91  signal w_NE             : std_logic; --w_NE0,   w_NE
       : std_logic; -- Not equal to zero
    signal w_LEZ            : std_logic; --w_LEZ0, w_LEZ
       : std_logic; -- Less than or equal to zero
93  signal w_GTZ            : std_logic; --w_GTZ0, w_GTZ
       : std_logic; -- Greater than zero
    -- Inverted Input Signals
95  signal w_S31_n  : std_logic;
    signal w_S30_n  : std_logic;
97  signal w_S29_n  : std_logic;
    signal w_S28_n  : std_logic;
99  signal w_S27_n  : std_logic;
    signal w_S26_n  : std_logic;
101 signal w_S25_n  : std_logic;
    signal w_S24_n  : std_logic;
103 signal w_S23_n  : std_logic;
    signal w_S22_n  : std_logic;
105 signal w_S21_n  : std_logic;
    signal w_S20_n  : std_logic;
107 signal w_S19_n  : std_logic;
    signal w_S18_n  : std_logic;
109 signal w_S17_n  : std_logic;
```

```
    signal w_S16_n  : std_logic;
    signal w_S15_n  : std_logic;
    signal w_S14_n  : std_logic;
    signal w_S13_n  : std_logic;
    signal w_S12_n  : std_logic;
    signal w_S11_n  : std_logic;
    signal w_S10_n  : std_logic;
    signal w_S09_n  : std_logic;
    signal w_S08_n  : std_logic;
    signal w_S07_n  : std_logic;
    signal w_S06_n  : std_logic;
    signal w_S05_n  : std_logic;
    signal w_S04_n  : std_logic;
    signal w_S03_n  : std_logic;
    signal w_S02_n  : std_logic;
    signal w_S01_n  : std_logic;
    signal w_S00_n  : std_logic;

    -- Intermediate terms in the 32-bit OR function
    signal w_S31_30, w_S31_30_n : std_logic;
    signal w_S29_28, w_S29_28_n : std_logic;
    signal w_S27_26, w_S27_26_n : std_logic;
    signal w_S25_24, w_S25_24_n : std_logic;
    signal w_S23_22, w_S23_22_n : std_logic;
    signal w_S21_20, w_S21_20_n : std_logic;
    signal w_S19_18, w_S19_18_n : std_logic;
    signal w_S17_16, w_S17_16_n : std_logic;
    signal w_S15_14, w_S15_14_n : std_logic;
    signal w_S13_12, w_S13_12_n : std_logic;
    signal w_S11_10, w_S11_10_n : std_logic;
    signal w_S09_08, w_S09_08_n : std_logic;
    signal w_S07_06, w_S07_06_n : std_logic;
    signal w_S05_04, w_S05_04_n : std_logic;
    signal w_S03_02, w_S03_02_n : std_logic;
    signal w_S01_00, w_S01_00_n : std_logic;

    signal w_S31_28, w_S31_28_n : std_logic;
    signal w_S27_24, w_S27_24_n : std_logic;
    signal w_S23_20, w_S23_20_n : std_logic;
    signal w_S19_16, w_S19_16_n : std_logic;
    signal w_S15_12, w_S15_12_n : std_logic;
    signal w_S11_08, w_S11_08_n : std_logic;
    signal w_S07_04, w_S07_04_n : std_logic;
```

```
153   signal w_S03_00, w_S03_00_n : std_logic;

155   signal w_S31_24, w_S31_24_n : std_logic;
      signal w_S23_16, w_S23_16_n : std_logic;
157   signal w_S15_08, w_S15_08_n : std_logic;
      signal w_S07_00, w_S07_00_n : std_logic;
159
      signal w_S31_16, w_S31_16_n : std_logic;
161   signal w_S15_00, w_S15_00_n : std_logic;
    -- Result of the 32-bit or function
163   signal w_S_OR : std_logic;
    -- Intermediate signals to determine comparisons when an
     overflow is possible
165--  signal w_comp0 : std_logic;
--    signal w_comp1 : std_logic;
167--  signal w_comp2, w_comp2_n : std_logic;
--    signal w_comp3 : std_logic;
169   signal w_comp4 : std_logic;
      signal w_comp5 : std_logic;
171   signal w_comp6, w_comp6_n : std_logic;
      signal w_comp7 : std_logic;
173   signal w_unsigned_n : std_logic; -- inverted i_unsigned
      signal w_overcheck : std_logic;
175   signal w_overcheck_n : std_logic;

177   -- Define Constants
      constant k_zero : std_logic := '0'; -- w_EQ
179   constant k_one  : std_logic := '1';
  begin
181   -- If i_S(31) is 1, the incoming number is negative and
       therefore less than 0.
      -- However, this can be erroneous if an overflow occured.
       Another means is
183   -- used to handle overflows and override this value
      w_LTZ0 <= i_S(31);
185
      --| Instantiate Inverters to create inverted signals
187   u_myINV_31 : myINV
      port map (
189     i_A => i_S(31),
        o_Z => w_S31_n);
191
      u_myINV_30 : myINV
```

```
193     port map (
          i_A => i_S(30),
195       o_Z => w_S30_n);

197     u_myINV_29 : myINV
        port map (
199       i_A => i_S(29),
          o_Z => w_S29_n);
201
        u_myINV_28 : myINV
203     port map (
          i_A => i_S(28),
205       o_Z => w_S28_n);

207     u_myINV_27 : myINV
        port map (
209       i_A => i_S(27),
          o_Z => w_S27_n);
211
        u_myINV_26 : myINV
213     port map (
          i_A => i_S(26),
215       o_Z => w_S26_n);

217     u_myINV_25 : myINV
        port map (
219       i_A => i_S(25),
          o_Z => w_S25_n);
221
        u_myINV_24 : myINV
223     port map (
          i_A => i_S(24),
225       o_Z => w_S24_n);

227     u_myINV_23 : myINV
        port map (
229       i_A => i_S(23),
          o_Z => w_S23_n);
231
        u_myINV_22 : myINV
233     port map (
          i_A => i_S(22),
235       o_Z => w_S22_n);
```

```
237   u_myINV_21 : myINV
      port map (
239     i_A => i_S(21),
        o_Z => w_S21_n);
241
      u_myINV_20 : myINV
243   port map (
        i_A => i_S(20),
245     o_Z => w_S20_n);

247   u_myINV_19 : myINV
      port map (
249     i_A => i_S(19),
        o_Z => w_S19_n);
251
      u_myINV_18 : myINV
253   port map (
        i_A => i_S(18),
255     o_Z => w_S18_n);

257   u_myINV_17 : myINV
      port map (
259     i_A => i_S(17),
        o_Z => w_S17_n);
261
      u_myINV_16 : myINV
263   port map (
        i_A => i_S(16),
265     o_Z => w_S16_n);

267   u_myINV_15 : myINV
      port map (
269     i_A => i_S(15),
        o_Z => w_S15_n);
271
      u_myINV_14 : myINV
273   port map (
        i_A => i_S(14),
275     o_Z => w_S14_n);

277   u_myINV_13 : myINV
      port map (
```

```
279     i_A => i_S(13),
        o_Z => w_S13_n);
281
    u_myINV_12 : myINV
283 port map (
        i_A => i_S(12),
285     o_Z => w_S12_n);

287 u_myINV_11 : myINV
    port map (
289     i_A => i_S(11),
        o_Z => w_S11_n);
291
    u_myINV_10 : myINV
293 port map (
        i_A => i_S(10),
295     o_Z => w_S10_n);

297 u_myINV_09 : myINV
    port map (
299     i_A => i_S(9),
        o_Z => w_S09_n);
301
    u_myINV_08 : myINV
303 port map (
        i_A => i_S(8),
305     o_Z => w_S08_n);

307 u_myINV_07 : myINV
    port map (
309     i_A => i_S(7),
        o_Z => w_S07_n);
311
    u_myINV_06 : myINV
313 port map (
        i_A => i_S(6),
315     o_Z => w_S06_n);

317 u_myINV_05 : myINV
    port map (
319     i_A => i_S(05),
        o_Z => w_S05_n);
321
```

```
    u_myINV_04 : myINV
323 port map (
      i_A => i_S(4),
325   o_Z => w_S04_n);

327 u_myINV_03 : myINV
    port map (
329   i_A => i_S(3),
      o_Z => w_S03_n);
331
    u_myINV_02 : myINV
333 port map (
      i_A => i_S(2),
335   o_Z => w_S02_n);

337 u_myINV_01 : myINV
    port map (
339   i_A => i_S(1),
      o_Z => w_S01_n);
341
    u_myINV_00 : myINV
343 port map (
      i_A => i_S(0),
345   o_Z => w_S00_n);

347 u_myINV_31_30 : myINV
    port map (
349   i_A => w_S31_30,
      o_Z => w_S31_30_n);
351
    u_myINV_29_28 : myINV
353 port map (
      i_A => w_S29_28,
355   o_Z => w_S29_28_n);

357 u_myINV_27_26 : myINV
    port map (
359   i_A => w_S27_26,
      o_Z => w_S27_26_n);
361
    u_myINV_25_24 : myINV
363 port map (
      i_A => w_S25_24,
```

```
365       o_Z => w_S25_24_n);

367     u_myINV_23_22 : myINV
        port map (
369       i_A => w_S23_22,
          o_Z => w_S23_22_n);
371
        u_myINV_21_20 : myINV
373     port map (
          i_A => w_S21_20,
375       o_Z => w_S21_20_n);

377     u_myINV_19_18 : myINV
        port map (
379       i_A => w_S19_18,
          o_Z => w_S19_18_n);
381
        u_myINV_17_16 : myINV
383     port map (
          i_A => w_S17_16,
385       o_Z => w_S17_16_n);

387     u_myINV_15_14 : myINV
        port map (
389       i_A => w_S15_14,
          o_Z => w_S15_14_n);
391
        u_myINV_13_12 : myINV
393     port map (
          i_A => w_S13_12,
395       o_Z => w_S13_12_n);

397     u_myINV_11_10 : myINV
        port map (
399       i_A => w_S11_10,
          o_Z => w_S11_10_n);
401
        u_myINV_09_08 : myINV
403     port map (
          i_A => w_S09_08,
405       o_Z => w_S09_08_n);

407     u_myINV_07_06 : myINV
```

```
    port map (
409   i_A => w_S07_06,
      o_Z => w_S07_06_n);
411
    u_myINV_05_04 : myINV
413 port map (
      i_A => w_S05_04,
415   o_Z => w_S05_04_n);

417 u_myINV_03_02 : myINV
    port map (
419   i_A => w_S03_02,
      o_Z => w_S03_02_n);
421
    u_myINV_01_00 : myINV
423 port map (
      i_A => w_S01_00,
425   o_Z => w_S01_00_n);

427 u_myINV_31_28 : myINV
    port map (
429   i_A => w_S31_28,
      o_Z => w_S31_28_n);
431
    u_myINV_27_24 : myINV
433 port map (
      i_A => w_S27_24,
435   o_Z => w_S27_24_n);

437 u_myINV_23_20 : myINV
    port map (
439   i_A => w_S23_20,
      o_Z => w_S23_20_n);
441
    u_myINV_19_16 : myINV
443 port map (
      i_A => w_S19_16,
445   o_Z => w_S19_16_n);

447 u_myINV_15_12 : myINV
    port map (
449   i_A => w_S15_12,
      o_Z => w_S15_12_n);
```

```
451
    u_myINV_11_08 : myINV
453 port map (
      i_A => w_S11_08,
455   o_Z => w_S11_08_n);

457 u_myINV_07_04 : myINV
    port map (
459   i_A => w_S07_04,
      o_Z => w_S07_04_n);
461
    u_myINV_03_00 : myINV
463 port map (
      i_A => w_S03_00,
465   o_Z => w_S03_00_n);

467 u_myINV_31_24 : myINV
    port map (
469   i_A => w_S31_24,
      o_Z => w_S31_24_n);
471
    u_myINV_23_16 : myINV
473 port map (
      i_A => w_S23_16,
475   o_Z => w_S23_16_n);

477 u_myINV_15_08 : myINV
    port map (
479   i_A => w_S15_08,
      o_Z => w_S15_08_n);
481
    u_myINV_07_00 : myINV
483 port map (
      i_A => w_S07_00,
485   o_Z => w_S07_00_n);

487 u_myINV_31_16 : myINV
    port map (
489   i_A => w_S31_16,
      o_Z => w_S31_16_n);
491
    u_myINV_15_00 : myINV
493 port map (
```

```
    i_A => w_S15_00,
    o_Z => w_S15_00_n);

  u_myINV_overcheck : myINV
  port map (
    i_A => w_overcheck_n,
    o_Z => w_overcheck);

  u_myNAND_31_30 : myNAND2
  port map (
    i_A => w_S31_n,
    i_B => w_S30_n,
    o_Z => w_S31_30);

  u_myNAND_29_28 : myNAND2
  port map (
    i_A => w_S29_n,
    i_B => w_S28_n,
    o_Z => w_S29_28);

  u_myNAND_27_26 : myNAND2
  port map (
    i_A => w_S27_n,
    i_B => w_S26_n,
    o_Z => w_S27_26);

  u_myNAND_25_24 : myNAND2
  port map (
    i_A => w_S25_n,
    i_B => w_S24_n,
    o_Z => w_S25_24);

  u_myNAND_23_22 : myNAND2
  port map (
    i_A => w_S23_n,
    i_B => w_S22_n,
    o_Z => w_S23_22);

  u_myNAND_21_20 : myNAND2
  port map (
    i_A => w_S21_n,
    i_B => w_S20_n,
    o_Z => w_S21_20);
```

```
537
    u_myNAND_19_18 : myNAND2
539 port map (
      i_A => w_S19_n,
541   i_B => w_S18_n,
      o_Z => w_S19_18);
543
    u_myNAND_17_16 : myNAND2
545 port map (
      i_A => w_S17_n,
547   i_B => w_S16_n,
      o_Z => w_S17_16);
549
    u_myNAND_15_14 : myNAND2
551 port map (
      i_A => w_S15_n,
553   i_B => w_S14_n,
      o_Z => w_S15_14);
555
    u_myNAND_13_12 : myNAND2
557 port map (
      i_A => w_S13_n,
559   i_B => w_S12_n,
      o_Z => w_S13_12);
561
    u_myNAND_11_10 : myNAND2
563 port map (
      i_A => w_S11_n,
565   i_B => w_S10_n,
      o_Z => w_S11_10);
567
    u_myNAND_09_08 : myNAND2
569 port map (
      i_A => w_S09_n,
571   i_B => w_S08_n,
      o_Z => w_S09_08);
573
    u_myNAND_07_06 : myNAND2
575 port map (
      i_A => w_S07_n,
577   i_B => w_S06_n,
      o_Z => w_S07_06);
579
```

```
    u_myNAND_05_04 : myNAND2
581 port map (
      i_A => w_S05_n,
583   i_B => w_S04_n,
      o_Z => w_S05_04);
585
    u_myNAND_03_02 : myNAND2
587 port map (
      i_A => w_S03_n,
589   i_B => w_S02_n,
      o_Z => w_S03_02);
591
    u_myNAND_01_00 : myNAND2
593 port map (
      i_A => w_S01_n,
595   i_B => w_S00_n,
      o_Z => w_S01_00);
597
    u_myNAND_31_28 : myNAND2
599 port map (
      i_A => w_S31_30_n,
601   i_B => w_S29_28_n,
      o_Z => w_S31_28);
603
    u_myNAND_27_24 : myNAND2
605 port map (
      i_A => w_S27_26_n,
607   i_B => w_S25_24_n,
      o_Z => w_S27_24);
609
    u_myNAND_23_20 : myNAND2
611 port map (
      i_A => w_S23_22_n,
613   i_B => w_S21_20_n,
      o_Z => w_S23_20);
615
    u_myNAND_19_16 : myNAND2
617 port map (
      i_A => w_S19_18_n,
619   i_B => w_S17_16_n,
      o_Z => w_S19_16);
621
    u_myNAND_15_12 : myNAND2
```

```
623     port map (
          i_A => w_S15_14_n,
625       i_B => w_S13_12_n,
          o_Z => w_S15_12);
627
        u_myNAND_11_08 : myNAND2
629     port map (
          i_A => w_S11_10_n,
631       i_B => w_S09_08_n,
          o_Z => w_S11_08);
633
        u_myNAND_07_04 : myNAND2
635     port map (
          i_A => w_S07_06_n,
637       i_B => w_S05_04_n,
          o_Z => w_S07_04);
639
        u_myNAND_03_00 : myNAND2
641     port map (
          i_A => w_S03_02_n,
643       i_B => w_S01_00_n,
          o_Z => w_S03_00);
645
        u_myNAND_31_24 : myNAND2
647     port map (
          i_A => w_S31_28_n,
649       i_B => w_S27_24_n,
          o_Z => w_S31_24);
651
        u_myNAND_23_16 : myNAND2
653     port map (
          i_A => w_S23_20_n,
655       i_B => w_S19_16_n,
          o_Z => w_S23_16);
657
        u_myNAND_15_08 : myNAND2
659     port map (
          i_A => w_S15_12_n,
661       i_B => w_S11_08_n,
          o_Z => w_S15_08);
663
        u_myNAND_07_00 : myNAND2
665     port map (
```

```
    i_A => w_S07_04_n,
    i_B => w_S03_00_n,
    o_Z => w_S07_00);

u_myNAND_31_16 : myNAND2
port map (
    i_A => w_S31_24_n,
    i_B => w_S23_16_n,
    o_Z => w_S31_16);

u_myNAND_15_00 : myNAND2
port map (
    i_A => w_S15_08_n,
    i_B => w_S07_00_n,
    o_Z => w_S15_00);

u_myNAND_overcheck : myNAND2
port map (
    i_A => i_overflow,
    i_B => i_XNOR_AB,
    o_z => w_overcheck_n);

-- Compute value of Not Equal to Zero. This can be erroneous if an
-- overflow occured. Another means is used to handle overflows and
-- override this value
u_myNAND_NE : myNAND2
port map (
    i_A => w_S31_16_n,
    i_B => w_S15_00_n,
    o_Z => w_NE);
    --o_Z => w_NE0);

-- Compute value of Greater Than or Equal to Zero. This can be erroneous if an
-- overflow occured. Another means is used to handle overflows and
-- override this value
u_myINV_GEZ : myINV
port map (
    i_A => w_LTZ0,
    o_Z => w_GEZ);
```

```
705 --  u_myINV_GEZ : myINV
    --  port map (
707 --    i_A => w_LTZ0,
    --    o_Z => w_GEZ0);
709
    -- Compute value of Equal to Zero. This can be erroneous if
       an
711 -- overflow occured.  Another means is used to handle
     overflows and
    -- override this value
713 u_myINV_EQ : myINV
    port map (
715   i_A => w_NE,
      o_Z => w_EQ);
717 --  u_myINV_EQ : myINV
    --  port map (
719 --    i_A => w_NE0,
    --    o_Z => w_EQ0);
721
    -- Compute value of Less Than or Equal to Zero. This can be
       erroneous if an
723 -- overflow occured.  Another means is used to handle
     overflows and
    -- override this value
725 u_myNAND_LEZ : myNAND2
    port map (
727   i_A => w_GEZ,
      i_B => w_NE,
729   o_Z => w_LEZ);
    --  u_myNAND_LEZ : myNAND2
731 --  port map (
    --    i_A => w_GEZ0,
733 --    i_B => w_NE0,
    --    o_Z => w_LEZ0);
735
    -- Compute value of Greater Than Zero. This can be
     erroneous if an
737 -- overflow occured.  Another means is used to handle
     overflows and
    -- override this value
739 u_myINV_GTZ : myINV
    port map (
741   i_A => w_LEZ,
```

```
    o_Z => w_GTZ);
--  u_myINV_GTZ : myINV
--  port map (
--    i_A => w_LEZ0,
--    o_Z => w_GTZ0);

  -- Invert the i_signed input
  u_myINV_unsigned : myINV
  port map (
    i_A => i_unsigned,
    o_Z => w_unsigned_n);
  -- Compute GTZ, GEZ assuming an overflow occured
--  u_myNAND_comp0 : myNAND2
--  port map (
--    i_A => i_unsigned,
--    i_B => i_NAND_AB,
--    o_Z => w_comp0);
--
--  u_myNAND_comp1 : myNAND2
--  port map (
--    i_A => w_unsigned_n,
--    i_B => i_OR_AB,
--    o_Z => w_comp1);
--
--  u_myNAND_comp2 : myNAND2
--  port map (
--    i_A => w_comp0,
--    i_B => w_comp1,
--    o_Z => w_comp2);
--
--  u_myINV_comp2_n : myINV
--  port map (
--    i_A => w_comp2,
--    o_Z => w_comp2_n);
--
--  u_myNAND_comp3 : myNAND2
--  port map (
--    i_A => w_comp2_n,
--    i_B => i_XNOR_AB,
--    o_Z => w_comp3);
--
--  u_myINV_GZ : myINV
--  port map (
```

```
785 --      i_A => w_comp3,
    --      o_Z => w_GEZ1);
787
    -- Compute LTZ, LEZ assuming an overflow occured
789 u_myNAND_comp4 : myNAND2
    port map (
791   i_A => i_unsigned,
      i_B => i_OR_AB,
793   o_Z => w_comp4);

795 u_myNAND_comp5 : myNAND2
    port map (
797   i_A => w_unsigned_n,
      i_B => i_NAND_AB,
799   o_Z => w_comp5);

801 u_myNAND_comp6 : myNAND2
    port map (
803   i_A => w_comp4,
      i_B => w_comp5,
805   o_Z => w_comp6);

807 u_myINV_comp6_n : myINV
    port map (
809   i_A => w_comp6,
      o_Z => w_comp6_n);
811
    u_myNAND_comp7 : myNAND2
813 port map (
      i_A => w_comp6_n,
815   i_B => i_XNOR_AB,
      o_Z => w_comp7);
817
    u_myINV_LZ : myINV
819 port map (
      i_A => w_comp7,
821   o_Z => w_LTZ1);

823 -- Determine whether ALU comparisons or Overflow
     Comparisons should be used
--  u_myMUX2_GEZ : myMUX2_1
825 --  port map (i_0 => w_GEZ0,
    --           i_1 => w_GEZ1,
```

```
827 --           i_S => i_XNOR_AB,
    --           o_Z => w_GEZ);
829
    u_myMUX2_LTZ : myMUX2_1
831 port map (i_0 => w_LTZ0,
           i_1 => w_LTZ1,
833        i_S => w_overcheck,
    --       i_S => i_XNOR_AB,
835        o_Z => w_LTZ);

837 --  u_myMUX2_EQ : myMUX2_1
    --  port map (i_0 => w_EQ0,
839 --           i_1 => k_zero,
    --           i_S => i_XNOR_AB,
841 --           o_Z => w_EQ);
    --
843 --  u_myMUX2_NE : myMUX2_1
    --  port map (i_0 => w_NE0,
845 --           i_1 => k_one,
    --           i_S => i_XNOR_AB,
847 --           o_Z => w_NE);
    --
849 --  u_myMUX2_LEZ : myMUX2_1
    --  port map (i_0 => w_LEZ0,
851 --           i_1 => w_LTZ1,
    --           i_S => i_XNOR_AB,
853 --           o_Z => w_LEZ);
    --
855 --  u_myMUX2_GTZ : myMUX2_1
    --  port map (i_0 => w_GTZ0,
857 --           i_1 => w_GEZ1,
    --           i_S => i_XNOR_AB,
859 --           o_Z => w_GTZ);

861  -- Create MUX to select output.
    u_myMUX8_1: myMUX8_1
863 port map (i_0 => w_GEZ,
           i_1 => w_LTZ,
865        i_2 => w_EQ,
           i_3 => w_NE,
867        i_4 => w_LEZ,
           i_5 => w_GTZ,
869        i_6 => w_GEZ,
```

```
            i_7 => w_GEZ,
871         i_S => i_COMP_SEL,
            o_Z => o_Z);
873 end a_Comparator;
```

Listing A.31. Comparator.vhd Code

1.12 Signed/Unsigned Extensions

1.12.1 Signed 16-Bit to 32-Bit Extension

```
--| Sign_Extend16_32.vhd
--| Sign Extend a 16-bit signed value to a 32-bit signed value
--|
--| INPUTS:
--| i_A - Input A
--|
--| OUTPUTS:
--| o_Z - Sign extended input
library IEEE;
use IEEE.std_logic_1164.all;

entity Sign_Extend16_32 is
  port (i_A : in  std_logic_vector(15 downto 0);
        o_Z : out std_logic_vector(31 downto 0));
end Sign_Extend16_32;

architecture a_Sign_Extend16_32 of Sign_Extend16_32 is
begin
  o_Z <= i_A(15) & i_A(15) & i_A(15) & i_A(15) &
         i_A(15) & i_A(15) & i_A(15) & i_A(15) &
         i_A(15) & i_A(15) & i_A(15) & i_A(15) &
         i_A(15) & i_A(15) & i_A(15) & i_A(15) & i_A;
end a_Sign_Extend16_32;
```

Listing A.32. Sign_Extend16_32.vhd Code

1.12.2 Unsigned 16-Bit to 32-Bit Extension

```
--| Extend16_32.vhd
--| Zero Extend a 16-bit signed value to a 32-bit value
--|
--| INPUTS:
--| i_A - Input A
--|
--| OUTPUTS:
--| o_Z - Sign extended input
library IEEE;
use IEEE.std_logic_1164.all;

entity Extend16_32 is
  port (i_A : in  std_logic_vector(15 downto 0);
        o_Z : out std_logic_vector(31 downto 0));
end Extend16_32;

architecture a_Extend16_32 of Extend16_32 is
begin
  o_Z <= B"0000000000000000" & i_A;
end a_Extend16_32;
```

Listing A.33. Extend16_32.vhd Code

1.12.3 Unsigned 5-Bit to 32-Bit Extension

```
--| Extend5_32.vhd
--| Zero Extend a 5-bit value to a 32-bit value
--|
--| INPUTS:
--| i_A - Input A
--|
--| OUTPUTS:
--| o_Z - Extended input
library IEEE;
use IEEE.std_logic_1164.all;

entity Extend5_32 is
  port (i_A : in  std_logic_vector(4 downto 0);
        o_Z : out std_logic_vector(31 downto 0));
end Extend5_32;

architecture a_Extend5_32 of Extend5_32 is
begin
  o_Z <= B"000_0000_0000_0000_0000_0000_0000" & i_A;
end a_Extend5_32;
```

Listing A.34. Extend5_32.vhd Code

1.13 1-Bit Register

```
--| myReg1.vhd
--| 1-bit register
--|
--| INPUTS:
--| i_clk - Clock
--| i_reset - Reset
--| i_D   - Data
--|
--| OUTPUTS:
--| o_Q - Output
library IEEE;
use IEEE.std_logic_1164.all;

entity myReg1 is
  port (i_clk   : in  std_logic;
        i_reset : in  std_logic;
        i_D     : in  std_logic;
        o_Q     : out std_logic);
end myReg1;

architecture a_myReg1 of myReg1 is
--| Declare signals
  signal f_data : std_logic := '0';
begin
  o_Q <= f_data;

  d_flip_flop: process(i_clk, i_reset)
  begin
    if (i_reset = '1') then
      f_data <= '0';
    elsif rising_edge(i_clk) then
      f_data <= i_D;
    end if;
  end process;
end a_myReg1;
```

Listing A.35. myReg1.vhd Code

1.14 ALU Core

```
1  --| ALU_CORE.vhd
   --| Determine the A and B inputs to the ALU submodules,
   connect all of the
3  --| ALU submodules, and select which submodule should have
   its output be
   --| ALU output.
5  --|
   --| INPUTS:
7  --| i_ALU_SRC_A - ALU Source A selector
   --| i_ALU_SRC_B - ALU Source B selector
9  --| i_ALU_INV_B - ALU Invert B selector
   --| i_COMP_SEL  - Select the value to output from the
   Comparator
11 --| i_OVER_CTRL - Select which overflow detection means
   should be used
   --|           (i.e. ADD, ADDI, SUB, or no overflow detection)
   .
13 --| i_ALU_OUTPUT  - Select which ALU function should be
   output
   --|           0 - Shift Left
15 --|           1 - Shift Right
   --|           2 - Adder
17 --|           3 - AND
   --|           4 - OR
19 --|           5 - XOR
   --|           6 - NOR
21 --|           7 - Comparator
   --|           8 - LUI
23 --|           9, 10, 11, 12, 13, 14, 15 - Shift Left
   --| i_RS_DATA   - Data from the register specified by RS
25 --| i_RT_DATA   - Data from the register specified by RS
   --| i_RD_DATA   - Data from the register specified by RS
27 --| i_PC        - Address stored in the PC register
   --| i_imm     - Immediate/offset value
29 --| i_UNSIGNED    - Determine if the comparator is comparing
   sigend numbers
   --|           (0) or unsigned numbers (1).
31 --| i_overflow    - Determine if the comparator is performing
    a Less Than comparison that will
   --|                 or will not produce an overflow.  No
   overflow can occur in BLTZ instructions,
```

```
33 --|            but overflow can occur in SLT, SLTU, SLTI, and
   SLTIU instructions.  0 - no
--|            overflow can occur, 1 - overflow can occur
35 --| i_imm_extend  - Determine if the immediate value should
   be 0 extended or
--|            sign extended. 0 for zero extended.  1 for sign
    extended.
37 --|
--| OUTPUTS:
39 --| o_ALU_RESULT
library IEEE;
41 use IEEE.std_logic_1164.all;

43 entity ALU_CORE is
  port (i_clk        : in std_logic;
45       i_reset      : in std_logic;
       i_ALU_SRC_A    : in  std_logic_vector(1 downto 0);
47       i_ALU_SRC_B    : in  std_logic_vector(1 downto 0);
       i_ALU_INV_B    : in  std_logic;
49       i_COMP_SEL     : in  std_logic_vector(2 downto 0);
       i_OVER_CTRL    : in  std_logic_vector(1 downto 0);
51       i_ALU_OUTPUT   : in  std_logic_vector(3 downto 0);
       i_RS_DATA    : in  std_logic_vector(31 downto 0);
53       i_RT_DATA    : in  std_logic_vector(31 downto 0);
       i_RD_DATA    : in  std_logic_vector(31 downto 0);
55       i_PC         : in  std_logic_vector(31 downto 0);
       i_imm        : in  std_logic_vector(15 downto 0);
57       i_UNSIGNED     : in  std_logic;
       i_overflow     : in  std_logic;
59       i_imm_extend   : in  std_logic;
       o_ALU_RESULT   : out std_logic_vector(31 downto 0));
61 end ALU_CORE;

63 architecture a_ALU_CORE of ALU_CORE is
--| Define Components
65   component myINV_N is
    generic(m_width : integer := 32);
67    port (i_A : in  std_logic_vector(m_width-1 downto 0);
        o_Z : out std_logic_vector(m_width-1 downto 0));
69   end component;

71   component myMUX16_N is
    generic (m_width : integer := 32);
```

```
73      port (i_0 : in  std_logic_vector(m_width-1 downto 0);
              i_1 : in  std_logic_vector(m_width-1 downto 0);
75            i_2 : in  std_logic_vector(m_width-1 downto 0);
              i_3 : in  std_logic_vector(m_width-1 downto 0);
77            i_4 : in  std_logic_vector(m_width-1 downto 0);
              i_5 : in  std_logic_vector(m_width-1 downto 0);
79            i_6 : in  std_logic_vector(m_width-1 downto 0);
              i_7 : in  std_logic_vector(m_width-1 downto 0);
81            i_8 : in  std_logic_vector(m_width-1 downto 0);
              i_9 : in  std_logic_vector(m_width-1 downto 0);
83            i_10 : in  std_logic_vector(m_width-1 downto 0);
              i_11 : in  std_logic_vector(m_width-1 downto 0);
85            i_12 : in  std_logic_vector(m_width-1 downto 0);
              i_13 : in  std_logic_vector(m_width-1 downto 0);
87            i_14 : in  std_logic_vector(m_width-1 downto 0);
              i_15 : in  std_logic_vector(m_width-1 downto 0);
89            i_S : in  std_logic_vector(3 downto 0);
              o_Z : out std_logic_vector(m_width-1 downto 0));
91    end component;

93    component myMUX4_N is
        generic (m_width : integer := 32);
95      port (i_0 : in  std_logic_vector(m_width-1 downto 0);
              i_1 : in  std_logic_vector(m_width-1 downto 0);
97            i_2 : in  std_logic_vector(m_width-1 downto 0);
              i_3 : in  std_logic_vector(m_width-1 downto 0);
99            i_S : in  std_logic_vector(1 downto 0);
              o_Z : out std_logic_vector(m_width-1 downto 0));
101   end component;

103   component myMUX2_N is
        generic (m_width : integer := 32);
105     port (i_0 : in  std_logic_vector(m_width-1 downto 0);
              i_1 : in  std_logic_vector(m_width-1 downto 0);
107           i_S : in  std_logic;
              o_Z : out std_logic_vector(m_width-1 downto 0));
109   end component;

111   component myMUX2_1 is
        port (i_0 : in  std_logic;
113           i_1 : in  std_logic;
              i_S : in  std_logic;
115           o_Z : out std_logic);
```

```
  end component;

  component Sign_Extend16_32 is
    port (i_A : in  std_logic_vector(15 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

  component Extend16_32 is
    port (i_A : in  std_logic_vector(15 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

  component Extend5_32 is
    port (i_A : in  std_logic_vector(4 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

  component SLL2_32 is
    port (i_A : in  std_logic_vector(31 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

  component myReg1 is
    port (i_clk   : in  std_logic;
          i_reset : in  std_logic;
          i_D     : in  std_logic;
          o_Q     : out std_logic);
  end component;

  component CARRY_SELECT_ADDER_32 is
    port (i_A      : in  std_logic_vector(31 downto 0);
          i_A_n    : in  std_logic_vector(31 downto 0);
          i_B      : in  std_logic_vector(31 downto 0);
          i_B_n    : in  std_logic_vector(31 downto 0);
          i_AB_AND    : in  std_logic_vector(31 downto 0);
          i_AB_NAND : in  std_logic_vector(31 downto 0);
          i_AB_NOR    : in  std_logic_vector(31 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(31 downto 0);
          o_C      : out std_logic);
  end component;

  component LEFT_SHIFT is
```

```
159     port (i_A : in  std_logic_vector(31 downto 0);
            i_SA  : in  std_logic_vector(4 downto 0);
161         o_Z : out std_logic_vector(31 downto 0)
            );
163   end component;

165   component RIGHT_SHIFT is
        port (i_A    : in  std_logic_vector(31 downto 0);
167         i_SA     : in  std_logic_vector(4 downto 0);
            i_ARITH : in std_logic;
169         o_Z    : out std_logic_vector(31 downto 0)
            );
171   end component;

173   component Bitwise_Ops2_32 is
        port (i_A    : in  std_logic_vector(31 downto 0);
175         i_B    : in  std_logic_vector(31 downto 0);
            o_A_n    : out std_logic_vector(31 downto 0);
177         o_B_n    : out std_logic_vector(31 downto 0);
            o_AND    : out std_logic_vector(31 downto 0);
179         o_NAND   : out std_logic_vector(31 downto 0);
            o_OR     : out std_logic_vector(31 downto 0);
181         o_XOR    : out std_logic_vector(31 downto 0);
            o_XNOR   : out std_logic_vector(31 downto 0);
183         o_NOR    : out std_logic_vector(31 downto 0));
    end component;
185
    component Comparator is
187   port (i_S      : in  std_logic_vector(31 downto 0);
          i_NAND_AB : in  std_logic;
189       i_OR_AB   : in  std_logic;
          i_XNOR_AB : in  std_logic;
191       i_unsigned  : in  std_logic;
          i_overflow  : in  std_logic;
193       i_COMP_SEL  : in  std_logic_vector(2 downto 0);
          o_Z      : out std_logic);
195 end component;

197 component LUI is
      port (i_A : in  std_logic_vector(15 downto 0);
199       o_Z : out std_logic_vector(31 downto 0)
          );
201 end component;
```

```
203   component Over_Detect is
        port (i_AB_NAND : in  std_logic; -- From bitwise
      operations on bit 31 - NAND(A,B)
205         i_AB_OR   : in  std_logic; -- From bitwise operations
       on bit 31 - OR(A,B)
            i_AB_XNOR : in  std_logic; -- From bitwise operations
       on bit 31 - XNOR(A,B)
207         i_S       : in  std_logic_vector(31 downto 0);
            i_RD      : in  std_logic_vector(31 downto 0);
209         i_RT      : in  std_logic_vector(31 downto 0);
            i_OVER_CTRL : in  std_logic_vector(1 downto 0);
211         o_Z       : out std_logic_vector(31 downto 0)
            );
213   end component;

215 --| Define Signals
    signal w_A          : std_logic_vector(31 downto 0); -- ALU
      Input A
217   signal w_A_n        : std_logic_vector(31 downto 0); --
      Inverted ALU Input A
    signal w_B          : std_logic_vector(31 downto 0); -- ALU
      Input B
219   signal w_B_n        : std_logic_vector(31 downto 0); --
      Inverted ALU Input B
    signal w_B0         : std_logic_vector(31 downto 0); --
      Preliminary ALU Input - B may become B0 or INV(B0)
      depending on the value of i_ALU_INV_B
221   signal w_B0_n       : std_logic_vector(31 downto 0); -- INV(
      B0)
    signal w_AB_NAND    : std_logic_vector(31 downto 0); --
      NAND(A,B)
223   signal w_SA_EXT     : std_logic_vector(31 downto 0); -- Zero
      extended i_SA
    signal w_IMM_EXT0   : std_logic_vector(31 downto 0); --
      Zero extended immediate
225   signal w_IMM_EXT1   : std_logic_vector(31 downto 0); --
      Sign extended immediate
    signal w_IMM_EXT    : std_logic_vector(31 downto 0); --
      Zero or Sign Extended immediate
227   signal w_IMM_SLL2   : std_logic_vector(31 downto 0); --
      Sign extended immediate left shifted by 2
```

```
    signal w_comp_out    : std_logic;                -- Output of
     the comparator
229 signal f_comp_out    : std_logic;                -- Output of
     the comparator delayed one clock cycle
    signal ff_comp_out   : std_logic;                -- Output of
     the comparator delayed two clock cycles
231 signal w_SLX       : std_logic_vector(31 downto 0); --
     Output of the LEFT_SHIFT
    signal w_SRX       : std_logic_vector(31 downto 0); --
     Output of the RIGHT_SHIFT
233 signal w_ADD0      : std_logic_vector(31 downto 0); --
     Output of the adder
    signal w_ADD       : std_logic_vector(31 downto 0); --
     Output of the adder after overflow detection
235 signal w_AND       : std_logic_vector(31 downto 0); --
     Output of the bitwise AND
    signal w_OR        : std_logic_vector(31 downto 0); --
     Output of the bitwsie OR
237 signal w_XOR       : std_logic_vector(31 downto 0); --
     Output of the bitwise XOR
    signal w_XNOR      : std_logic_vector(31 downto 0); --
     Output of the bitwise XNOR - inverted XOR signal
239 signal w_NOR       : std_logic_vector(31 downto 0); --
     Output of the bitwise NOR
    signal w_COMP      : std_logic_vector(31 downto 0); --
     Result of a logical comparison (k_zero32 for false and
     k_one32 for true).
241 signal w_LUI       : std_logic_vector(31 downto 0); --
     Output of the bitwise LUI
    signal w_ALU_SRC_B_3 : std_logic_vector(31 downto 0); --
     Input 3 to the ALU_SRC_B selection MUX
243
--| Define Constants
245 constant k_zero32 : std_logic_vector(31 downto 0) := (
     others => '0');
    constant k_one32  : std_logic_vector(31 downto 0) := B"0000
     _0000_0000_0000_0000_0000_0000_0001";
247 constant k_four32 : std_logic_vector(31 downto 0) := B"0000
     _0000_0000_0000_0000_0000_0000_0100";

249 begin
  -- Extend the SA value
251 u_Extend_SA: Extend5_32
```

```
    port map(i_A => i_imm(10 downto 6), --i_SA
253         o_Z => w_SA_EXT);

255 -- Zero extend the immediate value
    u_Extend_IMM: Extend16_32
257 port map(i_A => i_imm,
            o_Z => w_IMM_EXT0);
259
    -- Sign extend the immediate value
261 u_Sign_Extend_IMM: Sign_Extend16_32
    port map(i_A => i_imm,
263         o_Z => w_IMM_EXT1);

265 -- Determine the input to ALU_SRC_B selection MUX input 3 (
    B"11")
    u_myMUX2_IMM_EXTEND: myMUX2_N
267 generic map (m_width => 32)
    port map (i_0 => w_IMM_EXT0,
269           i_1 => w_IMM_EXT1,
              i_S => i_imm_extend,
271           o_Z => w_IMM_EXT);

273 -- Shift the sign extended immediate value left by 2
    u_SLL2_IMM: SLL2_32
275 port map(i_A => w_IMM_EXT1,
            o_Z => w_IMM_SLL2);
277
    -- Determine the input to ALU_SRC_B selection MUX input 3 (
    B"11")
279 u_myMUX2_ALU_SRC_B_3: myMUX2_N
    generic map (m_width => 32)
281 port map (i_0 => k_four32,
              i_1 => w_IMM_SLL2,
283           i_S => ff_comp_out,
              o_Z => w_ALU_SRC_B_3);
285
    -- Determine the A input to the ALU
287 u_ALU_SRC_A: myMUX4_N
    generic map (m_width => 32)
289 port map (i_0 => i_RS_DATA,
              i_1 => w_SA_EXT,
291           i_2 => i_PC,
              i_3 => i_PC,
```

```
293             i_S => i_ALU_SRC_A,
                o_Z => w_A);
295
    -- Determine the B0 input to the ALU
297 u_ALU_SRC_B: myMUX4_N
    generic map (m_width => 32)
299 port map (i_0 => k_zero32,
                i_1 => i_RT_DATA,
301             i_2 => w_IMM_EXT,
                i_3 => w_ALU_SRC_B_3,
303             i_S => i_ALU_SRC_B,
                o_Z => w_B0);
305
    -- Create inverted B0
307 u_myINV_B0: myINV_N
    generic map (m_width => 32)
309 port map (i_A => w_B0,
                o_Z => w_B0_n);
311
    -- Determine whether to use B0 or B0_n as the B input
313 u_myMUX2_B: myMUX2_N
    generic map (m_width => 32)
315 port map (i_0 => w_B0,
                i_1 => w_B0_n,
317             i_S => i_ALU_INV_B,
                o_Z => w_B);
319
    -- Connect the LEFT_SHIFT Module
321 u_LEFT_SHIFT: LEFT_SHIFT
    port map (i_A => w_B,
323             i_SA => w_A(4 downto 0),
                o_Z => w_SLX);
325
    -- Connect the RIGHT_SHIFT Module
327 u_RIGHT_SHIFT: RIGHT_SHIFT
    port map (i_A => w_B,
329             i_SA => w_A(4 downto 0),
                i_ARITH => i_imm(0), --i_instr0
331             o_Z => w_SRX);

333 -- Connect the Adder Module
    u_CARRY_SELECT_ADDER_32: CARRY_SELECT_ADDER_32
335 port map (i_A => w_A,
```

```
            i_A_n => w_A_n,
            i_B => w_B,
            i_B_n => w_B_n,
            i_AB_AND => w_AND,
            i_AB_NAND => w_AB_NAND,
            i_AB_NOR => w_NOR,
            i_C => i_ALU_INV_B,
            o_S => w_ADD0,
            o_C => OPEN);

    -- Connect the Bitwise Operations Module
    u_Bitwise_Ops2_32: Bitwise_Ops2_32
    port map (i_A => w_A,
            i_B => w_B,
            o_A_n => w_A_n,
            o_B_n => w_B_n,
            o_AND => w_AND,
            o_NAND => w_AB_NAND,
            o_OR => w_OR,
            o_XOR => w_XOR,
            o_XNOR => w_XNOR,
            o_NOR => w_NOR);

    -- Connect Load Upper Immediate Module
    u_LUI: LUI
    port map (i_A => w_B(15 downto 0),
            o_Z => w_LUI);

    -- Connect Comparator
    u_Comparator: Comparator
    port map (i_S => w_ADD0,
            i_NAND_AB => w_AB_NAND(31),
            i_OR_AB => w_OR(31),
            i_XNOR_AB => w_XNOR(31),
            i_unsigned => i_UNSIGNED,
            i_overflow => i_overflow,
            i_COMP_SEL => i_COMP_SEL,
            o_Z => w_comp_out);

    -- Connect Register to store value of comparator
    u_myReg_delay_1: myReg1
    port map (i_clk => i_clk,
            i_reset => i_reset,
```

```
379             i_D => w_comp_out,
                o_Q => f_comp_out);
381
    u_myReg_delay_2: myReg1
383 port map (i_clk => i_clk,
                i_reset => i_reset,
385             i_D => f_comp_out,
                o_Q => ff_comp_out);
387
    -- Determine Output for a Set On Less Than Instruction
389 u_myMUX2_COMP: myMUX2_N
    generic map (m_width => 32)
391 port map (i_0 => k_zero32,
                i_1 => k_one32,
393             i_S => w_comp_out,
                o_Z => w_COMP);
395
    -- Connect Overflow Detection
397 u_Over_Detect: Over_Detect
    port map(i_AB_NAND => w_AB_NAND(31),
399         i_AB_OR => w_OR(31),
            i_AB_XNOR => w_XNOR(31),
401         i_S => w_ADD0,
            i_RD => i_RD_DATA,
403         i_RT => i_RT_DATA,
            i_OVER_CTRL => i_OVER_CTRL,
405         o_Z => w_ADD);

407 -- Determine ALU Output
    u_myMUX16_ALU_OUT: myMUX16_N
409 generic map (m_width => 32)
    port map (i_0 => w_SLX,
411             i_1 => w_SRX,
                i_2 => w_ADD,
413             i_3 => w_AND,
                i_4 => w_OR,
415             i_5 => w_XOR,
                i_6 => w_NOR,
417             i_7 => w_COMP,
                i_8 => w_LUI,
419             i_9 => w_SRX,  -- MUX inputs 9-15 were chosen to
     minimize the error should a bit flip occur
```

```
        i_10 => w_ADD, -- in the MSB position of the
    ALU_OUTPUT control signal.  For i_9 "1001" is
        i_11 => w_AND, -- w_SRX which is also conneted to
    i_1 "0001".For i_14 "1110" is w_NOR which
        i_12 => w_OR,  -- is also connected to i_6 "0110".
        i_13 => w_XOR,
        i_14 => w_NOR,
        i_15 => w_COMP,
        i_S => i_ALU_OUTPUT,
        o_Z => o_ALU_RESULT);

end a_ALU_CORE;
```

Listing A.36. ALU_CORE.vhd Code

1.15 5-Bit to 32-Bit Decoder

```
1  --| Decoder5_32.vhd
   --| Decodes a 5-bit input signal and activates one of 32
       outputs.  For example:
3  --|    B"00000" => Z(0) is 1 and other 31 outputs are 0
   --|    B"10100" => Z(20) is 1 and other 31 outputs are 0
5  --|
   --| INPUTS:
7  --| i_S - Encoded input signal
   --|
9  --| OUTPUTS:
   --| o_Z - Decoded output signal
11 library IEEE;
   use IEEE.std_logic_1164.all;
13
   entity Decoder5_32 is
15   port (i_S : in  std_logic_vector(4 downto 0);
         o_Z : out std_logic_vector(31 downto 0));
17 end Decoder5_32;

19 architecture a_Decoder5_32 of Decoder5_32 is
   --| Declare Components
21   -- Declare NAND2
     component myNAND2 is
23     port (i_A : in  std_logic;
           i_B : in  std_logic;
25         o_Z : out std_logic
           );
27   end component;
     -- Declare Inverter
29   component myINV is
       port (i_A : in  std_logic;
31         o_Z : out std_logic
           );
33   end component;

35   component myINV_N is
       generic(m_width : integer := 32);  -- Number of bits in
       the inputs and output
37     port (i_A : in  std_logic_vector(m_width-1 downto 0);
           o_Z : out std_logic_vector(m_width-1 downto 0));
39   end component;
```

```
41   --| Declare Signals
     -- Inverted Input Signals
43   signal w_S_n : std_logic_vector(4 downto 0);

45   -- Intermediate decoder signals
     signal w_dec_00, w_dec_00_n : std_logic;
47   signal w_dec_01, w_dec_01_n : std_logic;
     signal w_dec_02, w_dec_02_n : std_logic;
49   signal w_dec_03, w_dec_03_n : std_logic;
     signal w_dec_04, w_dec_04_n : std_logic;
51   signal w_dec_05, w_dec_05_n : std_logic;
     signal w_dec_06, w_dec_06_n : std_logic;
53   signal w_dec_07, w_dec_07_n : std_logic;
     signal w_dec_08, w_dec_08_n : std_logic;
55   signal w_dec_09, w_dec_09_n : std_logic;
     signal w_dec_10, w_dec_10_n : std_logic;
57   signal w_dec_11, w_dec_11_n : std_logic;
     signal w_dec_12, w_dec_12_n : std_logic;
59   signal w_dec_13, w_dec_13_n : std_logic;
     signal w_dec_14, w_dec_14_n : std_logic;
61   signal w_dec_15, w_dec_15_n : std_logic;
     signal w_dec_16, w_dec_16_n : std_logic;
63   signal w_dec_17, w_dec_17_n : std_logic;
     signal w_dec_18, w_dec_18_n : std_logic;
65   signal w_dec_19, w_dec_19_n : std_logic;
     signal w_dec_20, w_dec_20_n : std_logic;
67   signal w_dec_21, w_dec_21_n : std_logic;
     signal w_dec_22, w_dec_22_n : std_logic;
69   signal w_dec_23, w_dec_23_n : std_logic;

71   -- Inverted Output Signal
     signal w_Z_n : std_logic_vector(31 downto 0);
73
   begin
75   --| Create inverted input signals
     u_myINV_S : myINV_N
77   generic map (m_width => 5)
     port map (
79     i_A => i_S,
       o_Z => w_S_n);
81
     --| Invert intermediate decoder signals
```

```
83    u_myINV_00 : myINV
      port map (
85      i_A => w_dec_00,
        o_Z => w_dec_00_n);
87
      u_myINV_01 : myINV
89    port map (
        i_A => w_dec_01,
91      o_Z => w_dec_01_n);

93    u_myINV_02 : myINV
      port map (
95      i_A => w_dec_02,
        o_Z => w_dec_02_n);
97
      u_myINV_03 : myINV
99    port map (
        i_A => w_dec_03,
101     o_Z => w_dec_03_n);

103   u_myINV_04 : myINV
      port map (
105     i_A => w_dec_04,
        o_Z => w_dec_04_n);
107
      u_myINV_05 : myINV
109   port map (
        i_A => w_dec_05,
111     o_Z => w_dec_05_n);

113   u_myINV_06 : myINV
      port map (
115     i_A => w_dec_06,
        o_Z => w_dec_06_n);
117
      u_myINV_07 : myINV
119   port map (
        i_A => w_dec_07,
121     o_Z => w_dec_07_n);

123   u_myINV_08 : myINV
      port map (
125     i_A => w_dec_08,
```

```
    o_Z => w_dec_08_n);

  u_myINV_09 : myINV
  port map (
    i_A => w_dec_09,
    o_Z => w_dec_09_n);

  u_myINV_10 : myINV
  port map (
    i_A => w_dec_10,
    o_Z => w_dec_10_n);

  u_myINV_11 : myINV
  port map (
    i_A => w_dec_11,
    o_Z => w_dec_11_n);

  u_myINV_12 : myINV
  port map (
    i_A => w_dec_12,
    o_Z => w_dec_12_n);

  u_myINV_13 : myINV
  port map (
    i_A => w_dec_13,
    o_Z => w_dec_13_n);

  u_myINV_14 : myINV
  port map (
    i_A => w_dec_14,
    o_Z => w_dec_14_n);

  u_myINV_15 : myINV
  port map (
    i_A => w_dec_15,
    o_Z => w_dec_15_n);

  u_myINV_16 : myINV
  port map (
    i_A => w_dec_16,
    o_Z => w_dec_16_n);

  u_myINV_17 : myINV
```

```
169     port map (
          i_A => w_dec_17,
171       o_Z => w_dec_17_n);

173     u_myINV_18 : myINV
        port map (
175       i_A => w_dec_18,
          o_Z => w_dec_18_n);
177
        u_myINV_19 : myINV
179     port map (
          i_A => w_dec_19,
181       o_Z => w_dec_19_n);

183     u_myINV_20 : myINV
        port map (
185       i_A => w_dec_20,
          o_Z => w_dec_20_n);
187
        u_myINV_21 : myINV
189     port map (
          i_A => w_dec_21,
191       o_Z => w_dec_21_n);

193     u_myINV_22 : myINV
        port map (
195       i_A => w_dec_22,
          o_Z => w_dec_22_n);
197
        u_myINV_23 : myINV
199     port map (
          i_A => w_dec_23,
201       o_Z => w_dec_23_n);

203     --| Instantiate NAND gates to create intermediate control
         signals
        u_myNAND_00 : myNAND2
205     port map (
          i_A => w_S_n(4),
207       i_B => w_S_n(3),
          o_Z => w_dec_00);
209
        u_myNAND_01 : myNAND2
```

```
211   port map (
        i_A => w_S_n(2),
213     i_B => w_S_n(1),
        o_Z => w_dec_01);
215
      u_myNAND_02 : myNAND2
217   port map (
        i_A => w_dec_00_n,
219     i_B => w_dec_01_n,
        o_Z => w_dec_02);
221
      u_myNAND_03 : myNAND2
223   port map (
        i_A => w_S_n(2),
225     i_B => i_S(1),
        o_Z => w_dec_03);
227
      u_myNAND_04 : myNAND2
229   port map (
        i_A => w_dec_00_n,
231     i_B => w_dec_03_n,
        o_Z => w_dec_04);
233
      u_myNAND_05 : myNAND2
235   port map (
        i_A => i_S(2),
237     i_B => w_S_n(1),
        o_Z => w_dec_05);
239
      u_myNAND_06 : myNAND2
241   port map (
        i_A => w_dec_00_n,
243     i_B => w_dec_05_n,
        o_Z => w_dec_06);
245
      u_myNAND_07 : myNAND2
247   port map (
        i_A => i_S(2),
249     i_B => i_S(1),
        o_Z => w_dec_07);
251
      u_myNAND_08 : myNAND2
253   port map (
```

```
    i_A => w_dec_00_n,
    i_B => w_dec_07_n,
    o_Z => w_dec_08);

u_myNAND_09 : myNAND2
port map (
    i_A => w_S_n(4),
    i_B => i_S(3),
    o_Z => w_dec_09);

u_myNAND_10 : myNAND2
port map (
    i_A => w_dec_09_n,
    i_B => w_dec_01_n,
    o_Z => w_dec_10);

u_myNAND_11 : myNAND2
port map (
    i_A => w_dec_09_n,
    i_B => w_dec_03_n,
    o_Z => w_dec_11);

u_myNAND_12 : myNAND2
port map (
    i_A => w_dec_09_n,
    i_B => w_dec_05_n,
    o_Z => w_dec_12);

u_myNAND_13 : myNAND2
port map (
    i_A => w_dec_09_n,
    i_B => w_dec_07_n,
    o_Z => w_dec_13);

u_myNAND_14 : myNAND2
port map (
    i_A => i_S(4),
    i_B => w_S_n(3),
    o_Z => w_dec_14);

u_myNAND_15 : myNAND2
port map (
    i_A => w_dec_14_n,
```

```
297       i_B => w_dec_01_n,
          o_Z => w_dec_15);
299
      u_myNAND_16 : myNAND2
301   port map (
          i_A => w_dec_14_n,
303       i_B => w_dec_03_n,
          o_Z => w_dec_16);
305
      u_myNAND_17 : myNAND2
307   port map (
          i_A => w_dec_14_n,
309       i_B => w_dec_05_n,
          o_Z => w_dec_17);
311
      u_myNAND_18 : myNAND2
313   port map (
          i_A => w_dec_14_n,
315       i_B => w_dec_07_n,
          o_Z => w_dec_18);
317
      u_myNAND_19 : myNAND2
319   port map (
          i_A => i_S(4),
321       i_B => i_S(3),
          o_Z => w_dec_19);
323
      u_myNAND_20 : myNAND2
325   port map (
          i_A => w_dec_19_n,
327       i_B => w_dec_01_n,
          o_Z => w_dec_20);
329
      u_myNAND_21 : myNAND2
331   port map (
          i_A => w_dec_19_n,
333       i_B => w_dec_03_n,
          o_Z => w_dec_21);
335
      u_myNAND_22 : myNAND2
337   port map (
          i_A => w_dec_19_n,
339       i_B => w_dec_05_n,
```

```
    o_Z => w_dec_22);

  u_myNAND_23 : myNAND2
  port map (
    i_A => w_dec_19_n,
    i_B => w_dec_07_n,
    o_Z => w_dec_23);

  --| Instantiate Inverters and NAND gates to generate output
    signals
  u_myNAND_Z0 : myNAND2
  port map (
    i_A => w_dec_02_n,
    i_B => w_S_n(0),
    o_Z => w_Z_n(0));

  u_myNAND_Z1 : myNAND2
  port map (
    i_A => w_dec_02_n,
    i_B => i_S(0),
    o_Z => w_Z_n(1));

  u_myNAND_Z2 : myNAND2
  port map (
    i_A => w_dec_04_n,
    i_B => w_S_n(0),
    o_Z => w_Z_n(2));

  u_myNAND_Z3 : myNAND2
  port map (
    i_A => w_dec_04_n,
    i_B => i_S(0),
    o_Z => w_Z_n(3));

  u_myNAND_Z4 : myNAND2
  port map (
    i_A => w_dec_06_n,
    i_B => w_S_n(0),
    o_Z => w_Z_n(4));

  u_myNAND_Z5 : myNAND2
  port map (
    i_A => w_dec_06_n,
```

```
    i_B => i_S(0),
383 o_Z => w_Z_n(5));

385 u_myNAND_Z6 : myNAND2
    port map (
387   i_A => w_dec_08_n,
      i_B => w_S_n(0),
389   o_Z => w_Z_n(6));

391 u_myNAND_Z7 : myNAND2
    port map (
393   i_A => w_dec_08_n,
      i_B => i_S(0),
395   o_Z => w_Z_n(7));

397 u_myNAND_Z8 : myNAND2
    port map (
399   i_A => w_dec_10_n,
      i_B => w_S_n(0),
401   o_Z => w_Z_n(8));

403 u_myNAND_Z9 : myNAND2
    port map (
405   i_A => w_dec_10_n,
      i_B => i_S(0),
407   o_Z => w_Z_n(9));

409 u_myNAND_Z10 : myNAND2
    port map (
411   i_A => w_dec_11_n,
      i_B => w_S_n(0),
413   o_Z => w_Z_n(10));

415 u_myNAND_Z11 : myNAND2
    port map (
417   i_A => w_dec_11_n,
      i_B => i_S(0),
419   o_Z => w_Z_n(11));

421 u_myNAND_Z12 : myNAND2
    port map (
423   i_A => w_dec_12_n,
      i_B => w_S_n(0),
```

```
425     o_Z => w_Z_n(12));

427   u_myNAND_Z13 : myNAND2
      port map (
429     i_A => w_dec_12_n,
        i_B => i_S(0),
431     o_Z => w_Z_n(13));

433   u_myNAND_Z14 : myNAND2
      port map (
435     i_A => w_dec_13_n,
        i_B => w_S_n(0),
437     o_Z => w_Z_n(14));

439   u_myNAND_Z15 : myNAND2
      port map (
441     i_A => w_dec_13_n,
        i_B => i_S(0),
443     o_Z => w_Z_n(15));

445   u_myNAND_Z16 : myNAND2
      port map (
447     i_A => w_dec_15_n,
        i_B => w_S_n(0),
449     o_Z => w_Z_n(16));

451   u_myNAND_Z17 : myNAND2
      port map (
453     i_A => w_dec_15_n,
        i_B => i_S(0),
455     o_Z => w_Z_n(17));

457   u_myNAND_Z18 : myNAND2
      port map (
459     i_A => w_dec_16_n,
        i_B => w_S_n(0),
461     o_Z => w_Z_n(18));

463   u_myNAND_Z19 : myNAND2
      port map (
465     i_A => w_dec_16_n,
        i_B => i_S(0),
467     o_Z => w_Z_n(19));
```

```
469   u_myNAND_Z20 : myNAND2
      port map (
471     i_A => w_dec_17_n,
        i_B => w_S_n(0),
473     o_Z => w_Z_n(20));

475   u_myNAND_Z21 : myNAND2
      port map (
477     i_A => w_dec_17_n,
        i_B => i_S(0),
479     o_Z => w_Z_n(21));

481   u_myNAND_Z22 : myNAND2
      port map (
483     i_A => w_dec_18_n,
        i_B => w_S_n(0),
485     o_Z => w_Z_n(22));

487   u_myNAND_Z23 : myNAND2
      port map (
489     i_A => w_dec_18_n,
        i_B => i_S(0),
491     o_Z => w_Z_n(23));

493   u_myNAND_Z24 : myNAND2
      port map (
495     i_A => w_dec_20_n,
        i_B => w_S_n(0),
497     o_Z => w_Z_n(24));

499   u_myNAND_Z25 : myNAND2
      port map (
501     i_A => w_dec_20_n,
        i_B => i_S(0),
503     o_Z => w_Z_n(25));

505   u_myNAND_Z26 : myNAND2
      port map (
507     i_A => w_dec_21_n,
        i_B => w_S_n(0),
509     o_Z => w_Z_n(26));
```

```
511   u_myNAND_Z27 : myNAND2
      port map (
513     i_A => w_dec_21_n,
        i_B => i_S(0),
515     o_Z => w_Z_n(27));

517   u_myNAND_Z28 : myNAND2
      port map (
519     i_A => w_dec_22_n,
        i_B => w_S_n(0),
521     o_Z => w_Z_n(28));

523   u_myNAND_Z29 : myNAND2
      port map (
525     i_A => w_dec_22_n,
        i_B => i_S(0),
527     o_Z => w_Z_n(29));

529   u_myNAND_Z30 : myNAND2
      port map (
531     i_A => w_dec_23_n,
        i_B => w_S_n(0),
533     o_Z => w_Z_n(30));

535   u_myNAND_Z31 : myNAND2
      port map (
537     i_A => w_dec_23_n,
        i_B => i_S(0),
539     o_Z => w_Z_n(31));

541   -- Create output signal
      u_myINV_Z : myINV_N
543   generic map (m_width => 32)
      port map (
545     i_A => w_Z_n,
        o_Z => o_Z);
547 end a_Decoder5_32;
```

Listing A.37. Decoder5_32.vhd Code

1.16 N-Bit Register

```
--| myRegN.vhd
--| N-bit register
--|
--| INPUTS:
--| i_clk - Clock
--| i_reset - Reset
--| i_D   - Data
--|
--| OUTPUTS:
--| o_Q - Output
library IEEE;
use IEEE.std_logic_1164.all;

entity myRegN is
  generic(m_width : integer := 32);  -- Number of bits in the
    inputs and output
  port (i_clk   : in  std_logic;
        i_reset : in  std_logic;
        i_D     : in  std_logic_vector(m_width-1 downto 0);
        o_Q     : out std_logic_vector(m_width-1 downto 0));
end myRegN;

architecture a_myRegN of myRegN is
--| Declare Components
  component myReg1 is
    port (i_clk   : in  std_logic;
          i_reset : in  std_logic;
          i_D     : in  std_logic;
          o_Q     : out std_logic);
  end component;

begin
  reg_array: for bit_index in 0 to m_width-1 generate
    u_myReg1 : myReg1
    port map (i_clk => i_clk,
              i_reset => i_reset,
              i_D => i_D(bit_index),
              o_Q => o_Q(bit_index));
  end generate reg_array;
end a_myRegN;
```

Listing A.38. myRegN.vhd Code

1.17 Enabled Register

```
1  --| Enabled_Register.vhd
   --| On the rising clock edge, the contents of the register
     are modified
3  --|to be equal to the value of i_data if i_en is 1.
     Otherwise, the
   --| register retains its previous value.
5  --|
   --| INPUTS:
7  --| i_clk - Clock
   --| i_reset - Reset
9  --| i_data  - Data
   --| i_en    - Enable the register to store incoming data
11 --|
   --| OUTPUTS:
13 --| o_Q - Output of register 0
   library IEEE;
15 use IEEE.std_logic_1164.all;

17 entity Enabled_Register is
     port (i_clk   : in  std_logic;
19       i_reset : in  std_logic;
         i_data  : in  std_logic_vector(31 downto 0);
21       i_en    : in  std_logic;
         o_Q     : out std_logic_vector(31 downto 0));
23 end Enabled_Register;

25 architecture a_Enabled_Register of Enabled_Register is
   --| Declare Components
27   component myRegN is
       generic (m_width : integer := 32);
29     port (i_clk   : in  std_logic;
           i_reset : in  std_logic;
31         i_D     : in  std_logic_vector(m_width-1 downto 0);
           o_Q     : out std_logic_vector(m_width-1 downto 0));
33   end component;

35   component myMUX2_N is
       generic (m_width : integer := 32);
37     port (i_0 : in  std_logic_vector(m_width-1 downto 0);
           i_1 : in  std_logic_vector(m_width-1 downto 0);
39         i_S : in  std_logic;
```

```
        o_Z : out std_logic_vector(m_width-1 downto 0)
        );
  end component;

--| Declare Signals
  -- Register Input
  signal w_D : std_logic_vector(31 downto 0);
  -- Register Output
  signal f_Q : std_logic_vector(31 downto 0);

  signal w_EN : std_logic_vector(31 downto 0); -- Enable
   Signal
begin
  u_myMUX2: myMUX2_N
  generic map (m_width => 32)
  port map (i_0 => f_Q,
            i_1 => i_data,
            i_S => i_en,
            o_Z => w_D);
  --| Store the selected value to the register
  u_myReg: myRegN
  generic map (m_width => 32)
  port map (i_clk => i_clk,
            i_reset => i_reset,
            i_D => w_D,
            o_Q => f_Q);

  --| Connect output signals
  o_Q <= f_Q;
end a_Enabled_Register;
```

Listing A.39. Enabled_Register.vhd Code

1.18 GPR Bank

```
--| GPR_Bank.vhd
--| Bank of 32, 32-bit registers. On the rising clock edge, only the contents of
--| the selected register are modified to be equal to the value of i_data. All
--| other registers retain their previous value.
--|
--| INPUTS:
--| i_clk - Clock
--| i_reset - Reset
--| i_data - Data
--| i_sel - Encoded select signal to determine which register should be enabled
--|
--| OUTPUTS:
--| o_Q00 - Output of register 0
--| o_Q01 - Output of register 1
--| ...
--| o_Q30 - Output of register 30
--| o_Q31 - Output of register 31
library IEEE;
use IEEE.std_logic_1164.all;
use work.my_word_array_package.all;

entity GPR_Bank is
  port (i_clk   : in  std_logic;
        i_reset : in  std_logic;
        i_data  : in  std_logic_vector(31 downto 0);
        i_sel   : in  std_logic_vector(4 downto 0);
        o_Q     : out word_array);
--      o_Q00   : out std_logic_vector(31 downto 0);
--      o_Q01   : out std_logic_vector(31 downto 0);
--      o_Q02   : out std_logic_vector(31 downto 0);
--      o_Q03   : out std_logic_vector(31 downto 0);
--      o_Q04   : out std_logic_vector(31 downto 0);
--      o_Q05   : out std_logic_vector(31 downto 0);
--      o_Q06   : out std_logic_vector(31 downto 0);
--      o_Q07   : out std_logic_vector(31 downto 0);
--      o_Q08   : out std_logic_vector(31 downto 0);
--      o_Q09   : out std_logic_vector(31 downto 0);
--      o_Q10   : out std_logic_vector(31 downto 0);
```

```
--        o_Q11    : out std_logic_vector(31 downto 0);
--        o_Q12    : out std_logic_vector(31 downto 0);
--        o_Q13    : out std_logic_vector(31 downto 0);
--        o_Q14    : out std_logic_vector(31 downto 0);
--        o_Q15    : out std_logic_vector(31 downto 0);
--        o_Q16    : out std_logic_vector(31 downto 0);
--        o_Q17    : out std_logic_vector(31 downto 0);
--        o_Q18    : out std_logic_vector(31 downto 0);
--        o_Q19    : out std_logic_vector(31 downto 0);
--        o_Q20    : out std_logic_vector(31 downto 0);
--        o_Q21    : out std_logic_vector(31 downto 0);
--        o_Q22    : out std_logic_vector(31 downto 0);
--        o_Q23    : out std_logic_vector(31 downto 0);
--        o_Q24    : out std_logic_vector(31 downto 0);
--        o_Q25    : out std_logic_vector(31 downto 0);
--        o_Q26    : out std_logic_vector(31 downto 0);
--        o_Q27    : out std_logic_vector(31 downto 0);
--        o_Q28    : out std_logic_vector(31 downto 0);
--        o_Q29    : out std_logic_vector(31 downto 0);
--        o_Q30    : out std_logic_vector(31 downto 0);
--        o_Q31    : out std_logic_vector(31 downto 0));
end GPR_Bank;

architecture a_GPR_Bank of GPR_Bank is
----| Declare Type for a 32 entry array of 32-bit
    std_logic_vectors.
--  type word_array is array (31 downto 0) of
    std_logic_vector(31 downto 0);

--| Declare Components
  component myRegN is
    generic (m_width : integer := 32);
    port (i_clk    : in  std_logic;
          i_reset  : in  std_logic;
          i_D      : in  std_logic_vector(31 downto 0);
          o_Q      : out std_logic_vector(31 downto 0));
  end component;

  component Decoder5_32 is
    port (i_S : in  std_logic_vector(4 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;
```

```
80
   component myMUX2_N is
82   generic (m_width : integer := 32);
     port (i_0 : in  std_logic_vector(31 downto 0);
84       i_1 : in  std_logic_vector(31 downto 0);
         i_S : in  std_logic;
86       o_Z : out std_logic_vector(31 downto 0)
         );
88 end component;

90 --| Declare Signals

92 -- Input To Each Register
   signal w_D : word_array;
94 -- Output From Each Register
   signal f_Q : word_array;
96
   signal w_EN : std_logic_vector(31 downto 0); -- Enable
    Signal
98 begin
   --| Enable Register Signal
100 u_Decoder5_32: Decoder5_32
   port map (i_S => i_sel,
102        o_Z => w_EN);

104 --| Create the array of registers (excluding the 0 register
   reg_array: for bit_index in 1 to 31 generate
106  -- For each register, determine whether the the register
    input
     -- should be its previous value or i_data.
108  u_myMUX2_N: myMUX2_N
     generic map (m_width => 32)
110  port map (i_0 => f_Q(bit_index),
            i_1 => i_data,
112         i_S => w_EN(bit_index),
            o_Z => w_D(bit_index));
114  -- Store the selected value to the register
     u_myRegN: myRegN
116  generic map (m_width => 32)
     port map (i_clk => i_clk,
118         i_reset => i_reset,
            i_D => w_D(bit_index),
120         o_Q => f_Q(bit_index));
```

```
  end generate reg_array;

  --| Register 0 should retain its zero value regardless of
   its being
  --| selected or the value of i_data.
  u_myMUX2_00: myMUX2_N
  generic map (m_width => 32)
  port map (i_0 => f_Q(0),
            i_1 => f_Q(0),
            i_S => w_EN(0),
            o_Z => w_D(0));
  --| Store the selected value to the register
  u_myReg00: myRegN
  generic map (m_width => 32)
  port map (i_clk => i_clk,
            i_reset => i_reset,
            i_D => w_D(0),
            o_Q => f_Q(0));

  --| Connect output signals
  o_Q <= f_Q;
--  o_Q00 <= f_Q(0);
--  o_Q01 <= f_Q(1);
--  o_Q02 <= f_Q(2);
--  o_Q03 <= f_Q(3);
--  o_Q04 <= f_Q(4);
--  o_Q05 <= f_Q(5);
--  o_Q06 <= f_Q(6);
--  o_Q07 <= f_Q(7);
--  o_Q08 <= f_Q(8);
--  o_Q09 <= f_Q(9);
--  o_Q10 <= f_Q(10);
--  o_Q11 <= f_Q(11);
--  o_Q12 <= f_Q(12);
--  o_Q13 <= f_Q(13);
--  o_Q14 <= f_Q(14);
--  o_Q15 <= f_Q(15);
--  o_Q16 <= f_Q(16);
--  o_Q17 <= f_Q(17);
--  o_Q18 <= f_Q(18);
--  o_Q19 <= f_Q(19);
--  o_Q20 <= f_Q(20);
--  o_Q21 <= f_Q(21);
```

```
--    o_Q22 <= f_Q(22);
--    o_Q23 <= f_Q(23);
--    o_Q24 <= f_Q(24);
--    o_Q25 <= f_Q(25);
--    o_Q26 <= f_Q(26);
--    o_Q27 <= f_Q(27);
--    o_Q28 <= f_Q(28);
--    o_Q29 <= f_Q(29);
--    o_Q30 <= f_Q(30);
--    o_Q31 <= f_Q(31);

end a_GPR_Bank;
```

Listing A.40. Enabled_Register.vhd Code

1.19 Datapath

```
--| Datapath.vhd
--| Execute the MIPS instructions as controlled by the controller.
--|
--| INPUTS:
--| i_ALU_SRC_A - ALU Source A selector
--| i_ALU_SRC_B - ALU Source B selector
--| i_ALU_INV_B - ALU Invert B selector
--| i_COMP_SEL  - Select the value to output from the Comparator
--| i_OVER_CTRL - Select which overflow detection means should be used
--|             (i.e. ADD, ADDI, SUB, or no overflow detection).
--| i_ALU_OUTPUT  - Select which ALU function should be output
--|             0 - Shift Left
--|             1 - Shift Right
--|             2 - Adder
--|             3 - AND
--|             4 - OR
--|             5 - XOR
--|             6 - NOR
--|             7 - Comparator
--|             8 - LUI
--|             9, 10, 11, 12, 13, 14, 15 - Shift Left
--| i_RS_SEL    - Select the register from which the RS_DATA will be supplied to the ALU
--| i_RT_SEL    - Select the register from which the RT_DATA will be supplied to the ALU
--| i_imm       - Immediate/offset value
--| i_unsigned    - Determine if the comparator is comparing sigend numbers (0) or unsigned
--|             numbers (1).
--| i_overflow    - Determine if the comparator is performing a Less Than comparison that will
--|                 or will not produce an overflow.  No overflow can occur in BLTZ instructions,
--|             but overflow can occur in SLT, SLTU, SLTI, and SLTIU instructions.  0 - no
--|             overflow can occur, 1 - overflow can occur
```

```
--| i_STORE_FROM_MEM - Indicates data should be stored to a GPR from memory
--|                    rather than from the ALU output.
--| i_REG_SEL      - Determine which register to which the ALU output should be stored
--| i_PC_EN        - When '1', PC=PC+4.  When '0', the next value of PC depends on i_REG_SEL
--| i_MEM_ADDRESS_SEL - Determines whether the output of the ALU or the PC value should be
--|                    used as the adddress to access memory.
--| i_MEM_OUT      - Output from memory to be stored into a register
--| i_imm_extend   - Determine if the immediate value should be 0 extended or sign extended
--|
--| OUTPUTS:
--| o_MEM_ADDRESS - Address used for accessing memory
--| o_RT_DATA     - Data to write to memory from the register indicated by i_RT_SEL
--|
library IEEE;
use IEEE.std_logic_1164.all;
use work.my_word_array_package.all;

entity Datapath is
  port (i_clk          : in  std_logic;
        i_reset        : in  std_logic;
        i_ALU_SRC_A    : in  std_logic_vector(1 downto 0);
        i_ALU_SRC_B    : in  std_logic_vector(1 downto 0);
        i_ALU_INV_B    : in  std_logic;
        i_COMP_SEL     : in  std_logic_vector(2 downto 0);
        i_OVER_CTRL    : in  std_logic_vector(1 downto 0);
        i_ALU_OUTPUT   : in  std_logic_vector(3 downto 0);
        i_RS_SEL       : in  std_logic_vector(4 downto 0);
        i_RT_SEL       : in  std_logic_vector(4 downto 0);
        i_imm          : in  std_logic_vector(15 downto 0);
        i_UNSIGNED     : in  std_logic;
        i_overflow     : in  std_logic;
        i_imm_extend   : in  std_logic;
        i_STORE_FROM_MEM : in  std_logic;
        i_REG_SEL      : in  std_logic_vector(1 downto 0);
        i_PC_EN        : in  std_logic;
        i_MEM_ADDRESS_SEL : in  std_logic;
```

```
      i_MEM_OUT      : in  std_logic_vector(31 downto 0);
      o_MEM_ADDRESS  : out std_logic_vector(31 downto 0);
      o_RT_DATA      : out std_logic_vector(31 downto 0));
end Datapath;

architecture a_Datapath of Datapath is
--| Define Components
  -- General Purpose Register Bank
  component GPR_Bank is
    port (i_clk    : in  std_logic;
          i_reset  : in  std_logic;
          i_data   : in  std_logic_vector(31 downto 0);
          i_sel    : in  std_logic_vector(4 downto 0);
          o_Q      : out word_array);
--          o_Q00    : out std_logic_vector(31 downto 0);
--          o_Q01    : out std_logic_vector(31 downto 0);
--          o_Q02    : out std_logic_vector(31 downto 0);
--          o_Q03    : out std_logic_vector(31 downto 0);
--          o_Q04    : out std_logic_vector(31 downto 0);
--          o_Q05    : out std_logic_vector(31 downto 0);
--          o_Q06    : out std_logic_vector(31 downto 0);
--          o_Q07    : out std_logic_vector(31 downto 0);
--          o_Q08    : out std_logic_vector(31 downto 0);
--          o_Q09    : out std_logic_vector(31 downto 0);
--          o_Q10    : out std_logic_vector(31 downto 0);
--          o_Q11    : out std_logic_vector(31 downto 0);
--          o_Q12    : out std_logic_vector(31 downto 0);
--          o_Q13    : out std_logic_vector(31 downto 0);
--          o_Q14    : out std_logic_vector(31 downto 0);
--          o_Q15    : out std_logic_vector(31 downto 0);
--          o_Q16    : out std_logic_vector(31 downto 0);
--          o_Q17    : out std_logic_vector(31 downto 0);
--          o_Q18    : out std_logic_vector(31 downto 0);
--          o_Q19    : out std_logic_vector(31 downto 0);
--          o_Q20    : out std_logic_vector(31 downto 0);
--          o_Q21    : out std_logic_vector(31 downto 0);
--          o_Q22    : out std_logic_vector(31 downto 0);
--          o_Q23    : out std_logic_vector(31 downto 0);
--          o_Q24    : out std_logic_vector(31 downto 0);
--          o_Q25    : out std_logic_vector(31 downto 0);
--          o_Q26    : out std_logic_vector(31 downto 0);
--          o_Q27    : out std_logic_vector(31 downto 0);
--          o_Q28    : out std_logic_vector(31 downto 0);
```

```
110 --         o_Q29    : out std_logic_vector(31 downto 0);
    --         o_Q30    : out std_logic_vector(31 downto 0);
112 --         o_Q31    : out std_logic_vector(31 downto 0));
    end component;
114
    -- ALU Core
116 component ALU_CORE is
      port (i_clk        : in std_logic;
118         i_reset      : in std_logic;
            i_ALU_SRC_A  : in  std_logic_vector(1 downto 0);
120         i_ALU_SRC_B  : in  std_logic_vector(1 downto 0);
            i_ALU_INV_B  : in  std_logic;
122         i_COMP_SEL   : in  std_logic_vector(2 downto 0);
            i_OVER_CTRL  : in  std_logic_vector(1 downto 0);
124         i_ALU_OUTPUT : in  std_logic_vector(3 downto 0);
            i_RS_DATA    : in  std_logic_vector(31 downto 0);
126         i_RT_DATA    : in  std_logic_vector(31 downto 0);
            i_RD_DATA    : in  std_logic_vector(31 downto 0);
128         i_PC         : in  std_logic_vector(31 downto 0);
            i_imm        : in  std_logic_vector(15 downto 0);
130         i_UNSIGNED   : in  std_logic;
            i_overflow   : in  std_logic;
132         i_imm_extend : in  std_logic;
            o_ALU_RESULT : out std_logic_vector(31 downto 0));
134 end component;

136 -- 2-input 32-bit mux
    component myMUX2_N is
138   generic (m_width : integer := 32);
      port (i_0 : in  std_logic_vector(m_width-1 downto 0);
140         i_1 : in  std_logic_vector(m_width-1 downto 0);
            i_S : in  std_logic;
142         o_Z : out std_logic_vector(m_width-1 downto 0)
            );
144 end component;

146 -- 2-input 1-bit mux
    component myMUX2_1 is
148   port (i_0 : in  std_logic;
            i_1 : in  std_logic;
150         i_S : in  std_logic;
            o_Z : out std_logic
152         );
```

```
  end component;

  -- 4-input N-bit mux
  component myMUX4_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(m_width-1 downto 0);
          i_1 : in  std_logic_vector(m_width-1 downto 0);
          i_2 : in  std_logic_vector(m_width-1 downto 0);
          i_3 : in  std_logic_vector(m_width-1 downto 0);
          i_S : in  std_logic_vector(1 downto 0);
          o_Z : out std_logic_vector(m_width-1 downto 0)
          );
  end component;

  -- 4-input 1-bit mux
  component myMUX4_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_S : in  std_logic_vector(1 downto 0);
          o_Z : out std_logic
          );
  end component;

  -- 32-input N-bit mux
  component myMUX32_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(m_width-1 downto 0);
          i_1 : in  std_logic_vector(m_width-1 downto 0);
          i_2 : in  std_logic_vector(m_width-1 downto 0);
          i_3 : in  std_logic_vector(m_width-1 downto 0);
          i_4 : in  std_logic_vector(m_width-1 downto 0);
          i_5 : in  std_logic_vector(m_width-1 downto 0);
          i_6 : in  std_logic_vector(m_width-1 downto 0);
          i_7 : in  std_logic_vector(m_width-1 downto 0);
          i_8 : in  std_logic_vector(m_width-1 downto 0);
          i_9 : in  std_logic_vector(m_width-1 downto 0);
          i_10 : in  std_logic_vector(m_width-1 downto 0);
          i_11 : in  std_logic_vector(m_width-1 downto 0);
          i_12 : in  std_logic_vector(m_width-1 downto 0);
          i_13 : in  std_logic_vector(m_width-1 downto 0);
          i_14 : in  std_logic_vector(m_width-1 downto 0);
```

```
196         i_15 : in  std_logic_vector(m_width-1 downto 0);
            i_16 : in  std_logic_vector(m_width-1 downto 0);
198         i_17 : in  std_logic_vector(m_width-1 downto 0);
            i_18 : in  std_logic_vector(m_width-1 downto 0);
200         i_19 : in  std_logic_vector(m_width-1 downto 0);
            i_20 : in  std_logic_vector(m_width-1 downto 0);
202         i_21 : in  std_logic_vector(m_width-1 downto 0);
            i_22 : in  std_logic_vector(m_width-1 downto 0);
204         i_23 : in  std_logic_vector(m_width-1 downto 0);
            i_24 : in  std_logic_vector(m_width-1 downto 0);
206         i_25 : in  std_logic_vector(m_width-1 downto 0);
            i_26 : in  std_logic_vector(m_width-1 downto 0);
208         i_27 : in  std_logic_vector(m_width-1 downto 0);
            i_28 : in  std_logic_vector(m_width-1 downto 0);
210         i_29 : in  std_logic_vector(m_width-1 downto 0);
            i_30 : in  std_logic_vector(m_width-1 downto 0);
212         i_31 : in  std_logic_vector(m_width-1 downto 0);
            i_S : in  std_logic_vector(4 downto 0);
214         o_Z : out std_logic_vector(m_width-1 downto 0));
    end component;
216
    -- Inverter
218 component myINV is
      port (i_A : in  std_logic;
220         o_Z : out std_logic
            );
222 end component;

224 -- Enabled Register
    component Enabled_Register is
226   port (i_clk   : in  std_logic;
            i_reset : in  std_logic;
228         i_data  : in  std_logic_vector(31 downto 0);
            i_en    : in  std_logic;
230         o_Q     : out std_logic_vector(31 downto 0));
    end component;
232
    -- ADDER + 4
234 component CARRY_SELECT_ADDER_30_SC is
      port (i_A     : in  std_logic_vector(29 downto 0);
236         i_B     : in  std_logic_vector(29 downto 0);
            i_C     : in  std_logic;
238         o_S     : out std_logic_vector(29 downto 0);
```

```
        o_C     : out std_logic);
  end component;

--| Define Signals
  -- GPR_BANK Signals
  signal w_GPR_DATA_IN  : std_logic_vector(31 downto 0);  -- Data in to the GPR_BANK
  signal w_GPR_SEL    : std_logic_vector(4 downto 0); -- Select which register will to which to store data
  signal w_Q        : word_array;
--  signal w_Q00      : std_logic_vector(31 downto 0);  -- Data out from register 0
--  signal w_Q01      : std_logic_vector(31 downto 0);  -- Data out from register 1
--  signal w_Q02      : std_logic_vector(31 downto 0);
--  signal w_Q03      : std_logic_vector(31 downto 0);
--  signal w_Q04      : std_logic_vector(31 downto 0);
--  signal w_Q05      : std_logic_vector(31 downto 0);
--  signal w_Q06      : std_logic_vector(31 downto 0);
--  signal w_Q07      : std_logic_vector(31 downto 0);
--  signal w_Q08      : std_logic_vector(31 downto 0);
--  signal w_Q09      : std_logic_vector(31 downto 0);
--  signal w_Q10      : std_logic_vector(31 downto 0);
--  signal w_Q11      : std_logic_vector(31 downto 0);
--  signal w_Q12      : std_logic_vector(31 downto 0);
--  signal w_Q13      : std_logic_vector(31 downto 0);
--  signal w_Q14      : std_logic_vector(31 downto 0);
--  signal w_Q15      : std_logic_vector(31 downto 0);
--  signal w_Q16      : std_logic_vector(31 downto 0);
--  signal w_Q17      : std_logic_vector(31 downto 0);
--  signal w_Q18      : std_logic_vector(31 downto 0);
--  signal w_Q19      : std_logic_vector(31 downto 0);
--  signal w_Q20      : std_logic_vector(31 downto 0);
--  signal w_Q21      : std_logic_vector(31 downto 0);
--  signal w_Q22      : std_logic_vector(31 downto 0);
--  signal w_Q23      : std_logic_vector(31 downto 0);
--  signal w_Q24      : std_logic_vector(31 downto 0);
--  signal w_Q25      : std_logic_vector(31 downto 0);
--  signal w_Q26      : std_logic_vector(31 downto 0);
--  signal w_Q27      : std_logic_vector(31 downto 0);
--  signal w_Q28      : std_logic_vector(31 downto 0);
--  signal w_Q29      : std_logic_vector(31 downto 0);
```

```
--  signal w_Q30      : std_logic_vector(31 downto 0);  -- Data out from register 30
--  signal w_Q31      : std_logic_vector(31 downto 0);  -- Data out from register 31

  -- ALU Signals
  signal w_RS_DATA    : std_logic_vector(31 downto 0); -- Data from register indicated by i_RS_SEL
  signal w_RT_DATA    : std_logic_vector(31 downto 0); -- Data from register indicated by i_RT_SEL
  signal w_RD_DATA    : std_logic_vector(31 downto 0); -- Data from register indicated by i_RD_SEL
  signal w_ALU_RESULT : std_logic_vector(31 downto 0); -- Output of ALU Core

  -- PC Signals
  signal w_PC         : std_logic_vector(31 downto 0); -- PC Data
  signal w_PC_4       : std_logic_vector(31 downto 0); -- PC Data + 4
  signal w_PC_IN      : std_logic_vector(31 downto 0); -- PC Register Input
  signal w_PC_ENABLE  : std_logic; -- Enable the PC Register Signal
  signal w_PC_ENABLE0 : std_logic; -- Output of MUX to determine if the PC Register should be enabled based on the i_REG_SEL signal
  signal w_PC_EN_n    : std_logic; -- Inverted i_PC_EN

--| Define Constants
  constant k_zero_5 : std_logic_vector(4 downto 0) := (others => '0'); -- 5-bit zero
  constant k_zero_1 : std_logic := '0'; -- 1-bit zero
  constant k_one_1  : std_logic := '1'; -- 1 bit one
  constant k_one_30 : std_logic_vector(29 downto 0) := B"00_0000_0000_0000_0000_0000_0000_0001";

begin
  -- MUX to determine the input data to the GPR_BANK
  u_myMUX_GPR_DATA_IN: myMUX2_N
  generic map (m_width => 32)
  port map (i_0 => w_ALU_RESULT,
            i_1 => i_MEM_OUT,
```

```
306             i_S => i_STORE_FROM_MEM,
                o_Z => w_GPR_DATA_IN);
308
    -- MUX to select which register will be enabled
310     u_myMUX4_GPR_SEL: myMUX4_N
        generic map (m_width => 5)
312     port map (i_0 => k_zero_5,
                  i_1 => i_imm(15 downto 11),--i_RD_SEL
314               i_2 => i_RT_SEL,
                  i_3 => k_zero_5,
316               i_S => i_REG_SEL,
                  o_Z => w_GPR_SEL);
318
    -- Connect the GPR_BANK
320     u_GPR_BANK: GPR_BANK
        port map (i_clk => i_clk,
322               i_reset => i_reset,
                  i_data => w_GPR_DATA_IN,
324               i_sel => w_GPR_SEL,
                  o_Q => w_Q);
326 --                o_Q00 => w_Q00,
    --                o_Q01 => w_Q01,
328 --                o_Q02 => w_Q02,
    --                o_Q03 => w_Q03,
330 --                o_Q04 => w_Q04,
    --                o_Q05 => w_Q05,
332 --                o_Q06 => w_Q06,
    --                o_Q07 => w_Q07,
334 --                o_Q08 => w_Q08,
    --                o_Q09 => w_Q09,
336 --                o_Q10 => w_Q10,
    --                o_Q11 => w_Q11,
338 --                o_Q12 => w_Q12,
    --                o_Q13 => w_Q13,
340 --                o_Q14 => w_Q14,
    --                o_Q15 => w_Q15,
342 --                o_Q16 => w_Q16,
    --                o_Q17 => w_Q17,
344 --                o_Q18 => w_Q18,
    --                o_Q19 => w_Q19,
346 --                o_Q20 => w_Q20,
    --                o_Q21 => w_Q21,
348 --                o_Q22 => w_Q22,
```

```
--            o_Q23 => w_Q23,
--            o_Q24 => w_Q24,
--            o_Q25 => w_Q25,
--            o_Q26 => w_Q26,
--            o_Q27 => w_Q27,
--            o_Q28 => w_Q28,
--            o_Q29 => w_Q29,
--            o_Q30 => w_Q30,
--            o_Q31 => w_Q31);

  -- Connect the MUX to select the RS Data from the GPR Bank
  u_myMUX32_N_RS: myMUX32_N
  generic map (m_width => 32)
  port map (i_0 => w_Q(0),
            i_1 => w_Q(1),
            i_2 => w_Q(2),
            i_3 => w_Q(3),
            i_4 => w_Q(4),
            i_5 => w_Q(5),
            i_6 => w_Q(6),
            i_7 => w_Q(7),
            i_8 => w_Q(8),
            i_9 => w_Q(9),
            i_10 => w_Q(10),
            i_11 => w_Q(11),
            i_12 => w_Q(12),
            i_13 => w_Q(13),
            i_14 => w_Q(14),
            i_15 => w_Q(15),
            i_16 => w_Q(16),
            i_17 => w_Q(17),
            i_18 => w_Q(18),
            i_19 => w_Q(19),
            i_20 => w_Q(20),
            i_21 => w_Q(21),
            i_22 => w_Q(22),
            i_23 => w_Q(23),
            i_24 => w_Q(24),
            i_25 => w_Q(25),
            i_26 => w_Q(26),
            i_27 => w_Q(27),
            i_28 => w_Q(28),
            i_29 => w_Q(29),
```

```
392             i_30 => w_Q(30),
                i_31 => w_Q(31),
394             i_S => i_RS_SEL,
                o_Z => w_RS_DATA);
396
    -- Connect the MUX to select the RT Data from the GPR Bank
398 u_myMUX32_N_RT: myMUX32_N
    generic map (m_width => 32)
400 port map (i_0 => w_Q(0),
                i_1 => w_Q(1),
402             i_2 => w_Q(2),
                i_3 => w_Q(3),
404             i_4 => w_Q(4),
                i_5 => w_Q(5),
406             i_6 => w_Q(6),
                i_7 => w_Q(7),
408             i_8 => w_Q(8),
                i_9 => w_Q(9),
410             i_10 => w_Q(10),
                i_11 => w_Q(11),
412             i_12 => w_Q(12),
                i_13 => w_Q(13),
414             i_14 => w_Q(14),
                i_15 => w_Q(15),
416             i_16 => w_Q(16),
                i_17 => w_Q(17),
418             i_18 => w_Q(18),
                i_19 => w_Q(19),
420             i_20 => w_Q(20),
                i_21 => w_Q(21),
422             i_22 => w_Q(22),
                i_23 => w_Q(23),
424             i_24 => w_Q(24),
                i_25 => w_Q(25),
426             i_26 => w_Q(26),
                i_27 => w_Q(27),
428             i_28 => w_Q(28),
                i_29 => w_Q(29),
430             i_30 => w_Q(30),
                i_31 => w_Q(31),
432             i_S => i_RT_SEL,
                o_Z => w_RT_DATA);
434
```

```
-- Connect the MUX to select the RD Data from the GPR Bank
u_myMUX32_N_RD: myMUX32_N
generic map (m_width => 32)
port map (i_0 => w_Q(0),
          i_1 => w_Q(1),
          i_2 => w_Q(2),
          i_3 => w_Q(3),
          i_4 => w_Q(4),
          i_5 => w_Q(5),
          i_6 => w_Q(6),
          i_7 => w_Q(7),
          i_8 => w_Q(8),
          i_9 => w_Q(9),
          i_10 => w_Q(10),
          i_11 => w_Q(11),
          i_12 => w_Q(12),
          i_13 => w_Q(13),
          i_14 => w_Q(14),
          i_15 => w_Q(15),
          i_16 => w_Q(16),
          i_17 => w_Q(17),
          i_18 => w_Q(18),
          i_19 => w_Q(19),
          i_20 => w_Q(20),
          i_21 => w_Q(21),
          i_22 => w_Q(22),
          i_23 => w_Q(23),
          i_24 => w_Q(24),
          i_25 => w_Q(25),
          i_26 => w_Q(26),
          i_27 => w_Q(27),
          i_28 => w_Q(28),
          i_29 => w_Q(29),
          i_30 => w_Q(30),
          i_31 => w_Q(31),
          i_S => i_imm(15 downto 11), --i_RD_SEL
          o_Z => w_RD_DATA);

-- Connect the ALU_CORE
u_ALU_CORE: ALU_CORE
port map(i_clk   => i_clk,
         i_reset => i_reset,
         i_ALU_SRC_A => i_ALU_SRC_A,
```

```
478         i_ALU_SRC_B => i_ALU_SRC_B,
            i_ALU_INV_B => i_ALU_INV_B,
480         i_COMP_SEL => i_COMP_SEL,
            i_OVER_CTRL => i_OVER_CTRL,
482         i_ALU_OUTPUT => i_ALU_OUTPUT,
            i_RS_DATA => w_RS_DATA,
484         i_RT_DATA => w_RT_DATA,
            i_RD_DATA => w_RD_DATA,
486         i_PC => w_PC,
            i_imm => i_imm,
488         i_UNSIGNED => i_UNSIGNED,
            i_overflow => i_overflow,
490         i_imm_extend => i_imm_extend,
            o_ALU_RESULT => w_ALU_RESULT);
492
    -- Connect the MUX to determine the input data to the PC
     Register
494 u_myMUX_PC_IN: myMUX2_N
    generic map (m_width => 32)
496 port map (i_0 => w_PC_4,
            i_1 => w_ALU_RESULT,
498         i_S => w_PC_EN_n,
            o_Z => w_PC_IN);
500
    -- Connect the MUX to determine whether the PC register
     should be enabled based on i_REG_SEL
502 u_myMUX4_PC_ENABLE0: myMUX4_1
    port map (i_0 => k_zero_1,
504         i_1 => k_zero_1,
            i_2 => k_zero_1,
506         i_3 => k_one_1,
            i_S => i_REG_SEL,
508         o_Z => w_PC_ENABLE0);

510 -- Connect the MUX to determine whether the PC register
     should be enabled based on i_PC_EN or i_REG_SEL
    u_myMUX2_PC_ENABLE: myMUX2_1
512 port map (i_0 => w_PC_ENABLE0,
            i_1 => k_one_1,
514         i_S => i_PC_EN,
            o_Z => w_PC_ENABLE);
516
    -- Connect the inverter to invert i_PC_EN
```

```
518   u_myINV: myINV
      port map (i_A => i_PC_EN,
520             o_Z => w_PC_EN_n);

522   -- Connect the PC Register
      u_Enabled_Register: Enabled_Register
524   port map (i_clk => i_clk,
              i_reset => i_reset,
526           i_data => w_PC_IN,
              i_en => w_PC_ENABLE,
528           o_Q => w_PC);

530   -- Connect the adder to calculate PC+4
      u_CARRY_SELECT_ADDER_30_SC: CARRY_SELECT_ADDER_30_SC
532   port map (i_A => w_PC(31 downto 2),
              i_B => k_one_30,
534           i_C => k_zero_1,
              o_S => w_PC_4(31 downto 2),
536           o_C => open);
      w_PC_4(1 downto 0) <= w_PC(1 downto 0);
538
      -- Connect the MUX to determine o_MEM_ADDRESS
540   u_myMUX_MEM_ADDRESS: myMUX2_N
      generic map (m_width => 32)
542   port map (i_0 => w_PC,
              i_1 => w_ALU_RESULT,
544           i_S => i_MEM_ADDRESS_SEL,
              o_Z => o_MEM_ADDRESS);
546
      -- Connect remaining output
548   o_RT_DATA <= w_RT_DATA;
  end a_Datapath;
```

Listing A.41. Datapath.vhd Code

1.20 Instruction Decoder

```
1  --| Instruction_Decoder
   --| Decodes the 32-bit MIPS instruction to determine which
   portions of the ALU to activate.
3  --|   ALU_SRC_A determines whether the A input to the ALU is
   the regester specified by RS, SA, or
   --|     the Program Counter (PC)
5  --|
   --|   ALU_SRC_B determines whether the B input to the ALU is
   the register specified by RT or if
7  --|   the input is zero, a 16 bit sign extended immediate or
   offset value, or a 16 bit sign
   --|   extended offset value shifted left by 2 bits.
9  --|
   --| INV_B determines whether the B input should be negated to
    perform a subtraction rather
11 --|   than an addition
   --|
13 --| Comparison Outputs
   --| LTZ is high if the output of the comparator should be the
    less than zero output
15 --| EQ is high if the output of the comparator should be the
   equal to zero output
   --| NE is high if the output of the comparator should be the
   not equal to zero output
17 --| LEZ is high if the output of the comparator should be the
    less than or equal to zero output
   --| GTZ is high if the output of the comparator should be the
    greater than or equal to zero output
19 --|   Note that there is no outputput for greater than or
   equal to zero.  There is separate
   --|   encoder outside the Instruction_Decoder that takes the
   5 comparison signals and encodes them
21 --|   as a 3-bit signal.  If all of these signals are 0, it
   is assumed that the greater than or
   --|   equal to zero output should be selected.  Additionally,
    these comarisons are performed on
23 --|   the output of the adder, so in reality, the comparison
   is performed on the result of S = A+B
   --|   where B may be inverted and then the comparison is
   really between A and B since (or S and 0).
25 --|
```

```
--| ALU Output Selection
--| RIGHT_SHIFT is high if the ALU output should come from the right shifter
--| ADDER is high if the ALU output should come from the adder
--| AND is high if the ALU output should come from the bitwise and
--| OR is high if the ALU output should come from the bitwise or
--| XOR is high if the ALU output should come from the bitwise xor
--| NOR is high if the ALU output should come from the bitwise nor
--| COMPARE is high if the ALU output should come from the comparator
--|    (i.e. output a 32-bit 0 if the comparison is false or a 32-bit 1 if the comparison is true)
--| LUI is high if the ALU output should come from the load upper immediate
--|    If all of the ALU output selection values are 0, the default output of LEFT_SHIFT is
--|    selected.  This is accomplished in a separate encoder which converts these individual
--|    signals into a 4-bit encoded signal.
--|
--| REG_SEL determines the regester to which the results of the ALU computation will be
--|    writen.  This could be the zero register, the PC register, or the register specified by
--|    RT or RD.
--|
--| OVER_CTRL determines whether the output of the adder should be checked for an overflow.
--|    The only instructions requiring overflow detection are add (ADD), add immediate (ADDI),
--|    and subtract (SUB).
--|
--| FSM_CTRL determines what the next state of the FSM should be after the instruction is stored
--|    from memory.  It depends on whether the instruction is a branch, load word, or store word
--|    instruction.  If it is none of these, FSM_CTRL = B"00".  For a branch, FSM_CTRL = B"01".
```

```
51 --|    For a load word, FSM_CTRL = B"10".  For a store word,
   FSM_CTRL = B"11".
   --|
53 --| INPUTS:
   --| i_instr31 - Instruction inputs
55 --| ...
   --| i_instr00
57 --|
   --| OUTPUTS:
59 --| o_ALU_SRC_A0    - ALU Source of Input A Selector Signal
   bit 0
   --| o_ALU_SRC_B   - ALU Source of Input B Selector SIgnal
61 --| o_INV_B       - Invert ALU Input B Signal
   --| o_LTZ        - Less than zero
63 --| o_EQ          - Equal to zero
   --| o_NE          - Not equal to zero
65 --| o_LEZ        - Less than or equal to zero
   --| o_GTZ        - Greater than zero
67 --| o_RIGHT_SHIFT   - ALU Output = Right Shift Output
   --| o_ADDER       - ALU Output = Adder Output
69 --| o_AND        - ALU Output = AND output
   --| o_OR          - ALU Output = OR output
71 --| o_XOR        - ALU Output = XOR output
   --| o_NOR        - ALU Output = NOR output
73 --| o_COMPARE     - ALU Output = Comparator output
   --| o_LUI        - ALU Output = Load upper immediate output
75 --| o_REG_SEL     - Register selct Signal
   --| o_OVER_CTRL   - Overflow Control Signal
77 --| o_UNSIGNED      - Determine if the comparator is
   comparing sigend numbers (0) or unsigned
   --|                numbers (1).
79 --| o_overflow      - Determine if the comparator is
   performing a Less Than comparison that will
   --|                     or will not produce an overflow.  No
   overflow can occur in BLTZ instructions,
81 --|                but overflow can occur in SLT, SLTU, SLTI,
   and SLTIU instructions.  0 - no
   --|               overflow can occur, 1 - overflow can occur
83 --| o_imm_extend    - Determine if the immediate value should
    be 0 extended or sign extended
   --| o_FSM_CTRL      - Finite State Machine Control Signal
85 --| o_INVALID_INSTR - Signals when an invalid instruction is
   received
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Instruction_Decoder is
  port (i_instr31    : in  std_logic;
        i_instr30    : in  std_logic;
        i_instr29    : in  std_logic;
        i_instr28    : in  std_logic;
        i_instr27    : in  std_logic;
        i_instr26    : in  std_logic;
        i_instr20    : in  std_logic;
        i_instr19    : in  std_logic;
        i_instr18    : in  std_logic;
        i_instr17    : in  std_logic;
        i_instr16    : in  std_logic;
        i_instr05    : in  std_logic;
        i_instr04    : in  std_logic;
        i_instr03    : in  std_logic;
        i_instr02    : in  std_logic;
        i_instr01    : in  std_logic;
        i_instr00    : in  std_logic;
        o_ALU_SRC_A0  : out std_logic;
        o_ALU_SRC_B   : out std_logic_vector(1 downto 0);
        o_INV_B       : out std_logic;
        o_LTZ         : out std_logic;
        o_EQ          : out std_logic;
        o_NE          : out std_logic;
        o_LEZ         : out std_logic;
        o_GTZ         : out std_logic;
        o_RIGHT_SHIFT : out std_logic;
        o_ADDER       : out std_logic;
        o_AND         : out std_logic;
        o_OR          : out std_logic;
        o_XOR         : out std_logic;
        o_NOR         : out std_logic;
        o_COMPARE     : out std_logic;
        o_LUI         : out std_logic;
        o_REG_SEL     : out std_logic_vector(1 downto 0);
        o_OVER_CTRL   : out std_logic_vector(1 downto 0);
        o_UNSIGNED    : out std_logic;
        o_overflow    : out std_logic;
        o_imm_extend  : out std_logic;
        o_FSM_CTRL    : out std_logic_vector(1 downto 0);
```

```
129         o_INVALID_INSTR: out std_logic);
end Instruction_Decoder;

architecture a_Instruction_Decoder of Instruction_Decoder is
--| Declare Components
  -- Declare NAND_2
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;
  -- Declare Inverter
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  --| Declare Signals
  -- Inverted Instruction Signals
  signal w_instr31_n : std_logic;
  signal w_instr30_n : std_logic;
  signal w_instr29_n : std_logic;
  signal w_instr28_n : std_logic;
  signal w_instr27_n : std_logic;
  signal w_instr26_n : std_logic;
  signal w_instr20_n : std_logic;
  signal w_instr19_n : std_logic;
  signal w_instr18_n : std_logic;
  signal w_instr17_n : std_logic;
  signal w_instr16_n : std_logic;
  signal w_instr05_n : std_logic;
  signal w_instr04_n : std_logic;
  signal w_instr03_n : std_logic;
  signal w_instr02_n : std_logic;
  signal w_instr01_n : std_logic;
  signal w_instr00_n : std_logic;
  -- Intermediate control signals (and inverted control
   signals) used to compute outputs
  signal w_ctrl_000, w_ctrl_000_n : std_logic;
  signal w_ctrl_001, w_ctrl_001_n : std_logic;
  signal w_ctrl_002, w_ctrl_002_n : std_logic;
```

```
171   signal w_ctrl_003, w_ctrl_003_n : std_logic;
      signal w_ctrl_004, w_ctrl_004_n : std_logic;
173   signal w_ctrl_005, w_ctrl_005_n : std_logic;
      signal w_ctrl_006, w_ctrl_006_n : std_logic;
175   signal w_ctrl_007, w_ctrl_007_n : std_logic;
      signal w_ctrl_008 : std_logic;
177   signal w_ctrl_009, w_ctrl_009_n : std_logic;
      signal w_ctrl_010 : std_logic;
179   signal w_ctrl_011, w_ctrl_011_n : std_logic;
      signal w_ctrl_012 : std_logic;
181   signal w_ctrl_013 : std_logic;
      signal w_ctrl_014, w_ctrl_014_n : std_logic;
183   signal w_ctrl_015, w_ctrl_015_n : std_logic;
      signal w_ctrl_018, w_ctrl_018_n : std_logic;
185   signal w_ctrl_020, w_ctrl_020_n : std_logic;
      signal w_ctrl_025 : std_logic;
187   signal w_ctrl_028, w_ctrl_028_n : std_logic;
      signal w_ctrl_029, w_ctrl_029_n : std_logic;
189   signal w_ctrl_030 : std_logic;
      signal w_ctrl_031 : std_logic;
191   signal w_ctrl_032 : std_logic;
      signal w_ctrl_033 : std_logic;
193   signal w_ctrl_034, w_ctrl_034_n : std_logic;
      signal w_ctrl_035 : std_logic;
195   signal w_ctrl_036 : std_logic;
      signal w_ctrl_037, w_ctrl_037_n : std_logic;
197   signal w_ctrl_038, w_ctrl_038_n : std_logic;
      signal w_ctrl_039, w_ctrl_039_n : std_logic;
199   signal w_ctrl_040 : std_logic;
      signal w_ctrl_041, w_ctrl_041_n : std_logic;
201   signal w_ctrl_042 : std_logic;
      signal w_ctrl_043, w_ctrl_043_n : std_logic;
203   signal w_ctrl_044, w_ctrl_044_n : std_logic;
      signal w_ctrl_046, w_ctrl_046_n : std_logic;
205   signal w_ctrl_047 : std_logic;
      signal w_ctrl_048, w_ctrl_048_n : std_logic;
207   signal w_ctrl_049 : std_logic;
      signal w_ctrl_050 : std_logic;
209   signal w_ctrl_051, w_ctrl_051_n : std_logic;
      signal w_ctrl_052 : std_logic;
211   signal w_ctrl_053, w_ctrl_053_n : std_logic;
      signal w_ctrl_054, w_ctrl_054_n : std_logic;
213   signal w_ctrl_055, w_ctrl_055_n : std_logic;
```

```
    signal w_ctrl_056, w_ctrl_056_n : std_logic;
215 signal w_ctrl_057, w_ctrl_057_n : std_logic;
    signal w_ctrl_058, w_ctrl_058_n : std_logic;
217 signal w_ctrl_059 : std_logic;
    signal w_ctrl_060 : std_logic;
219 signal w_ctrl_061 : std_logic;
    signal w_ctrl_062, w_ctrl_062_n : std_logic;
221 signal w_ctrl_063, w_ctrl_063_n : std_logic;
    signal w_ctrl_064 : std_logic;
223 signal w_ctrl_065, w_ctrl_065_n : std_logic;
    signal w_ctrl_066, w_ctrl_066_n : std_logic;
225 signal w_ctrl_067 : std_logic;
    signal w_ctrl_068 : std_logic;
227 signal w_ctrl_069 : std_logic;
    signal w_ctrl_070, w_ctrl_070_n : std_logic;
229 signal w_ctrl_071, w_ctrl_071_n : std_logic;
    signal w_ctrl_072 : std_logic;
231 signal w_ctrl_073, w_ctrl_073_n : std_logic;
    signal w_ctrl_074, w_ctrl_074_n : std_logic;
233 signal w_ctrl_075 : std_logic;
    signal w_ctrl_076 : std_logic;
235 signal w_ctrl_077, w_ctrl_077_n : std_logic;
    signal w_ctrl_078, w_ctrl_078_n : std_logic;
237 signal w_ctrl_079, w_ctrl_079_n : std_logic;
    signal w_ctrl_080, w_ctrl_080_n : std_logic;
239 signal w_ctrl_081 : std_logic;
    signal w_ctrl_082, w_ctrl_082_n : std_logic;
241 signal w_ctrl_083 : std_logic;
    signal w_ctrl_084 : std_logic;
243 signal w_ctrl_085, w_ctrl_085_n : std_logic;
    signal w_ctrl_086, w_ctrl_086_n : std_logic;
245 signal w_ctrl_087, w_ctrl_087_n : std_logic;
    signal w_ctrl_094 : std_logic;
247 signal w_ctrl_095 : std_logic;
    signal w_ctrl_098 : std_logic;
249 signal w_ctrl_099 : std_logic;
    signal w_ctrl_100 : std_logic;
251 signal w_ctrl_101 : std_logic;
    signal w_ctrl_102, w_ctrl_102_n : std_logic;
253 signal w_ctrl_103 : std_logic;
    signal w_ctrl_106, w_ctrl_106_n : std_logic;
255 signal w_ctrl_107 : std_logic;
    signal w_ctrl_108 : std_logic;
```

```
257   signal w_ctrl_109 : std_logic;
      signal w_ctrl_110 : std_logic;
259   signal w_ctrl_111 : std_logic;
      signal w_ctrl_112 : std_logic;
261   signal w_ctrl_113 : std_logic;
      signal w_ctrl_114 : std_logic;
263   signal w_ctrl_115 : std_logic;
      signal w_ctrl_116 : std_logic;
265   signal w_ctrl_117 : std_logic;
      signal w_ctrl_118, w_ctrl_118_n : std_logic;
267   signal w_ctrl_119 : std_logic;
      signal w_ctrl_120, w_ctrl_120_n : std_logic;
269   signal w_ctrl_121 : std_logic;
      signal w_ctrl_122 : std_logic;
271   signal w_ctrl_123 : std_logic;
      signal w_ctrl_124 : std_logic;
273   signal w_ctrl_125, w_ctrl_125_n : std_logic;
      signal w_ctrl_126, w_ctrl_126_n : std_logic;
275   signal w_ctrl_127, w_ctrl_127_n : std_logic;
      signal w_ctrl_128 : std_logic;
277   signal w_ctrl_129 : std_logic;
      signal w_ctrl_130, w_ctrl_130_n : std_logic;
279   signal w_ctrl_131 : std_logic;
      signal w_ctrl_132, w_ctrl_132_n : std_logic;
281   signal w_ctrl_133 : std_logic;
      signal w_ctrl_134 : std_logic;
283   signal w_ctrl_135 : std_logic;
      signal w_ctrl_136 : std_logic;
285   signal w_ctrl_137, w_ctrl_137_n : std_logic;
      signal w_ctrl_138 : std_logic;
287   signal w_ctrl_139 : std_logic;
      signal w_ctrl_140 : std_logic;
289   signal w_ctrl_141 : std_logic;
      signal w_ctrl_142 : std_logic;
291   signal w_ctrl_143, w_ctrl_143_n : std_logic;
      signal w_ctrl_144, w_ctrl_144_n : std_logic;
293   signal w_ctrl_145 : std_logic;
      signal w_ctrl_146 : std_logic;
295   signal w_ctrl_147 : std_logic;
      signal w_ctrl_148 : std_logic;
297
299
```

```
begin
  --| Instantiate Inverters to create inverted signals where
   needed
  u_myINV_31 : myINV
  port map (
    i_A => i_instr31,
    o_Z => w_instr31_n);

  u_myINV_30 : myINV
  port map (
    i_A => i_instr30,
    o_Z => w_instr30_n);

  u_myINV_29 : myINV
  port map (
    i_A => i_instr29,
    o_Z => w_instr29_n);

  u_myINV_28 : myINV
  port map (
    i_A => i_instr28,
    o_Z => w_instr28_n);

  u_myINV_27 : myINV
  port map (
    i_A => i_instr27,
    o_Z => w_instr27_n);

  u_myINV_26 : myINV
  port map (
    i_A => i_instr26,
    o_Z => w_instr26_n);

  u_myINV_20 : myINV
  port map (
    i_A => i_instr20,
    o_Z => w_instr20_n);

  u_myINV_19 : myINV
  port map (
    i_A => i_instr19,
    o_Z => w_instr19_n);

```

```
  u_myINV_18 : myINV
343 port map (
    i_A => i_instr18,
345 o_Z => w_instr18_n);

347 u_myINV_17 : myINV
  port map (
349 i_A => i_instr17,
    o_Z => w_instr17_n);
351
  u_myINV_16 : myINV
353 port map (
    i_A => i_instr16,
355 o_Z => w_instr16_n);

357 u_myINV_05 : myINV
  port map (
359 i_A => i_instr05,
    o_Z => w_instr05_n);
361
  u_myINV_04 : myINV
363 port map (
    i_A => i_instr04,
365 o_Z => w_instr04_n);
  u_myINV_03 : myINV
367 port map (
    i_A => i_instr03,
369 o_Z => w_instr03_n);

371 u_myINV_02 : myINV
  port map (
373 i_A => i_instr02,
    o_Z => w_instr02_n);
375
  u_myINV_01 : myINV
377 port map (
    i_A => i_instr01,
379 o_Z => w_instr01_n);

381 u_myINV_00 : myINV
  port map (
383 i_A => i_instr00,
    o_Z => w_instr00_n);
```

```
385
    u_myINV_000 : myINV
387 port map (
      i_A => w_ctrl_000,
389   o_Z => w_ctrl_000_n);

391 u_myINV_001 : myINV
    port map (
393   i_A => w_ctrl_001,
      o_Z => w_ctrl_001_n);
395
    u_myINV_002 : myINV
397 port map (
      i_A => w_ctrl_002,
399   o_Z => w_ctrl_002_n);

401 u_myINV_003 : myINV
    port map (
403   i_A => w_ctrl_003,
      o_Z => w_ctrl_003_n);
405
    u_myINV_004 : myINV
407 port map (
      i_A => w_ctrl_004,
409   o_Z => w_ctrl_004_n);

411 u_myINV_005 : myINV
    port map (
413   i_A => w_ctrl_005,
      o_Z => w_ctrl_005_n);
415
    u_myINV_006 : myINV
417 port map (
      i_A => w_ctrl_006,
419   o_Z => w_ctrl_006_n);

421 u_myINV_007 : myINV
    port map (
423   i_A => w_ctrl_007,
      o_Z => w_ctrl_007_n);
425
    u_myINV_009 : myINV
427 port map (
```

```
    i_A => w_ctrl_009,
    o_Z => w_ctrl_009_n);

  u_myINV_011 : myINV
  port map (
    i_A => w_ctrl_011,
    o_Z => w_ctrl_011_n);

  u_myINV_014 : myINV
  port map (
    i_A => w_ctrl_014,
    o_Z => w_ctrl_014_n);

  u_myINV_015 : myINV
  port map (
    i_A => w_ctrl_015,
    o_Z => w_ctrl_015_n);

  u_myINV_018 : myINV
  port map (
    i_A => w_ctrl_018,
    o_Z => w_ctrl_018_n);

  u_myINV_020 : myINV
  port map (
    i_A => w_ctrl_020,
    o_Z => w_ctrl_020_n);

  u_myINV_028 : myINV
  port map (
    i_A => w_ctrl_028,
    o_Z => w_ctrl_028_n);

  u_myINV_029 : myINV
  port map (
    i_A => w_ctrl_029,
    o_Z => w_ctrl_029_n);

  u_myINV_034 : myINV
  port map (
    i_A => w_ctrl_034,
    o_Z => w_ctrl_034_n);
```

```
471   u_myINV_037 : myINV
      port map (
473     i_A => w_ctrl_037,
        o_Z => w_ctrl_037_n);
475
      u_myINV_038 : myINV
477   port map (
        i_A => w_ctrl_038,
479     o_Z => w_ctrl_038_n);

481   u_myINV_039 : myINV
      port map (
483     i_A => w_ctrl_039,
        o_Z => w_ctrl_039_n);
485
      u_myINV_041 : myINV
487   port map (
        i_A => w_ctrl_041,
489     o_Z => w_ctrl_041_n);

491   u_myINV_043 : myINV
      port map (
493     i_A => w_ctrl_043,
        o_Z => w_ctrl_043_n);
495
      u_myINV_044 : myINV
497   port map (
        i_A => w_ctrl_044,
499     o_Z => w_ctrl_044_n);

501   u_myINV_046 : myINV
      port map (
503     i_A => w_ctrl_046,
        o_Z => w_ctrl_046_n);
505
      u_myINV_048 : myINV
507   port map (
        i_A => w_ctrl_048,
509     o_Z => w_ctrl_048_n);

511   u_myINV_051 : myINV
      port map (
513     i_A => w_ctrl_051,
```

```
    o_Z => w_ctrl_051_n);

  u_myINV_053 : myINV
  port map (
    i_A => w_ctrl_053,
    o_Z => w_ctrl_053_n);

  u_myINV_054 : myINV
  port map (
    i_A => w_ctrl_054,
    o_Z => w_ctrl_054_n);

  u_myINV_055 : myINV
  port map (
    i_A => w_ctrl_055,
    o_Z => w_ctrl_055_n);

  u_myINV_056 : myINV
  port map (
    i_A => w_ctrl_056,
    o_Z => w_ctrl_056_n);

  u_myINV_057 : myINV
  port map (
    i_A => w_ctrl_057,
    o_Z => w_ctrl_057_n);

  u_myINV_058 : myINV
  port map (
    i_A => w_ctrl_058,
    o_Z => w_ctrl_058_n);

  u_myINV_062 : myINV
  port map (
    i_A => w_ctrl_062,
    o_Z => w_ctrl_062_n);

  u_myINV_063 : myINV
  port map (
    i_A => w_ctrl_063,
    o_Z => w_ctrl_063_n);

  u_myINV_065 : myINV
```

```
557   port map (
        i_A => w_ctrl_065,
559     o_Z => w_ctrl_065_n);

561   u_myINV_066 : myINV
      port map (
563     i_A => w_ctrl_066,
        o_Z => w_ctrl_066_n);
565
      u_myINV_070 : myINV
567   port map (
        i_A => w_ctrl_070,
569     o_Z => w_ctrl_070_n);

571   u_myINV_071 : myINV
      port map (
573     i_A => w_ctrl_071,
        o_Z => w_ctrl_071_n);
575
      u_myINV_073 : myINV
577   port map (
        i_A => w_ctrl_073,
579     o_Z => w_ctrl_073_n);

581   u_myINV_074 : myINV
      port map (
583     i_A => w_ctrl_074,
        o_Z => w_ctrl_074_n);
585
      u_myINV_077 : myINV
587   port map (
        i_A => w_ctrl_077,
589     o_Z => w_ctrl_077_n);

591   u_myINV_078 : myINV
      port map (
593     i_A => w_ctrl_078,
        o_Z => w_ctrl_078_n);
595
      u_myINV_079 : myINV
597   port map (
        i_A => w_ctrl_079,
599     o_Z => w_ctrl_079_n);
```

```
601   u_myINV_080 : myINV
      port map (
603     i_A => w_ctrl_080,
        o_Z => w_ctrl_080_n);
605
      u_myINV_082 : myINV
607   port map (
        i_A => w_ctrl_082,
609     o_Z => w_ctrl_082_n);

611   u_myINV_085 : myINV
      port map (
613     i_A => w_ctrl_085,
        o_Z => w_ctrl_085_n);
615
      u_myINV_086 : myINV
617   port map (
        i_A => w_ctrl_086,
619     o_Z => w_ctrl_086_n);

621   u_myINV_087 : myINV
      port map (
623     i_A => w_ctrl_087,
        o_Z => w_ctrl_087_n);
625
      u_myINV_102 : myINV
627   port map (
        i_A => w_ctrl_102,
629     o_Z => w_ctrl_102_n);

631   u_myINV_106 : myINV
      port map (
633     i_A => w_ctrl_106,
        o_Z => w_ctrl_106_n);
635
      u_myINV_118 : myINV
637   port map (
        i_A => w_ctrl_118,
639     o_Z => w_ctrl_118_n);

641   u_myINV_120 : myINV
      port map (
```

```
643     i_A => w_ctrl_120,
        o_Z => w_ctrl_120_n);
645
    u_myINV_125 : myINV
647 port map (
        i_A => w_ctrl_125,
649     o_Z => w_ctrl_125_n);

651 u_myINV_126 : myINV
    port map (
653     i_A => w_ctrl_126,
        o_Z => w_ctrl_126_n);
655
    u_myINV_127 : myINV
657 port map (
        i_A => w_ctrl_127,
659     o_Z => w_ctrl_127_n);

661 u_myINV_130 : myINV
    port map (
663     i_A => w_ctrl_130,
        o_Z => w_ctrl_130_n);
665
    u_myINV_132 : myINV
667 port map (
        i_A => w_ctrl_132,
669     o_Z => w_ctrl_132_n);

671 u_myINV_137 : myINV
    port map (
673     i_A => w_ctrl_137,
        o_Z => w_ctrl_137_n);
675
    u_myINV_143 : myINV
677 port map (
        i_A => w_ctrl_143,
679     o_Z => w_ctrl_143_n);

681 u_myINV_144 : myINV
    port map (
683     i_A => w_ctrl_144,
        o_Z => w_ctrl_144_n);
685
```

```
  --| Instantiate NAND gates to create intermediate control
   signals
687 u_myNAND_000 : myNAND2
    port map (
689   i_A => w_instr31_n,
      i_B => w_instr30_n,
691   o_Z => w_ctrl_000);

693 u_myNAND_001 : myNAND2
    port map (
695   i_A => w_instr29_n,
      i_B => w_instr28_n,
697   o_Z => w_ctrl_001);

699 u_myNAND_002 : myNAND2
    port map (
701   i_A => w_instr27_n,
      i_B => w_instr26_n,
703   o_Z => w_ctrl_002);

705 u_myNAND_003 : myNAND2
    port map (
707   i_A => w_instr05_n,
      i_B => w_instr04_n,
709   o_Z => w_ctrl_003);

711 u_myNAND_004 : myNAND2
    port map (
713   i_A => w_instr03_n,
      i_B => w_instr02_n,
715   o_Z => w_ctrl_004);

717 u_myNAND_005 : myNAND2
    port map (
719   i_A => w_ctrl_000_n,
      i_B => w_ctrl_001_n,
721   o_Z => w_ctrl_005);

723 u_myNAND_006 : myNAND2
    port map (
725   i_A => w_ctrl_002_n,
      i_B => w_ctrl_003_n,
727   o_Z => w_ctrl_006);
```

```
729   u_myNAND_007 : myNAND2
      port map (
731     i_A => w_ctrl_005_n,
        i_B => w_ctrl_006_n,
733     o_Z => w_ctrl_007);

735   u_myNAND_008 : myNAND2
      port map (
737     i_A => w_ctrl_007_n,
        i_B => w_ctrl_004_n,
739     o_Z => w_ctrl_008);

741   u_myNAND_009 : myNAND2
      port map (
743     i_A => w_ctrl_000_n,
        i_B => w_instr29_n,
745     o_Z => w_ctrl_009);

747   u_myNAND_010 : myNAND2
      port map (
749     i_A => w_instr28_n,
        i_B => w_ctrl_002_n,
751     o_Z => w_ctrl_010);

753   u_myNAND_011 : myNAND2
      port map (
755     i_A => i_instr28,
        i_B => w_instr27_n,
757     o_Z => w_ctrl_011);

759   u_myNAND_012 : myNAND2
      port map (
761     i_A => w_ctrl_010,
        i_B => w_ctrl_011,
763     o_Z => w_ctrl_012);

765   u_myNAND_013 : myNAND2
      port map (
767     i_A => w_ctrl_009_n,
        i_B => w_ctrl_012,
769     o_Z => w_ctrl_013);
```

```
771   u_myNAND_014 : myNAND2
      port map (
773     i_A => w_instr29_n,
        i_B => w_instr27_n,
775     o_Z => w_ctrl_014);

777   u_myNAND_015 : myNAND2
      port map (
779     i_A => w_instr29_n,
        i_B => w_ctrl_011_n,
781     o_Z => w_ctrl_015);

783   u_myNAND_018 : myNAND2
      port map (
785     i_A => w_instr26_n,
        i_B => i_instr05,
787     o_Z => w_ctrl_018);

789   u_myNAND_020 : myNAND2
      port map (
791     i_A => w_ctrl_018_n,
        i_B => w_instr04_n,
793     o_Z => w_ctrl_020);

795   u_myNAND_025 : myNAND2
      port map (
797     i_A => w_instr27_n,
        i_B => i_instr26,
799     o_Z => w_ctrl_025);

801   u_myNAND_028 : myNAND2
      port map (
803     i_A => w_ctrl_000_n,
        i_B => w_instr28_n,
805     o_Z => w_ctrl_028);

807   u_myNAND_029 : myNAND2
      port map (
809     i_A => i_instr29,
        i_B => i_instr27,
811     o_Z => w_ctrl_029);

813   u_myNAND_030 : myNAND2
```

```
    port map (
815   i_A => i_instr26,
      i_B => i_instr16,
817   o_Z => w_ctrl_030);

819 u_myNAND_031 : myNAND2
    port map (
821   i_A => w_ctrl_014_n,
      i_B => w_ctrl_030,
823   o_Z => w_ctrl_031);

825 u_myNAND_032 : myNAND2
    port map (
827   i_A => w_ctrl_029,
      i_B => w_ctrl_031,
829   o_Z => w_ctrl_032);

831 u_myNAND_033 : myNAND2
    port map (
833   i_A => w_ctrl_028_n,
      i_B => w_ctrl_032,
835   o_Z => w_ctrl_033);

837 u_myNAND_034 : myNAND2
    port map (
839   i_A => w_ctrl_000_n,
      i_B => w_ctrl_015_n,
841   o_Z => w_ctrl_034);

843 u_myNAND_035 : myNAND2
    port map (
845   i_A => w_ctrl_034_n,
      i_B => w_instr26_n,
847   o_Z => w_ctrl_035);

849 u_myNAND_036 : myNAND2
    port map (
851   i_A => w_ctrl_034_n,
      i_B => i_instr26,
853   o_Z => w_ctrl_036);

855 u_myNAND_037 : myNAND2
    port map (
```

```
857       i_A => w_instr29_n,
          i_B => i_instr28,
859       o_Z => w_ctrl_037);

861   u_myNAND_038 : myNAND2
      port map (
863       i_A => i_instr27,
          i_B => w_instr26_n,
865       o_Z => w_ctrl_038);

867   u_myNAND_039 : myNAND2
      port map (
869       i_A => w_ctrl_000_n,
          i_B => w_ctrl_037_n,
871       o_Z => w_ctrl_039);

873   u_myNAND_040 : myNAND2
      port map (
875       i_A => w_ctrl_039_n,
          i_B => w_ctrl_038_n,
877       o_Z => w_ctrl_040);

879   u_myNAND_041 : myNAND2
      port map (
881       i_A => i_instr27,
          i_B => i_instr26,
883       o_Z => w_ctrl_041);

885   u_myNAND_042 : myNAND2
      port map (
887       i_A => w_ctrl_039_n,
          i_B => w_ctrl_041_n,
889       o_Z => w_ctrl_042);

891   u_myNAND_043 : myNAND2
      port map (
893       i_A => w_instr01_n,
          i_B => w_instr00_n,
895       o_Z => w_ctrl_043);

897   u_myNAND_044 : myNAND2
      port map (
899       i_A => w_ctrl_004_n,
```

```
    i_B => w_ctrl_043_n,
901 o_Z => w_ctrl_044);

903 u_myNAND_046 : myNAND2
    port map (
905 i_A => w_instr03_n,
    i_B => i_instr01,
907 o_Z => w_ctrl_046);

909 u_myNAND_047 : myNAND2
    port map (
911 i_A => w_ctrl_007_n,
    i_B => w_ctrl_046_n,
913 o_Z => w_ctrl_047);

915 u_myNAND_048 : myNAND2
    port map (
917 i_A => w_instr30_n,
    i_B => w_instr27_n,
919 o_Z => w_ctrl_048);

921 u_myNAND_049 : myNAND2
    port map (
923 i_A => w_ctrl_020_n,
    i_B => w_ctrl_004_n,
925 o_Z => w_ctrl_049);

927 u_myNAND_050 : myNAND2
    port map (
929 i_A => w_instr29_n,
    i_B => w_ctrl_049,
931 o_Z => w_ctrl_050);

933 u_myNAND_051 : myNAND2
    port map (
935 i_A => w_instr28_n,
    i_B => w_ctrl_048_n,
937 o_Z => w_ctrl_051);

939 u_myNAND_052 : myNAND2
    port map (
941 i_A => w_ctrl_051_n,
    i_B => w_ctrl_050,
```

```
943     o_Z => w_ctrl_052);

945   u_myNAND_053 : myNAND2
      port map (
947     i_A => w_ctrl_000_n,
        i_B => w_ctrl_002_n,
949     o_Z => w_ctrl_053);

951   u_myNAND_054 : myNAND2
      port map (
953     i_A => i_instr29,
        i_B => i_instr28,
955     o_Z => w_ctrl_054);

957   u_myNAND_055 : myNAND2
      port map (
959     i_A => i_instr05,
        i_B => w_instr04_n,
961     o_Z => w_ctrl_055);

963   u_myNAND_056 : myNAND2
      port map (
965     i_A => w_instr03_n,
        i_B => i_instr02,
967     o_Z => w_ctrl_056);

969   u_myNAND_057 : myNAND2
      port map (
971     i_A => w_ctrl_001_n,
        i_B => w_ctrl_055_n,
973     o_Z => w_ctrl_057);

975   u_myNAND_058 : myNAND2
      port map (
977     i_A => w_ctrl_056_n,
        i_B => w_ctrl_043_n,
979     o_Z => w_ctrl_058);

981   u_myNAND_059 : myNAND2
      port map (
983     i_A => w_ctrl_057_n,
        i_B => w_ctrl_058_n,
985     o_Z => w_ctrl_059);
```

```
987   u_myNAND_060 : myNAND2
      port map (
989     i_A => w_ctrl_054,
        i_B => w_ctrl_059,
991     o_Z => w_ctrl_060);

993   u_myNAND_061 : myNAND2
      port map (
995     i_A => w_ctrl_053_n,
        i_B => w_ctrl_060,
997     o_Z => w_ctrl_061);

999   u_myNAND_062 : myNAND2
      port map (
1001    i_A => w_instr01_n,
        i_B => i_instr00,
1003    o_Z => w_ctrl_062);

1005  u_myNAND_063 : myNAND2
      port map (
1007    i_A => w_ctrl_000_n,
        i_B => w_instr27_n,
1009    o_Z => w_ctrl_063);

1011  u_myNAND_064 : myNAND2
      port map (
1013    i_A => w_ctrl_054_n,
        i_B => i_instr26,
1015    o_Z => w_ctrl_064);

1017  u_myNAND_065 : myNAND2
      port map (
1019    i_A => w_ctrl_056_n,
        i_B => w_ctrl_062_n,
1021    o_Z => w_ctrl_065);

1023  u_myNAND_066 : myNAND2
      port map (
1025    i_A => w_ctrl_057_n,
        i_B => w_instr26_n,
1027    o_Z => w_ctrl_066);
```

```
1029    u_myNAND_067 : myNAND2
        port map (
1031      i_A => w_ctrl_066_n,
          i_B => w_ctrl_065_n,
1033      o_Z => w_ctrl_067);

1035    u_myNAND_068 : myNAND2
        port map (
1037      i_A => w_ctrl_064,
          i_B => w_ctrl_067,
1039      o_Z => w_ctrl_068);

1041    u_myNAND_069 : myNAND2
        port map (
1043      i_A => w_ctrl_063_n,
          i_B => w_ctrl_068,
1045      o_Z => w_ctrl_069);

1047    u_myNAND_070 : myNAND2
        port map (
1049      i_A => i_instr01,
          i_B => w_instr00_n,
1051      o_Z => w_ctrl_070);

1053    u_myNAND_071 : myNAND2
        port map (
1055      i_A => w_ctrl_000_n,
          i_B => w_instr26_n,
1057      o_Z => w_ctrl_071);

1059    u_myNAND_072 : myNAND2
        port map (
1061      i_A => w_ctrl_054_n,
          i_B => i_instr27,
1063      o_Z => w_ctrl_072);

1065    u_myNAND_073 : myNAND2
        port map (
1067      i_A => w_ctrl_056_n,
          i_B => w_ctrl_070_n,
1069      o_Z => w_ctrl_073);

1071    u_myNAND_074 : myNAND2
```

```
    port map (
1073  i_A => w_ctrl_057_n,
      i_B => w_ctrl_073_n,
1075  o_Z => w_ctrl_074);

1077  u_myNAND_075 : myNAND2
    port map (
1079  i_A => w_ctrl_072,
      i_B => w_ctrl_148,
1081  o_Z => w_ctrl_075);

1083  u_myNAND_076 : myNAND2
    port map (
1085  i_A => w_ctrl_071_n,
      i_B => w_ctrl_075,
1087  o_Z => w_ctrl_076);

1089  u_myNAND_077 : myNAND2
    port map (
1091  i_A => i_instr01,
      i_B => i_instr00,
1093  o_Z => w_ctrl_077);

1095  u_myNAND_078 : myNAND2
    port map (
1097  i_A => w_ctrl_002_n,
      i_B => w_ctrl_055_n,
1099  o_Z => w_ctrl_078);

1101  u_myNAND_079 : myNAND2
    port map (
1103  i_A => w_ctrl_056_n,
      i_B => w_ctrl_077_n,
1105  o_Z => w_ctrl_079);

1107  u_myNAND_080 : myNAND2
    port map (
1109  i_A => w_ctrl_005_n,
      i_B => w_ctrl_078_n,
1111  o_Z => w_ctrl_080);

1113  u_myNAND_081 : myNAND2
    port map (
```

```
1115      i_A => w_ctrl_080_n,
          i_B => w_ctrl_079_n,
1117      o_Z => w_ctrl_081);

1119    u_myNAND_082 : myNAND2
        port map (
1121      i_A => w_ctrl_000_n,
          i_B => w_ctrl_054_n,
1123      o_Z => w_ctrl_082);

1125    u_myNAND_083 : myNAND2
        port map (
1127      i_A => w_ctrl_082_n,
          i_B => w_ctrl_041_n,
1129      o_Z => w_ctrl_083);

1131    u_myNAND_084 : myNAND2
        port map (
1133      i_A => w_ctrl_029_n,
          i_B => w_instr28_n,
1135      o_Z => w_ctrl_084);

1137    u_myNAND_085 : myNAND2
        port map (
1139      i_A => i_instr03,
          i_B => w_instr02_n,
1141      o_Z => w_ctrl_085);

1143    u_myNAND_086 : myNAND2
        port map (
1145      i_A => w_ctrl_055_n,
          i_B => w_ctrl_085_n,
1147      o_Z => w_ctrl_086);

1149    u_myNAND_087 : myNAND2
        port map (
1151      i_A => w_instr26_n,
          i_B => i_instr01,
1153      o_Z => w_ctrl_087);

1155    u_myNAND_094 : myNAND2
        port map (
1157      i_A => w_ctrl_005_n,
```

```
    i_B => w_ctrl_002_n,
1159    o_Z => w_ctrl_094);

1161  u_myNAND_095 : myNAND2
    port map (
1163    i_A => i_instr31,
    i_B => i_instr29,
1165    o_Z => w_ctrl_095);

1167  u_myNAND_098 : myNAND2
    port map (
1169    i_A => i_instr31,
    i_B => w_instr29_n,
1171    o_Z => w_ctrl_098);

1173  u_myNAND_099 : myNAND2
    port map (
1175    i_A => w_ctrl_000_n,
    i_B => i_instr29,
1177    o_Z => w_ctrl_099);

1179  u_myNAND_100 : myNAND2
    port map (
1181    i_A => w_instr28_n,
    i_B => w_ctrl_025,
1183    o_Z => w_ctrl_100);

1185  u_myNAND_101 : myNAND2
    port map (
1187    i_A => w_ctrl_009_n,
    i_B => w_ctrl_100,
1189    o_Z => w_ctrl_101);

1191  u_myNAND_102 : myNAND2
    port map (
1193    i_A => i_instr29,
    i_B => w_instr28_n,
1195    o_Z => w_ctrl_102);

1197  u_myNAND_103 : myNAND2
    port map (
1199    i_A => w_ctrl_053_n,
    i_B => w_ctrl_102_n,
```

```
1201      o_Z => w_ctrl_103);

1203    u_myNAND_106 : myNAND2
        port map (
1205      i_A => w_ctrl_014_n,
          i_B => w_ctrl_087_n,
1207      o_Z => w_ctrl_106);

1209    u_myNAND_107 : myNAND2
        port map (
1211      i_A => w_ctrl_086_n,
          i_B => w_ctrl_106_n,
1213      o_Z => w_ctrl_107);

1215    u_myNAND_108 : myNAND2
        port map (
1217      i_A => w_ctrl_029,
          i_B => w_ctrl_107,
1219      o_Z => w_ctrl_108);

1221    u_myNAND_109 : myNAND2
        port map (
1223      i_A => w_ctrl_028_n,
          i_B => w_ctrl_108,
1225      o_Z => w_ctrl_109);

1227    u_myNAND_110 : myNAND2
        port map (
1229      i_A => w_instr02_n,
          i_B => i_instr01,
1231      o_Z => w_ctrl_110);

1233    u_myNAND_111 : myNAND2
        port map (
1235      i_A => w_instr03_n,
          i_B => w_ctrl_110,
1237      o_Z => w_ctrl_111);

1239    u_myNAND_112 : myNAND2
        port map (
1241      i_A => w_ctrl_020_n,
          i_B => w_ctrl_111,
1243      o_Z => w_ctrl_112);
```

```
1245  u_myNAND_113 : myNAND2
      port map (
1247    i_A => w_ctrl_028_n,
        i_B => w_ctrl_112,
1249    o_Z => w_ctrl_113);

1251  u_myNAND_114 : myNAND2
      port map (
1253    i_A => w_ctrl_014_n,
        i_B => w_ctrl_113,
1255    o_Z => w_ctrl_114);

1257  u_myNAND_115 : myNAND2
      port map (
1259    i_A => w_ctrl_084,
        i_B => w_ctrl_114,
1261    o_Z => w_ctrl_115);

1263  u_myNAND_116 : myNAND2
      port map (
1265    i_A => w_ctrl_000_n,
        i_B => w_ctrl_115,
1267    o_Z => w_ctrl_116);

1269  u_myNAND_117 : myNAND2
      port map (
1271    i_A => w_ctrl_080_n,
        i_B => w_ctrl_044_n,
1273    o_Z => w_ctrl_117);

1275  u_myNAND_118 : myNAND2
      port map (
1277    i_A => w_ctrl_004_n,
        i_B => w_ctrl_070_n,
1279    o_Z => w_ctrl_118);

1281  u_myNAND_119 : myNAND2
      port map (
1283    i_A => w_ctrl_080_n,
        i_B => w_ctrl_118_n,
1285    o_Z => w_ctrl_119);
```

```
1287    u_myNAND_120 : myNAND2
        port map (
1289      i_A => w_ctrl_003_n,
          i_B => w_instr03_n,
1291      o_Z => w_ctrl_120);

1293    u_myNAND_121 : myNAND2
        port map (
1295      i_A => i_instr03,
          i_B => w_ctrl_110,
1297      o_Z => w_ctrl_121);

1299    u_myNAND_122 : myNAND2
        port map (
1301      i_A => w_ctrl_055_n,
          i_B => w_ctrl_121,
1303      o_Z => w_ctrl_122);

1305    u_myNAND_123 : myNAND2
        port map (
1307      i_A => w_ctrl_120_n,
          i_B => w_ctrl_062,
1309      o_Z => w_ctrl_123);

1311    u_myNAND_124 : myNAND2
        port map (
1313      i_A => w_ctrl_122,
          i_B => w_ctrl_123,
1315      o_Z => w_ctrl_124);

1317    u_myNAND_125 : myNAND2
        port map (
1319      i_A => w_instr20_n,
          i_B => w_instr19_n,
1321      o_Z => w_ctrl_125);

1323    u_myNAND_126 : myNAND2
        port map (
1325      i_A => w_instr18_n,
          i_B => w_instr17_n,
1327      o_Z => w_ctrl_126);

1329    u_myNAND_127 : myNAND2
```

```
    port map (
1331    i_A => w_ctrl_125_n,
        i_B => w_ctrl_126_n,
1333    o_Z => w_ctrl_127);

1335  u_myNAND_128 : myNAND2
    port map (
1337    i_A => w_ctrl_127_n,
        i_B => w_instr16_n,
1339    o_Z => w_ctrl_128);

1341  u_myNAND_129 : myNAND2
    port map (
1343    i_A => i_instr27,
        i_B => w_ctrl_128,
1345    o_Z => w_ctrl_129);

1347  u_myNAND_130 : myNAND2
    port map (
1349    i_A => i_instr31,
        i_B => w_instr28_n,
1351    o_Z => w_ctrl_130);

1353  u_myNAND_131 : myNAND2
    port map (
1355    i_A => w_ctrl_041_n,
        i_B => w_ctrl_130_n,
1357    o_Z => w_ctrl_131);

1359  u_myNAND_132 : myNAND2
    port map (
1361    i_A => w_instr27_n,
        i_B => w_instr28_n,
1363    o_Z => w_ctrl_132);

1365  u_myNAND_133 : myNAND2
    port map (
1367    i_A => i_instr28,
        i_B => w_ctrl_129,
1369    o_Z => w_ctrl_133);

1371  u_myNAND_134 : myNAND2
    port map (
```

```
1373      i_A => w_instr26_n,
          i_B => w_ctrl_124,
1375      o_Z => w_ctrl_134);

1377    u_myNAND_135 : myNAND2
        port map (
1379      i_A => i_instr26,
          i_B => w_ctrl_127_n,
1381      o_Z => w_ctrl_135);

1383    u_myNAND_136 : myNAND2
        port map (
1385      i_A => w_ctrl_134,
          i_B => w_ctrl_135,
1387      o_Z => w_ctrl_136);

1389    u_myNAND_137 : myNAND2
        port map (
1391      i_A => w_instr29_n,
          i_B => w_ctrl_133,
1393      o_Z => w_ctrl_137);

1395    u_myNAND_138 : myNAND2
        port map (
1397      i_A => w_ctrl_132_n,
          i_B => w_ctrl_136,
1399      o_Z => w_ctrl_138);

1401    u_myNAND_139 : myNAND2
        port map (
1403      i_A => w_ctrl_137_n,
          i_B => w_ctrl_138,
1405      o_Z => w_ctrl_139);

1407    u_myNAND_140 : myNAND2
        port map (
1409      i_A => w_instr31_n,
          i_B => w_ctrl_139,
1411      o_Z => w_ctrl_140);

1413    u_myNAND_141 : myNAND2
        port map (
1415      i_A => w_ctrl_131,
```

```
    i_B => w_ctrl_140,
1417    o_Z => w_ctrl_141);

1419  u_myNAND_142 : myNAND2
    port map (
1421    i_A => w_ctrl_029_n,
    i_B => i_instr26,
1423    o_Z => w_ctrl_142);

1425  u_myNAND_143 : myNAND2
    port map (
1427    i_A => w_ctrl_014_n,
    i_B => w_instr26_n,
1429    o_Z => w_ctrl_143);

1431  u_myNAND_144 : myNAND2
    port map (
1433    i_A => w_ctrl_143_n,
    i_B => w_ctrl_086_n,
1435    o_Z => w_ctrl_144);

1437  u_myNAND_145 : myNAND2
    port map (
1439    i_A => w_ctrl_144_n,
    i_B => w_ctrl_077_n,
1441    o_Z => w_ctrl_145);

1443  u_myNAND_146 : myNAND2
    port map (
1445    i_A => w_ctrl_142,
    i_B => w_ctrl_145,
1447    o_Z => w_ctrl_146);

1449  u_myNAND_147 : myNAND2
    port map (
1451    i_A => w_ctrl_028_n,
    i_B => w_ctrl_146,
1453    o_Z => w_ctrl_147);

1455  u_myNAND_148 : myNAND2
    port map (
1457    i_A => w_ctrl_074_n,
    i_B => w_instr27_n,
```

```
1459      o_Z => w_ctrl_148);

1461    --| Instantiate Inverters and NAND gates to generate output
          signals
        u_myINV_ALU_SRC_A0 : myINV
1463    port map (
          i_A => w_ctrl_008,
1465      o_Z => o_ALU_SRC_A0);

1467    u_myNAND_ALU_SRC_B1 : myNAND2
        port map (
1469      i_A => w_instr31_n,
          i_B => w_instr29_n,
1471      o_Z => o_ALU_SRC_B(1));

1473    u_myINV_ALU_SRC_B0 : myINV
        port map (
1475      i_A => w_ctrl_013,
          o_Z => o_ALU_SRC_B(0));
1477
        u_myINV_INV_B : myINV
1479    port map (
          i_A => w_ctrl_116,
1481      o_Z => o_INV_B);

1483    u_myINV_LTZ : myINV
        port map (
1485      i_A => w_ctrl_033,
          o_Z => o_LTZ);
1487
        u_myINV_EQ : myINV
1489    port map (
          i_A => w_ctrl_035,
1491      o_Z => o_EQ);

1493    u_myINV_NE : myINV
        port map (
1495      i_A => w_ctrl_036,
          o_Z => o_NE);
1497
        u_myINV_LEZ : myINV
1499    port map (
          i_A => w_ctrl_040,
```

```
1501      o_Z => o_LEZ);

1503    u_myINV_GTZ : myINV
        port map (
1505      i_A => w_ctrl_042,
          o_Z => o_GTZ);
1507
        u_myINV_RIGHT_SHIFT : myINV
1509    port map (
          i_A => w_ctrl_047,
1511      o_Z => o_RIGHT_SHIFT);

1513    u_myNAND_ADDER : myNAND2
        port map (
1515      i_A => w_instr31_n,
          i_B => w_ctrl_052,
1517      o_Z => o_ADDER);

1519    u_myINV_AND : myINV
        port map (
1521      i_A => w_ctrl_061,
          o_Z => o_AND);
1523
        u_myINV_OR : myINV
1525    port map (
          i_A => w_ctrl_069,
1527      o_Z => o_OR);

1529    u_myINV_XOR : myINV
        port map (
1531      i_A => w_ctrl_076,
          o_Z => o_XOR);
1533
        u_myINV_NOR : myINV
1535    port map (
          i_A => w_ctrl_081,
1537      o_Z => o_NOR);

1539    u_myINV_COMPARE : myINV
        port map (
1541      i_A => w_ctrl_109,
          o_Z => o_COMPARE);
1543
```

```
    u_myINV_LUI : myINV
1545 port map (
      i_A => w_ctrl_083,
1547  o_Z => o_LUI);

1549 u_myINV_REG_SEL0 : myINV
    port map (
1551  i_A => w_ctrl_094,
      o_Z => o_REG_SEL(0));
1553
    u_myNAND_REG_SEL1 : myNAND2
1555 port map (
      i_A => w_ctrl_098,
1557  i_B => w_ctrl_099,
      o_Z => o_REG_SEL(1));
1559
    u_myNAND_OVER_CTRL0 : myNAND2
1561 port map (
      i_A => w_ctrl_117,
1563  i_b => w_ctrl_103,
      o_Z => o_OVER_CTRL(0));
1565
    u_myNAND_OVER_CTRL1 : myNAND2
1567 port map (
      i_A => w_ctrl_119,
1569  i_b => w_ctrl_103,
      o_Z => o_OVER_CTRL(1));
1571
    u_myINV_UNSIGNED : myINV
1573 port map (
      i_A => w_ctrl_147,
1575  o_Z => o_UNSIGNED);

1577 o_overflow <= w_ctrl_025;

1579 o_imm_extend <= w_ctrl_082;

1581 u_myNAND_FSM_CTRL0 : myNAND2
    port map (
1583  i_A => w_ctrl_095,
      i_b => w_ctrl_101,
1585  o_Z => o_FSM_CTRL(0));
```

```
1587    u_myNAND_FSM_CTRL1 : myNAND2
        port map (
1589      i_A => w_ctrl_095,
          i_b => w_ctrl_098,
1591      o_Z => o_FSM_CTRL(1));

1593    u_myNAND_INVALID_INSTR : myNAND2
        port map (
1595      i_A => w_instr30_n,
          i_b => w_ctrl_141,
1597      o_Z => o_INVALID_INSTR);
    end a_Instruction_Decoder;
```

Listing A.42. Instruction_Decoder.vhd Code

1.21 Encoders

1.21.1 Encoder6_3

```
--| Encoder6_3.vhd
--| Encode 6 input bits as a 3-bit output signal assuming only
--| one of the 6 inputs can be '1' at any time and the remaining
--| inputs are '0'.  Because input 0 is assigned the value '00',
--| if the remaining inputs are '0', input 0 is assumed to be '1',
--| encoder only requires inputs 1 to 5.
--| Input => Output
--| i_1 => '0001'
--| i_2 => '0010'
--| i_3 => '0011'
--| i_4 => '0100'
--| i_5 => '0101'
--| i_6 => '0110'
--| i_7 => '0111'
--| i_8 => '1000'
--|
--| INPUTS:
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_4 - Input 4
--| i_5 - Input 5
--|
--| OUTPUTS:
--| o_Z - Encoded Output
library IEEE;
use IEEE.std_logic_1164.all;

entity Encoder6_3 is
  port (i_1 : in  std_logic;
        i_2 : in  std_logic;
        i_3 : in  std_logic;
        i_4 : in  std_logic;
        i_5 : in  std_logic;
        o_Z : out std_logic_vector(2 downto 0));
end Encoder6_3;
```

```
38 architecture a_Encoder6_3 of Encoder6_3 is
   --| Declare Components
40   -- Declare NAND_2
     component myNAND2 is
42     port (i_A : in  std_logic;
           i_B : in  std_logic;
44         o_Z : out std_logic
           );
46   end component;
     -- Declare Inverter
48   component myINV is
       port (i_A : in  std_logic;
50         o_Z : out std_logic
           );
52   end component;

54 --| Declare Signals
     -- Inverted Input signals
56   signal w_1_n : std_logic; -- Inverted Input 1
     signal w_2_n : std_logic; -- Inverted Input 2
58   signal w_3_n : std_logic; -- Inverted Input 3
     signal w_4_n : std_logic; -- Inverted Input 4
60   signal w_5_n : std_logic; -- Inverted Input 5

62   -- Intermediate signals
     signal w_sig0, w_sig0_n : std_logic;
64
   begin
66
     --| Invert Inputs
68   u_myINV_1 : myINV
     port map (
70     i_A => i_1,
       o_Z => w_1_n);
72
     u_myINV_2 : myINV
74   port map (
       i_A => i_2,
76     o_Z => w_2_n);

78   u_myINV_3 : myINV
     port map (
```

```
80      i_A => i_3,
        o_Z => w_3_n);
82
    u_myINV_4 : myINV
84  port map (
        i_A => i_4,
86      o_Z => w_4_n);

88  u_myINV_5 : myINV
    port map (
90      i_A => i_5,
        o_Z => w_5_n);
92
    -- Invert itermediate signals
94  u_myINV_sig0 : myINV
    port map (
96      i_A => w_sig0,
        o_Z => w_sig0_n);
98
    --| Create Intermediate Signals
100 u_myNAND_sig0 : myNAND2
    port map (
102     i_A => w_1_n,
        i_B => w_3_n,
104     o_Z => w_sig0);

106 --| Create output signals
    u_myNAND_Z2 : myNAND2
108 port map (
        i_A => w_4_n,
110     i_B => w_5_n,
        o_Z => o_Z(2));
112
    u_myNAND_Z1 : myNAND2
114 port map (
        i_A => w_2_n,
116     i_B => w_3_n,
        o_Z => o_Z(1));
118
    u_myNAND_Z0 : myNAND2
120 port map (
        i_A => w_sig0_n,
122     i_B => w_5_n,
```

```
    o_Z => o_Z(0));
124 end a_Encoder6_3;
```

Listing A.43. Encoder6_3.vhd Code

1.21.2 Encoder9_4

```
--| Encoder9_4.vhd
--| Encode 9 input bits as a 4-bit output signal assuming only
--| one of the 9 inputs can be '1' at any time and the remaining
--| inputs are '0'.  Because input 0 is assigned the value '00',
--| if the remaining inputs are '0', input 0 is assumed to be '1',
--| encoder only requires inputs 1 to 8.
--| Input => Output
--| i_1 => '0001'
--| i_2 => '0010'
--| i_3 => '0011'
--| i_4 => '0100'
--| i_5 => '0101'
--| i_6 => '0110'
--| i_7 => '0111'
--| i_8 => '1000'
--|
--| INPUTS:
--| i_1 - Input 1
--| i_2 - Input 2
--| i_3 - Input 3
--| i_4 - Input 4
--| i_5 - Input 5
--| i_6 - Input 6
--| i_7 - Input 7
--| i_8 - Input 8
--|
--| OUTPUTS:
--| o_Z - Encoded Output
library IEEE;
use IEEE.std_logic_1164.all;

entity Encoder9_4 is
  port (i_1 : in  std_logic;
        i_2 : in  std_logic;
        i_3 : in  std_logic;
        i_4 : in  std_logic;
        i_5 : in  std_logic;
```

```
38        i_6 : in  std_logic;
          i_7 : in  std_logic;
40        i_8 : in  std_logic;
          o_Z : out std_logic_vector(3 downto 0));
42  end Encoder9_4;

44  architecture a_Encoder9_4 of Encoder9_4 is
    --| Declare Components
46    -- Declare NAND_2
      component myNAND2 is
48      port (i_A : in  std_logic;
            i_B : in  std_logic;
50          o_Z : out std_logic
            );
52    end component;
      -- Declare Inverter
54    component myINV is
        port (i_A : in  std_logic;
56          o_Z : out std_logic
            );
58    end component;

60  --| Declare Signals
      -- Inverted Input signals
62    signal w_1_n : std_logic; -- Inverted Input 1
      signal w_2_n : std_logic; -- Inverted Input 2
64    signal w_3_n : std_logic; -- Inverted Input 3
      signal w_4_n : std_logic; -- Inverted Input 4
66    signal w_5_n : std_logic; -- Inverted Input 5
      signal w_6_n : std_logic; -- Inverted Input 6
68    signal w_7_n : std_logic; -- Inverted Input 7

70    -- Intermediate signals
      signal w_sig0, w_sig0_n : std_logic;
72    signal w_sig1, w_sig1_n : std_logic;
      signal w_sig2, w_sig2_n : std_logic;
74    signal w_sig3, w_sig3_n : std_logic;
      signal w_sig4, w_sig4_n : std_logic;
76
    begin
78    o_Z(3) <= i_8;

80    --| Invert Inputs
```

```
  u_myINV_1 : myINV
82 port map (
    i_A => i_1,
84  o_Z => w_1_n);

86 u_myINV_2 : myINV
  port map (
88  i_A => i_2,
    o_Z => w_2_n);
90
  u_myINV_3 : myINV
92 port map (
    i_A => i_3,
94  o_Z => w_3_n);

96 u_myINV_4 : myINV
  port map (
98  i_A => i_4,
    o_Z => w_4_n);
100
  u_myINV_5 : myINV
102 port map (
    i_A => i_5,
104 o_Z => w_5_n);

106 u_myINV_6 : myINV
  port map (
108 i_A => i_6,
    o_Z => w_6_n);
110
  u_myINV_7 : myINV
112 port map (
    i_A => i_7,
114 o_Z => w_7_n);

116 -- Invert itermediate signals
  u_myINV_sig0 : myINV
118 port map (
    i_A => w_sig0,
120 o_Z => w_sig0_n);

122 u_myINV_sig1 : myINV
  port map (
```

```
124     i_A => w_sig1,
        o_Z => w_sig1_n);
126
    u_myINV_sig2 : myINV
128 port map (
        i_A => w_sig2,
130     o_Z => w_sig2_n);

132 u_myINV_sig3 : myINV
    port map (
134     i_A => w_sig3,
        o_Z => w_sig3_n);
136
    u_myINV_sig4 : myINV
138 port map (
        i_A => w_sig4,
140     o_Z => w_sig4_n);

142 --| Create Intermediate Signals
    u_myNAND_sig0 : myNAND2
144 port map (
        i_A => w_7_n,
146     i_B => w_6_n,
        o_Z => w_sig0);
148
    u_myNAND_sig1 : myNAND2
150 port map (
        i_A => w_5_n,
152     i_B => w_4_n,
        o_Z => w_sig1);
154
    u_myNAND_sig2 : myNAND2
156 port map (
        i_A => w_3_n,
158     i_B => w_2_n,
        o_Z => w_sig2);
160
    u_myNAND_sig3 : myNAND2
162 port map (
        i_A => w_7_n,
164     i_B => w_5_n,
        o_Z => w_sig3);
166
```

```
    u_myNAND_sig4 : myNAND2
168 port map (
      i_A => w_3_n,
170   i_B => w_1_n,
      o_Z => w_sig4);
172
  --| Create output signals
174 u_myNAND_Z2 : myNAND2
  port map (
176   i_A => w_sig0_n,
      i_B => w_sig1_n,
178   o_Z => o_Z(2));

180 u_myNAND_Z1 : myNAND2
  port map (
182   i_A => w_sig0_n,
      i_B => w_sig2_n,
184   o_Z => o_Z(1));

186 u_myNAND_Z0 : myNAND2
  port map (
188   i_A => w_sig3_n,
      i_B => w_sig4_n,
190   o_Z => o_Z(0));
end a_Encoder9_4;
```

Listing A.44. Encoder9_4.vhd Code

1.22 Instruction Decoder/Encoder

```
--| Instruction_Dec_Enc.vhd
--| Decodes the 32-bit MIPS instruction to determine which portions of the ALU to activate.
--| Encodes some of these signals to control muxes in the datapath.
--|
--| INPUTS:
--| i_instr31       - Instruction inputs
--| ...
--| i_instr00       - Instruction inputs
--| i_BRANCH_OVERRIDE- Modify ALU_SRC_A(1) so that the ALU will use the PC input for ALU_SRC_A.
--|                 - Override ALU_SRC_B and ALU_INV_B controls generated by decoding the
--|                   instruction.  Used by the controller to modify the behavior of the
--|                   datapath during the second cycle of a branch instruction.
--|                 - Modify ALU_OUTPUT to select the adder output.  Used in the second cycle
--|                   of branch instructions to add the PC and the branch offset or 4.
--| i_DO_NOT_STORE  - Override the REG_SEL control generated by decoding the instruction.
--|                   Used by the controller to ensure data is only written to registers
--|                   during the write phase of the particular instruction when the ALU
--|                   result or data from memory is ready to be sampled and stored.
--| i_PC_STORE      - Override the REG_SEL control generated by decoding the instruction.
--|                   Cause the output of the datapath to be stored to the PC register.
--|                   Used for branch instructions to store the result of a branch to
--|                   the PC register.
--|
--|
--| OUTPUTS:
--| o_ALU_SRC_A - ALU Source A selector
```

```
--| o_ALU_SRC_B - ALU Source B selector
--| o_ALU_INV_B - ALU Invert B selector
--| o_COMP_SEL  - Select the value to output from the Comparator
--| o_OVER_CTRL - Select which overflow detection means should be used
--|             (i.e. ADD, ADDI, SUB, or no overflow detection).
--| o_ALU_OUTPUT  - Select which ALU function should be output
--|             0 - Shift Left
--|             1 - Shift Right
--|             2 - Adder
--|             3 - AND
--|             4 - OR
--|             5 - XOR
--|             6 - NOR
--|             7 - Comparator
--|             8 - LUI
--|             9, 10, 11, 12, 13, 14, 15 - Shift Left
--| o_REG_SEL   - Select the destination register.
--| o_UNSIGNED    - Determine if the comparator is comparing sigend numbers (0) or unsigned
--|             numbers (1).
--| o_overflow    - Determine if the comparator is performing a Less Than comparison that will
--|                 or will not produce an overflow.  No overflow can occur in BLTZ instructions,
--|             but overflow can occur in SLT, SLTU, SLTI, and SLTIU instructions.  0 - no
--|             overflow can occur, 1 - overflow can occur
--| o_imm_extend    - Determine if the immediate value should be 0 extended or sign extended
--| o_FSM_CTRL    - Finite State Machine Control Signal
--| o_INVALID_INSTR - Signals when an invalid instruction is received
library IEEE;
use IEEE.std_logic_1164.all;

entity Instruction_Dec_Enc is
  port (i_instr31     : in  std_logic;
        i_instr30     : in  std_logic;
        i_instr29     : in  std_logic;
```

```
        i_instr28      : in  std_logic;
        i_instr27      : in  std_logic;
        i_instr26      : in  std_logic;
        i_instr20      : in  std_logic;
        i_instr19      : in  std_logic;
        i_instr18      : in  std_logic;
        i_instr17      : in  std_logic;
        i_instr16      : in  std_logic;
        i_instr05      : in  std_logic;
        i_instr04      : in  std_logic;
        i_instr03      : in  std_logic;
        i_instr02      : in  std_logic;
        i_instr01      : in  std_logic;
        i_instr00      : in  std_logic;
        i_BRANCH_OVERRIDE : in  std_logic;
        i_DO_NOT_STORE    : in  std_logic;
        i_PC_STORE        : in  std_logic;
        o_ALU_SRC_A       : out std_logic_vector(1 downto 0);
        o_ALU_SRC_B       : out std_logic_vector(1 downto 0);
        o_ALU_INV_B       : out std_logic;
        o_COMP_SEL        : out std_logic_vector(2 downto 0);
        o_OVER_CTRL       : out std_logic_vector(1 downto 0);
        o_ALU_OUTPUT      : out std_logic_vector(3 downto 0);
        o_REG_SEL      : out std_logic_vector(1 downto 0);
        o_UNSIGNED        : out std_logic;
        o_overflow        : out std_logic;
        o_imm_extend      : out std_logic;
        o_FSM_CTRL        : out std_logic_vector(1 downto 0);
        o_INVALID_INSTR : out std_logic);
end Instruction_Dec_Enc;

architecture a_Instruction_Dec_Enc of Instruction_Dec_Enc is
--| Define Components
  component myMUX2_N is
    generic (m_width : integer := 2);
    port (i_0 : in  std_logic_vector(m_width-1 downto 0);
          i_1 : in  std_logic_vector(m_width-1 downto 0);
          i_S : in  std_logic;
          o_Z : out std_logic_vector(m_width-1 downto 0));
  end component;

  component myMUX2_1 is
    port (i_0 : in  std_logic;
```

```
103         i_1 : in  std_logic;
            i_S : in  std_logic;
105         o_Z : out std_logic);
    end component;
107
    component Instruction_Decoder is
109   port (i_instr31   : in  std_logic;
            i_instr30   : in  std_logic;
111         i_instr29   : in  std_logic;
            i_instr28   : in  std_logic;
113         i_instr27   : in  std_logic;
            i_instr26   : in  std_logic;
115         i_instr20   : in  std_logic;
            i_instr19   : in  std_logic;
117         i_instr18   : in  std_logic;
            i_instr17   : in  std_logic;
119         i_instr16   : in  std_logic;
            i_instr05   : in  std_logic;
121         i_instr04   : in  std_logic;
            i_instr03   : in  std_logic;
123         i_instr02   : in  std_logic;
            i_instr01   : in  std_logic;
125         i_instr00   : in  std_logic;
            o_ALU_SRC_A0  : out std_logic;
127         o_ALU_SRC_B   : out std_logic_vector(1 downto 0);
            o_INV_B       : out std_logic;
129         o_LTZ         : out std_logic;
            o_EQ          : out std_logic;
131         o_NE          : out std_logic;
            o_LEZ         : out std_logic;
133         o_GTZ         : out std_logic;
            o_RIGHT_SHIFT : out std_logic;
135         o_ADDER       : out std_logic;
            o_AND         : out std_logic;
137         o_OR          : out std_logic;
            o_XOR         : out std_logic;
139         o_NOR         : out std_logic;
            o_COMPARE     : out std_logic;
141         o_LUI         : out std_logic;
            o_REG_SEL     : out std_logic_vector(1 downto 0);
143         o_OVER_CTRL   : out std_logic_vector(1 downto 0);
            o_UNSIGNED    : out std_logic;
145         o_overflow    : out std_logic;
```

```
        o_imm_extend   : out std_logic;
147     o_FSM_CTRL     : out std_logic_vector(1 downto 0);
        o_INVALID_INSTR: out std_logic);
149 end component;

151 component Encoder6_3 is
      port (i_1 : in  std_logic;
153       i_2 : in  std_logic;
          i_3 : in  std_logic;
155       i_4 : in  std_logic;
          i_5 : in  std_logic;
157       o_Z : out std_logic_vector(2 downto 0));
    end component;
159
    component Encoder9_4 is
161   port (i_1 : in  std_logic;
          i_2 : in  std_logic;
163       i_3 : in  std_logic;
          i_4 : in  std_logic;
165       i_5 : in  std_logic;
          i_6 : in  std_logic;
167       i_7 : in  std_logic;
          i_8 : in  std_logic;
169       o_Z : out std_logic_vector(3 downto 0));
    end component;
171
--| Define Signals
173 signal w_ALU_SRC_B   : std_logic_vector(1 downto 0);
    signal w_ALU_INV_B      : std_logic;
175 signal w_LTZ       : std_logic;
    signal w_EQ        : std_logic;
177 signal w_NE        : std_logic;
    signal w_LEZ       : std_logic;
179 signal w_GTZ       : std_logic;
    signal w_RIGHT_SHIFT   : std_logic;
181 signal w_ADDER       : std_logic;
    signal w_AND       : std_logic;
183 signal w_OR        : std_logic;
    signal w_XOR       : std_logic;
185 signal w_NOR       : std_logic;
    signal w_COMPARE    : std_logic;
187 signal w_LUI       : std_logic;
```

```
  signal w_REG_SEL0, w_REG_SEL1    : std_logic_vector(1 downto
    0);
  signal w_ALU_OUTPUT : std_logic_vector(3 downto 0);

--| Define Constants
  constant k_one2 : std_logic_vector(1 downto 0) := B"11";
  constant k_zero1 : std_logic := '0';
  constant k_zero2 : std_logic_vector(1 downto 0) := B"00";
  constant k_two4 : std_logic_vector(3 downto 0) := B"0010";

begin
  o_ALU_SRC_A(1) <= i_BRANCH_OVERRIDE;

  -- Connect the Instruction Decoder
  u_Instruction_Decoder: Instruction_Decoder
  port map (i_instr31 => i_instr31,
            i_instr30 => i_instr30,
            i_instr29 => i_instr29,
            i_instr28 => i_instr28,
            i_instr27 => i_instr27,
            i_instr26 => i_instr26,
            i_instr20 => i_instr20,
            i_instr19 => i_instr19,
            i_instr18 => i_instr18,
            i_instr17 => i_instr17,
            i_instr16 => i_instr16,
            i_instr05 => i_instr05,
            i_instr04 => i_instr04,
            i_instr03 => i_instr03,
            i_instr02 => i_instr02,
            i_instr01 => i_instr01,
            i_instr00 => i_instr00,
            o_ALU_SRC_A0 => o_ALU_SRC_A(0),
            o_ALU_SRC_B => w_ALU_SRC_B,
            o_INV_B => w_ALU_INV_B,
            o_LTZ => w_LTZ,
            o_EQ => w_EQ,
            o_NE => w_NE,
            o_LEZ => w_LEZ,
            o_GTZ => w_GTZ,
            o_RIGHT_SHIFT => w_RIGHT_SHIFT,
            o_ADDER => w_ADDER,
            o_AND => w_AND,
```

```
            o_OR => w_OR,
231         o_XOR => w_XOR,
            o_NOR => w_NOR,
233         o_COMPARE => w_COMPARE,
            o_LUI => w_LUI,
235         o_REG_SEL => w_REG_SEL0,
            o_OVER_CTRL => o_OVER_CTRL,
237         o_UNSIGNED => o_UNSIGNED,
            o_overflow => o_overflow,
239         o_imm_extend => o_imm_extend,
            o_FSM_CTRL => o_FSM_CTRL,
241         o_INVALID_INSTR => o_INVALID_INSTR);

243 -- Encode the COMP_SEL signal
    u_Encoder6_3 : Encoder6_3
245 port map (
      i_1 => w_LTZ,
247   i_2 => w_EQ,
      i_3 => w_NE,
249   i_4 => w_LEZ,
      i_5 => w_GTZ,
251   o_Z => o_COMP_SEL);

253 -- Encode the ALU_OUTPUT signal
    u_Encoder9_4 : Encoder9_4
255 port map (
      i_1 => w_RIGHT_SHIFT,
257   i_2 => w_ADDER,
      i_3 => w_AND,
259   i_4 => w_OR,
      i_5 => w_XOR,
261   i_6 => w_NOR,
      i_7 => w_COMPARE,
263   i_8 => w_LUI,
      o_Z => w_ALU_OUTPUT);
265
    -- Determine whether ALU_SRC_B is dependent on the
     instruction or the current state
267 u_myMUX2_SRC_B: myMUX2_N
    generic map (m_width => 2)
269 port map (i_0 => w_ALU_SRC_B,
            i_1 => k_one2,
271         i_S => i_BRANCH_OVERRIDE,
```

```
            o_Z => o_ALU_SRC_B);

    -- Determine whether ALU_INV_B is dependent on the
     instruction or the current state
    u_myMUX2_INV_B: myMUX2_1
    port map (i_0 => w_ALU_INV_B,
            i_1 => k_zero1,
            i_S => i_BRANCH_OVERRIDE,
            o_Z => o_ALU_INV_B);

    -- Determine whether data should be written to the PC
     register
    u_myMUX2_REG_SEL1: myMUX2_N
    generic map (m_width => 2)
    port map (i_0 => w_REG_SEL0,
            i_1 => k_one2,
            i_S => i_PC_STORE,
            o_Z => w_REG_SEL1);

    -- Determine whether ALU_OUTPUT is dependent on the
     instruction or the current state
    u_myMUX2_ALU_OUTPUT: myMUX2_N
    generic map (m_width => 4)
    port map (i_0 => w_ALU_OUTPUT,
            i_1 => k_two4,
            i_S => i_BRANCH_OVERRIDE,
            o_Z => o_ALU_OUTPUT);

    -- Determine whether data should be written to registers
    u_myMUX2_REG_SEL: myMUX2_N
    generic map (m_width => 2)
    port map (i_0 => w_REG_SEL1,
            i_1 => k_zero2,
            i_S => i_DO_NOT_STORE,
            o_Z => o_REG_SEL);

end a_Instruction_Dec_Enc;
```

Listing A.45. Instruction_Dec_Enc.vhd Code

1.23 Finite State Machine Control Signal Generator

```
1  --| FSM_Control_Signal_Generator.vhd
   --| Determines the control signals issued by the FSM Controller based on the
3  --| current state.
   --|
5  --| INPUTS:
   --| i_state - current state of the FSM
7  --|
   --| OUTPUTS:
9  --| o_MEM_READ         - Read from memory enable signal
   --| o_MEM_WRITE        - Write to memory enable signal
11 --| o_MEM_ADDRESS_SEL - Select whether the memory address comes from the PC register or the ALU Result
   --| o_STORE_FROM_MEM    - Store word from memory - select signal to determine if input to GPR_BANK is
13 --|                     from memory if '1' or from the ALU Result if '0'
   --| o_PC_STORE         - Override the normal value of REG_SEL to store the ALU Result to the PC register
15 --| o_PC_EN            - Enable the PC Register regardless of any other control signals.  Used to store
   --|                     PC+4 to the PC register.
17 --| o_DO_NOT_STORE     - Override the normal value of REG_SEL not store any incoming data to the GPR_BANK
   --| o_BRANCH_OVERRIDE - Override the normal value of the ALU_SRC_A, ALU_SRC_B, ALU_INV_B, and ALU_OUTPUT
19 --|                     control signals for a branch instruction
   --| o_STORE_INSTRUCTION - Enable the instruction register to store the data read from memory
21 library IEEE;
   use IEEE.std_logic_1164.all;
23
   entity FSM_Control_Signal_Generator is
25   port (i_state           : in  std_logic_vector(3 downto 0);
         o_MEM_READ          : out std_logic;
27       o_MEM_WRITE         : out std_logic;
         o_MEM_ADDRESS_SEL   : out std_logic;
29       o_STORE_FROM_MEM    : out std_logic;
         o_PC_STORE          : out std_logic;
31       o_PC_EN           : out std_logic;
         o_DO_NOT_STORE      : out std_logic;
```

```
33        o_BRANCH_OVERRIDE    : out std_logic;
          o_STORE_INSTRUCTION : out std_logic);
35  end FSM_Control_Signal_Generator;

37  architecture a_FSM_Control_Signal_Generator of
      FSM_Control_Signal_Generator is
--| Declare Components
39    -- Declare NAND_2
    component myNAND2 is
41      port (i_A : in  std_logic;
            i_B : in  std_logic;
43          o_Z : out std_logic
            );
45    end component;
    -- Declare Inverter
47    component myINV is
      port (i_A : in  std_logic;
49          o_Z : out std_logic
            );
51    end component;

53    --| Declare Signals
    -- Inverted State Signals
55    signal w_S3_n : std_logic;
    signal w_S2_n : std_logic;
57    signal w_S1_n : std_logic;
    signal w_S0_n : std_logic;
59
    -- Intermediate control signals (and inverted control
     signals) used to compute outputs
61    signal w_fsm_00 : std_logic;
    signal w_fsm_01, w_fsm_01_n : std_logic;
63    signal w_fsm_02, w_fsm_02_n : std_logic;
    signal w_fsm_03, w_fsm_03_n : std_logic;
65    signal w_fsm_04, w_fsm_04_n : std_logic;
    signal w_fsm_05, w_fsm_05_n : std_logic;
67    signal w_fsm_06 : std_logic;
    signal w_fsm_07, w_fsm_07_n : std_logic;
69    signal w_fsm_08, w_fsm_08_n : std_logic;
    signal w_fsm_09 : std_logic;
71    signal w_fsm_10, w_fsm_10_n : std_logic;
    signal w_fsm_11 : std_logic;
73    signal w_fsm_12 : std_logic;
```

```
--  signal w_fsm_13 : std_logic;
  signal w_fsm_14 : std_logic;
  signal w_fsm_15 : std_logic;
--  signal w_fsm_16 : std_logic;
--  signal w_fsm_17 : std_logic;
--  signal w_fsm_18 : std_logic;
  signal w_fsm_19 : std_logic;
  signal w_fsm_20, w_fsm_20_n : std_logic;
  signal w_fsm_21 : std_logic;
  signal w_fsm_22 : std_logic;
--  signal w_fsm_23 : std_logic;
  signal w_fsm_24 : std_logic;
  signal w_fsm_25, w_fsm_25_n : std_logic;
  signal w_fsm_26 : std_logic;
  signal w_fsm_27, w_fsm_27_n : std_logic;

begin
  --| Instantiate Inverters to create inverted signals where
   needed
  u_myINV_S0 : myINV
  port map (
    i_A => i_state(0),
    o_Z => w_S0_n);

  u_myINV_S1 : myINV
  port map (
    i_A => i_state(1),
    o_Z => w_S1_n);

  u_myINV_S2 : myINV
  port map (
    i_A => i_state(2),
    o_Z => w_S2_n);

  u_myINV_S3 : myINV
  port map (
    i_A => i_state(3),
    o_Z => w_S3_n);

  u_myINV_01 : myINV
  port map (
    i_A => w_fsm_01,
    o_Z => w_fsm_01_n);
```

```
117   u_myINV_02 : myINV
      port map (
119     i_A => w_fsm_02,
        o_Z => w_fsm_02_n);
121
      u_myINV_03 : myINV
123   port map (
        i_A => w_fsm_03,
125     o_Z => w_fsm_03_n);

127   u_myINV_04 : myINV
      port map (
129     i_A => w_fsm_04,
        o_Z => w_fsm_04_n);
131
      u_myINV_05 : myINV
133   port map (
        i_A => w_fsm_05,
135     o_Z => w_fsm_05_n);

137   u_myINV_07 : myINV
      port map (
139     i_A => w_fsm_07,
        o_Z => w_fsm_07_n);
141
      u_myINV_08 : myINV
143   port map (
        i_A => w_fsm_08,
145     o_Z => w_fsm_08_n);

147   u_myINV_10 : myINV
      port map (
149     i_A => w_fsm_10,
        o_Z => w_fsm_10_n);
151
      u_myINV_20 : myINV
153   port map (
        i_A => w_fsm_20,
155     o_Z => w_fsm_20_n);

157   u_myINV_25 : myINV
      port map (
```

```
159     i_A => w_fsm_25,
        o_Z => w_fsm_25_n);
161
    u_myINV_27 : myINV
163 port map (
        i_A => w_fsm_27,
165     o_Z => w_fsm_27_n);

167 --| Instantiate NAND gates to create intermediate control
     signals
    u_myNAND_00 : myNAND2
169 port map (
        i_A => i_state(2),
171     i_B => w_S1_n,
        o_Z => w_fsm_00);
173
    u_myNAND_01 : myNAND2
175 port map (
        i_A => w_S2_n,
177     i_B => i_state(1),
        o_Z => w_fsm_01);
179
    u_myNAND_02 : myNAND2
181 port map (
        i_A => w_S3_n,
183     i_B => w_S0_n,
        o_Z => w_fsm_02);
185
    u_myNAND_03 : myNAND2
187 port map (
        i_A => w_fsm_00,
189     i_B => w_fsm_01,
        o_Z => w_fsm_03);
191
    u_myNAND_04 : myNAND2
193 port map (
        i_A => i_state(3),
195     i_B => w_S2_n,
        o_Z => w_fsm_04);
197
    u_myNAND_05 : myNAND2
199 port map (
        i_A => w_S1_n,
```

```
201       i_B => w_S0_n,
          o_Z => w_fsm_05);
203
      u_myNAND_06 : myNAND2
205   port map (
          i_A => w_S3_n,
207       i_B => w_fsm_15,
          o_Z => w_fsm_06);
209
      u_myNAND_07 : myNAND2
211   port map (
          i_A => w_S3_n,
213       i_B => i_state(2),
          o_Z => w_fsm_07);
215
      u_myNAND_08 : myNAND2
217   port map (
          i_A => i_state(1),
219       i_B => i_state(0),
          o_Z => w_fsm_08);
221
      u_myNAND_09 : myNAND2
223   port map (
          i_A => w_fsm_07_n,
225       i_B => w_fsm_08_n,
          o_Z => w_fsm_09);
227
      u_myNAND_10 : myNAND2
229   port map (
          i_A => w_S1_n,
231       i_B => i_state(0),
          o_Z => w_fsm_10);
233
      u_myNAND_11 : myNAND2
235   port map (
          i_A => w_fsm_07_n,
237       i_B => w_fsm_10_n,
          o_Z => w_fsm_11);
239
      u_myNAND_12 : myNAND2
241   port map (
          i_A => w_fsm_04_n,
243       i_B => w_fsm_05_n,
```

```
    o_Z => w_fsm_12);

--  u_myNAND_13 : myNAND2
--  port map (
--    i_A => w_S3_n,
--    i_B => w_fsm_08_n,
--    o_Z => w_fsm_13);

  u_myNAND_14 : myNAND2
  port map (
    i_A => w_S2_n,
    i_B => w_S1_n,
    o_Z => w_fsm_14);

  u_myNAND_15 : myNAND2
  port map (
    i_A => w_S2_n,
    i_B => w_fsm_05_n,
    o_Z => w_fsm_15);

--  u_myNAND_16 : myNAND2
--  port map (
--    i_A => i_state(0),
--    i_B => w_fsm_14,
--    o_Z => w_fsm_16);
--
--  u_myNAND_17 : myNAND2
--  port map (
--    i_A => i_state(3),
--    i_B => w_fsm_15,
--    o_Z => w_fsm_17);
--
--  u_myNAND_18 : myNAND2
--  port map (
--    i_A => w_fsm_17,
--    i_B => w_fsm_16,
--    o_Z => w_fsm_18);

  u_myNAND_19 : myNAND2
  port map (
    i_A => w_fsm_07_n,
    i_B => w_S1_n,
    o_Z => w_fsm_19);
```

```
287
   u_myNAND_20 : myNAND2
289   port map (
        i_A => w_S3_n,
        i_B => w_S2_n,
291     o_Z => w_fsm_20);
293
   u_myNAND_21 : myNAND2
295   port map (
        i_A => w_fsm_20_n,
        i_B => w_fsm_10_n,
297     o_Z => w_fsm_21);
299
   u_myNAND_22 : myNAND2
301   port map (
        i_A => w_fsm_03_n,
        i_B => w_fsm_02_n,
303     o_Z => w_fsm_22);
305
--   u_myNAND_23 : myNAND2
307 --   port map (
--      i_A => w_fsm_04_n,
309 --      i_B => w_fsm_10_n,
--      o_Z => w_fsm_23);
311
   u_myNAND_24 : myNAND2
313   port map (
        i_A => i_state(3),
        i_B => i_state(0),
315     o_Z => w_fsm_24);
317
   u_myNAND_25 : myNAND2
319   port map (
        i_A => w_fsm_24,
        i_B => w_fsm_02,
321     o_Z => w_fsm_25);
323
   u_myNAND_26 : myNAND2
325   port map (
        i_A => w_fsm_01_n,
        i_B => w_fsm_25_n,
327     o_Z => w_fsm_26);
329
```

```
    u_myNAND_27 : myNAND2
331 port map (
      i_A => w_s3_n,
333   i_B => i_state(0),
      o_Z => w_fsm_27);
335
    --| Instantiate Inverters and NAND gates to generate output
      signals
337 u_myINV_MEM_READ : myINV
    port map (
339   i_A => w_fsm_22,
      o_Z => o_MEM_READ);
341
    u_myINV_MEM_WRITE : myINV
343 port map (
      i_A => w_fsm_12,
345   o_Z => o_MEM_WRITE);

347 u_myNAND_MEM_ADDRESS : myNAND2
    port map (
349   i_A => w_fsm_06,
      i_B => w_fsm_12,
351   o_Z => o_MEM_ADDRESS_SEL);

353 u_myINV_STORE_FROM_MEM : myINV
    port map (
355   i_A => w_fsm_09,
      o_Z => o_STORE_FROM_MEM);
357
    u_myINV_PC_STORE : myINV
359 port map (
      i_A => w_fsm_11,
361   o_Z => o_PC_STORE);

363 u_myINV_PC_EN : myINV
    port map (
365   i_A => w_fsm_26,
      o_Z => o_PC_EN);
367
--  u_myNAND_PC_EN : myNAND2
369 --  port map (
--    i_A => w_fsm_13,
371 --    i_B => w_fsm_23,
```

```
--      o_Z => o_PC_EN);

  u_myNAND_DO_NOT_STORE : myNAND2
  port map (
    i_A => w_fsm_27_n,
    i_B => w_fsm_14,
    o_Z => o_DO_NOT_STORE);

--  u_myINV_DO_NOT_STORE : myINV
--  port map (
--    i_A => w_fsm_18,
--    o_Z => o_DO_NOT_STORE);

  u_myINV_BRANCH_OVERRIDE : myINV
  port map (
    i_A => w_fsm_19,
    o_Z => o_BRANCH_OVERRIDE);

  u_myINV_STORE_INSTRUCTION : myINV
  port map (
    i_A => w_fsm_21,
    o_Z => o_STORE_INSTRUCTION);
end a_FSM_Control_Signal_Generator;
```

Listing A.46. FSM_Control_Signal_Generator.vhd Code

1.24 Controller Finite State Machine (FSM)

```
--| Controller_FSM.vhd
--| Based on instructions input from memory, determine the type of instruction,
--| the correct sequence of state transitions for that type of instruction, and
--| issue the appropriate instructions to the datapath during the appropriate
--| states for that instruction.
--|
--| INPUTS:
--| i_clk    - Clock input
--| i_reset    - Reset input
--| i_MEM_OUT - Output from memory to be stored into the instruction register
--| i_MEM_READY- Signal is 1 when memory has completed the read or write task
--|
--| OUTPUTS:
--| o_MEM_READ          - Read from memory enable signal
--| o_MEM_WRITE      - Write to memory enable signal
--| o_MEM_ADDRESS_SEL - Select whether the memory address comes from the PC register or the ALU Result
--| o_STORE_FROM_MEM    - Store word from memory - select signal to determine if input to GPR_BANK is
--|                 from memory if '1' or from the ALU Result if '0'
--| o_PC_EN          - Enable the PC Register regardless of any other control signals.  Used to store
--|                 PC+4 to the PC register.
--| o_ALU_SRC_A      - ALU Source A selector
--| o_ALU_SRC_B      - ALU Source B selector
--| o_ALU_INV_B      - ALU Invert B selector
--| o_COMP_SEL       - Select the value to output from the Comparator
--| o_OVER_CTRL      - Select which overflow detection means should be used
--|            (i.e. ADD, ADDI, SUB, or no overflow detection).
--| o_ALU_OUTPUT       - Select which ALU function should be output
--|                 0 - Shift Left
```

```
--|                 1 - Shift Right
--|                 2 - Adder
--|                 3 - AND
--|                 4 - OR
--|                 5 - XOR
--|                 6 - NOR
--|                 7 - Comparator
--|                 8 - LUI
--|                 9, 10, 11, 12, 13, 14, 15 - Shift Left
--| o_REG_SEL       - Select the destination register
--| o_UNSIGNED      - Determine if the comparator is comparing sigend numbers (0) or unsigned
--|                   numbers (1).
--| o_overflow      - Determine if the comparator is performing a Less Than comparison that will
--|                   or will not produce an overflow. No overflow can occur in BLTZ instructions,
--|                 but overflow can occur in SLT, SLTU, SLTI, and SLTIU instructions.  0 - no
--|                 overflow can occur, 1 - overflow can occur
--| o_imm_extend    - Determine if the immediate value should be 0 extended or sign extended
--| o_RS_SEL        - Reference to the RS register
--| o_RT_SEL        - Reference to the RT register
--| o_immediate     - The immediate value also contains the reference to
--|                 the RD register, the shift amount (SA), and bit zero
--|                 of the instruction (used to determine the difference
--|                 between a logical and arithmetic right shift by the
--|                 ALU Core's Right Shift module).
library IEEE;
use IEEE.std_logic_1164.all;

entity Controller_FSM is
  port (i_clk         : in  std_logic;
        i_reset       : in  std_logic;
        i_MEM_OUT     : in  std_logic_vector(31 downto 0);
        i_MEM_READY   : in std_logic;
        o_MEM_READ    : out std_logic;
        o_MEM_WRITE   : out std_logic;
```

```
      o_MEM_ADDRESS_SEL : out std_logic;
      o_STORE_FROM_MEM  : out std_logic;
      o_PC_EN           : out std_logic;
      o_ALU_SRC_A       : out std_logic_vector(1 downto 0);
      o_ALU_SRC_B       : out std_logic_vector(1 downto 0);
      o_ALU_INV_B       : out std_logic;
      o_COMP_SEL        : out std_logic_vector(2 downto 0);
      o_OVER_CTRL       : out std_logic_vector(1 downto 0);
      o_ALU_OUTPUT      : out std_logic_vector(3 downto 0);
      o_REG_SEL         : out std_logic_vector(1 downto 0);
      o_UNSIGNED        : out std_logic;
      o_overflow        : out std_logic;
      o_imm_extend      : out std_logic;
      o_RS_SEL          : out std_logic_vector(4 downto 0);
      o_RT_SEL          : out std_logic_vector(4 downto 0);
      o_immediate       : out std_logic_vector(15 downto 0));
end Controller_FSM;

architecture a_Controller_FSM of Controller_FSM is
--| Define Components
  -- Register to use for states
  component myRegN is
    generic (m_width : integer := 4);
    port (i_clk    : in  std_logic;
          i_reset  : in  std_logic;
          i_D      : in  std_logic_vector (m_width-1 downto 0);
          o_Q      : out std_logic_vector (m_width-1 downto 0));
  end component;

  -- Enabled Register
  component Enabled_Register is
    port (i_clk    : in  std_logic;
          i_reset  : in  std_logic;
          i_data   : in  std_logic_vector(31 downto 0);
          i_en     : in  std_logic;
          o_Q      : out std_logic_vector(31 downto 0));
  end component;

  -- 2-input N-bit mux
  component myMUX2_N is
    generic (m_width : integer := 4);
    port (i_0 : in  std_logic_vector(m_width-1 downto 0);
          i_1 : in  std_logic_vector(m_width-1 downto 0);
```

```
106         i_S : in  std_logic;
            o_Z : out std_logic_vector(m_width-1 downto 0));
108   end component;

110   -- 4-input N-bit mux
      component myMUX4_N is
112     generic (m_width : integer := 4);
        port (i_0 : in  std_logic_vector(m_width-1 downto 0);
114         i_1 : in  std_logic_vector(m_width-1 downto 0);
            i_2 : in  std_logic_vector(m_width-1 downto 0);
116         i_3 : in  std_logic_vector(m_width-1 downto 0);
            i_S : in  std_logic_vector(1 downto 0);
118         o_Z : out std_logic_vector(m_width-1 downto 0));
      end component;
120
      -- 16-input N-bit mux
122   component myMUX16_N is
        generic (m_width : integer := 4);
124     port (i_0 : in  std_logic_vector(m_width-1 downto 0);
            i_1 : in  std_logic_vector(m_width-1 downto 0);
126         i_2 : in  std_logic_vector(m_width-1 downto 0);
            i_3 : in  std_logic_vector(m_width-1 downto 0);
128         i_4 : in  std_logic_vector(m_width-1 downto 0);
            i_5 : in  std_logic_vector(m_width-1 downto 0);
130         i_6 : in  std_logic_vector(m_width-1 downto 0);
            i_7 : in  std_logic_vector(m_width-1 downto 0);
132         i_8 : in  std_logic_vector(m_width-1 downto 0);
            i_9 : in  std_logic_vector(m_width-1 downto 0);
134         i_10 : in  std_logic_vector(m_width-1 downto 0);
            i_11 : in  std_logic_vector(m_width-1 downto 0);
136         i_12 : in  std_logic_vector(m_width-1 downto 0);
            i_13 : in  std_logic_vector(m_width-1 downto 0);
138         i_14 : in  std_logic_vector(m_width-1 downto 0);
            i_15 : in  std_logic_vector(m_width-1 downto 0);
140         i_S : in  std_logic_vector(3 downto 0);
            o_Z : out std_logic_vector(m_width-1 downto 0));
142   end component;

144   -- FSM Control Signal Generator to generate state dependent
        control signals
      component FSM_Control_Signal_Generator is
146     port (i_state          : in  std_logic_vector(3 downto 0);
          o_MEM_READ         : out std_logic;
```

```
148         o_MEM_WRITE         : out std_logic;
            o_MEM_ADDRESS_SEL   : out std_logic;
150         o_STORE_FROM_MEM    : out std_logic;
            o_PC_STORE          : out std_logic;
152         o_PC_EN             : out std_logic;
            o_DO_NOT_STORE      : out std_logic;
154         o_BRANCH_OVERRIDE   : out std_logic;
            o_STORE_INSTRUCTION : out std_logic);
156   end component;

158   -- Instruction Decoder and Encoder to generate instruction
       and state dependent control signals
      component Instruction_Dec_Enc is
160     port (i_instr31     : in  std_logic;
            i_instr30       : in  std_logic;
162         i_instr29       : in  std_logic;
            i_instr28       : in  std_logic;
164         i_instr27       : in  std_logic;
            i_instr26       : in  std_logic;
166         i_instr20       : in  std_logic;
            i_instr19       : in  std_logic;
168         i_instr18       : in  std_logic;
            i_instr17       : in  std_logic;
170         i_instr16       : in  std_logic;
            i_instr05       : in  std_logic;
172         i_instr04       : in  std_logic;
            i_instr03       : in  std_logic;
174         i_instr02       : in  std_logic;
            i_instr01       : in  std_logic;
176         i_instr00       : in  std_logic;
           i_BRANCH_OVERRIDE : in  std_logic;
178        i_DO_NOT_STORE     : in  std_logic;
           i_PC_STORE         : in  std_logic;
180        o_ALU_SRC_A        : out std_logic_vector(1 downto 0);
           o_ALU_SRC_B        : out std_logic_vector(1 downto 0);
182        o_ALU_INV_B        : out std_logic;
           o_COMP_SEL         : out std_logic_vector(2 downto 0);
184        o_OVER_CTRL        : out std_logic_vector(1 downto 0);
           o_ALU_OUTPUT       : out std_logic_vector(3 downto 0);
186        o_REG_SEL       : out std_logic_vector(1 downto 0);
           o_UNSIGNED         : out std_logic;
188        o_overflow         : out std_logic;
           o_imm_extend       : out std_logic;
```

```
190         o_FSM_CTRL       : out std_logic_vector(1 downto 0);
            o_INVALID_INSTR  : out std_logic);
192   end component;

194 --| Define Signals
    -- Finite State Machine Signals
196   signal w_state      : std_logic_vector(3 downto 0); --
       Current state of the FSM
      signal w_next_state : std_logic_vector(3 downto 0); -- Next
        state of the FSM
198   signal w_next_state0  : std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM if there is no
       instruction error
      signal w_next_state12: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 1
200   signal w_next_state2  : std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 2
      signal w_next_state01: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 0
202   signal w_next_state67: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 6
      signal w_next_state70: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 7
204   signal w_next_state89: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 8
      signal w_next_state90: std_logic_vector(3 downto 0); --
       Potentially the next state if the FSM is currently in
       state 9
206

208   -- Instruction Signal
      signal w_instruction : std_logic_vector(31 downto 0);
210
      -- Internal Control SIgnals
212   signal w_BRANCH_OVERRIDE  : std_logic; -- See description
       in Instruction_Dec_Enc.vhd
```

```
    signal w_DO_NOT_STORE    : std_logic; -- See description in
     Instruction_Dec_Enc.vhd
214 signal w_PC_STORE        : std_logic; -- See description in
     Instruction_Dec_Enc.vhd
    signal w_STORE_INSTRUCTION  : std_logic; -- Enable the
     Instruction Register to store the data from memory
216 signal w_INVALID_INSTR: std_logic;     -- Signal to set the
     FSM to state 0 if an invalid instruction is detected
    signal w_FSM_CTRL    : std_logic_vector(1 downto 0); --
     Signal to determine the value of w_next_state2 depending
     on the instruction
218
--| Define Constants
220 constant k_zero_4    : std_logic_vector(3 downto 0) := (
     others => '0');
    constant k_one_4     : std_logic_vector(3 downto 0) := B"
     0001";
222 constant k_two_4     : std_logic_vector(3 downto 0) := B"
     0010";
    constant k_three_4   : std_logic_vector(3 downto 0) := B"
     0011";
224 constant k_four_4    : std_logic_vector(3 downto 0) := B"
     0100";
    constant k_five_4    : std_logic_vector(3 downto 0) := B"
     0101";
226 constant k_six_4     : std_logic_vector(3 downto 0) := B"
     0110";
    constant k_seven_4   : std_logic_vector(3 downto 0) := B"
     0111";
228 constant k_eight_4   : std_logic_vector(3 downto 0) := B"
     1000";
    constant k_nine_4    : std_logic_vector(3 downto 0) := B"
     1001";
230 constant k_ten_4     : std_logic_vector(3 downto 0) := B"
     1010";

232 begin
  -- Connect Instruction Register
234 u_Instruction_Register: Enabled_Register
    port map (i_clk => i_clk,
236           i_reset => i_reset,
              i_data => i_MEM_OUT,
238           i_en => w_STORE_INSTRUCTION,
```

```
        o_Q => w_instruction);
-- Connect instruction register related outputs
o_RS_SEL <= w_instruction(25 downto 21);
o_RT_SEL <= w_instruction(20 downto 16);
o_immediate <= w_instruction(15 downto 0);

-- Connect the State Register
u_State_Register: myRegN
generic map (m_width => 4)
port map (i_clk => i_clk,
        i_reset => i_reset,
        i_D => w_next_state,
        o_Q => w_state);

-- Connect the MUXes to determine the next state
u_myMUX2_INVALID: myMUX2_N
generic map (m_width => 4)
port map (i_0 => w_next_state0,
        i_1 => k_zero_4,
        i_S => w_INVALID_INSTR,
        o_Z => w_next_state);

u_myMUX2_next_state12: myMUX2_N
generic map (m_width => 4)
port map (i_0 => k_two_4,
        i_1 => k_one_4,
        i_S => i_MEM_READY,
        o_Z => w_next_state12);

u_myMUX2_next_state01: myMUX2_N
generic map (m_width => 4)
port map (i_0 => k_zero_4,
        i_1 => k_one_4,
        i_S => i_MEM_READY,
        o_Z => w_next_state01);

u_myMUX2_next_state67: myMUX2_N
generic map (m_width => 4)
port map (i_0 => k_six_4,
        i_1 => k_seven_4,
        i_S => i_MEM_READY,
        o_Z => w_next_state67);
```

```
282   u_myMUX2_next_state70: myMUX2_N
      generic map (m_width => 4)
284   port map (i_0 => k_ten_4,
              i_1 => k_seven_4,
286           i_S => i_MEM_READY,
              o_Z => w_next_state70);
288
      u_myMUX2_next_state89: myMUX2_N
290   generic map (m_width => 4)
      port map (i_0 => k_eight_4,
292           i_1 => k_nine_4,
              i_S => i_MEM_READY,
294           o_Z => w_next_state89);

296   u_myMUX2_next_state90: myMUX2_N
      generic map (m_width => 4)
298   port map (i_0 => k_ten_4,
              i_1 => k_nine_4,
300           i_S => i_MEM_READY,
              o_Z => w_next_state90);
302
      u_myMUX16_N: myMUX16_N
304   generic map (m_width => 4)
      port map (i_0 => w_next_state01,
306           i_1 => w_next_state12,
              i_2 => w_next_state2,
308           i_3 => k_zero_4,
              i_4 => k_five_4,
310           i_5 => k_zero_4,
              i_6 => w_next_state67,
312           i_7 => w_next_state70,
              i_8 => w_next_state89,
314           i_9 => w_next_state90,
              i_10 => k_zero_4,
316           i_11 => k_zero_4,
              i_12 => k_zero_4,
318           i_13 => k_zero_4,
              i_14 => k_zero_4,
320           i_15 => k_zero_4,
              i_S => w_state,
322           o_Z => w_next_state0);

324   u_myMUX4_INSTRUCTION_TYPE: myMUX4_N
```

```
    generic map (m_width => 4)
326 port map (i_0 => k_three_4,
              i_1 => k_four_4,
328           i_2 => k_six_4,
              i_3 => k_eight_4,
330           i_S => w_FSM_CTRL,
              o_Z => w_next_state2);
332 -- Connect the Control Signal Generator
    u_FSM_Control_Signal_Generator:
     FSM_Control_Signal_Generator
334 port map (i_state => w_state,
              o_MEM_READ => o_MEM_READ,
336           o_MEM_WRITE => o_MEM_WRITE,
              o_MEM_ADDRESS_SEL => o_MEM_ADDRESS_SEL,
338           o_STORE_FROM_MEM => o_STORE_FROM_MEM,
              o_PC_STORE => w_PC_STORE,
340           o_PC_EN => o_PC_EN,
              o_DO_NOT_STORE => w_DO_NOT_STORE,
342           o_BRANCH_OVERRIDE => w_BRANCH_OVERRIDE,
              o_STORE_INSTRUCTION => w_STORE_INSTRUCTION);
344 -- Connec the Instruction Decoder/Encoder
    u_Instruction_Dec_Enc: Instruction_Dec_Enc
346 port map (i_instr31 => w_instruction(31),
              i_instr30 => w_instruction(30),
348           i_instr29 => w_instruction(29),
              i_instr28 => w_instruction(28),
350           i_instr27 => w_instruction(27),
              i_instr26 => w_instruction(26),
352           i_instr20 => w_instruction(20),
              i_instr19 => w_instruction(19),
354           i_instr18 => w_instruction(18),
              i_instr17 => w_instruction(17),
356           i_instr16 => w_instruction(16),
              i_instr05 => w_instruction(5),
358           i_instr04 => w_instruction(4),
              i_instr03 => w_instruction(3),
360           i_instr02 => w_instruction(2),
              i_instr01 => w_instruction(1),
362           i_instr00 => w_instruction(0),
              i_BRANCH_OVERRIDE => w_BRANCH_OVERRIDE,
364           i_DO_NOT_STORE => w_DO_NOT_STORE,
              i_PC_STORE => w_PC_STORE,
366           o_ALU_SRC_A => o_ALU_SRC_A,
```

```
            o_ALU_SRC_B => o_ALU_SRC_B,
368         o_ALU_INV_B => o_ALU_INV_B,
            o_COMP_SEL => o_COMP_SEL,
370         o_OVER_CTRL => o_OVER_CTRL,
            o_ALU_OUTPUT => o_ALU_OUTPUT,
372         o_REG_SEL => o_REG_SEL,
            o_UNSIGNED => o_UNSIGNED,
374         o_overflow => o_overflow,
            o_imm_extend => o_imm_extend,
376         o_FSM_CTRL => w_FSM_CTRL,
            o_INVALID_INSTR => w_INVALID_INSTR);
378 end a_Controller_FSM;
```

Listing A.47. Controller_FSM.vhd Code

1.25 Basic MIPS Top Level

```
--| Basic_MIPS.vhd
--| Connect the Controller and Datapath of a "MIPS Like" architecture that implements the following
--| 33 commands. See MIPS Volume II, Revision 0.95 March 12, 2001 by MIPS Technologies Inc. for the
--| function of each of these instructions. One exception is that the branching instructions do not
--| implement the branch delay slot and instead, the 18-bit signed offset is added to the current PC
--| value to form a PC relative effective target address.
--| SLL - Shift Word Left Logical
--| NOP - No Operation
--| SRL - Shfit Word Right Logical
--| SRA - Shift Word Right Arithmetic
--| SLLV - Shift Word Left Logical Variable
--| SRLV - Shfit Word Right Logical Variable
--| SRAV - Shift Word Right Arithmetic Variable
--| ADD - Add Word
--| ADDU - Add Unsigned Word
--| SUB - Subtract Word
--| SUBU - Subtract Unsigned Word
--| AND - Bitwise And
--| OR - Bitwise Or
--| XOR - Bitwise Exclusive Or
--| NOR - Bitwise Nor
--| SLT - Set Less Than
--| SLTU - Set Less Than Unsigned
--| BGEZ - Branch on Greater Than or Equal to Zero
--| BLTZ - Branch on Less Than Zero
--| BEQ - Branch on Equal
--| BNE - Branch on Not Equal
--| BLEZ - Branch on Less Than or Equal to Zero
--| BGTZ - Branch on Greater Than Zero
--| ADDI - Add Immediate Word
--| ADDIU - Add Immediate Unsigned Word
--| SLTI - Set on Less Than Immediate
--| SLTIU - Set on Less Than Immediate Unsigned
--| ANDI - Bitwise And Immediate
--| ORI - Bitwise Or Immediate
--| XORI - Bitwise Exclusive Or Immediate
--| LUI - Load Upper Immediate
```

```
38 --| LW    - Load Word
   --| SW    - Store Word
40 --|
   --| INPUTS:
42 --| i_clk   - Clock input
   --| i_reset   - Reset input
44 --| i_MEM_OUT - Output from memory to be stored into the
   instruction register
   --|
46 --| OUTPUTS:
   --| o_MEM_READ        - Read from memory enable signal
48 --| o_MEM_WRITE     - Write to memory enable signal
   --| o_MEM_IN        - Output from Basic_MIPS to be stored in
   memory
50 --| o_MEM_ADDRESS     - Address used to access memory
   library IEEE;
52 use IEEE.std_logic_1164.all;

54 entity Basic_MIPS is
     port (i_clk          : in  std_logic;
56         i_reset        : in  std_logic;
           i_MEM_OUT      : in  std_logic_vector(31 downto 0);
58         i_MEM_READY      : in  std_logic;
           o_MEM_READ       : out std_logic;
60         o_MEM_WRITE      : out std_logic;
           o_MEM_IN         : out std_logic_vector(31 downto 0);
62         o_MEM_ADDRESS    : out std_logic_vector(31 downto 0));
   end Basic_MIPS;
64
   architecture a_Basic_MIPS of Basic_MIPS is
66 --| Define Components
     --| Controller Finite State Machine
68   component Controller_FSM is
       port (i_clk          : in  std_logic;
70           i_reset        : in  std_logic;
             i_MEM_OUT      : in  std_logic_vector(31 downto 0);
72           i_MEM_READY      : in std_logic;
             o_MEM_READ       : out std_logic;
74           o_MEM_WRITE      : out std_logic;
             o_MEM_ADDRESS_SEL : out std_logic;
76           o_STORE_FROM_MEM  : out std_logic;
             o_PC_EN          : out std_logic;
78           o_ALU_SRC_A      : out std_logic_vector(1 downto 0);
```

```
        o_ALU_SRC_B      : out std_logic_vector(1 downto 0);
        o_ALU_INV_B      : out std_logic;
        o_COMP_SEL       : out std_logic_vector(2 downto 0);
        o_OVER_CTRL      : out std_logic_vector(1 downto 0);
        o_ALU_OUTPUT     : out std_logic_vector(3 downto 0);
        o_REG_SEL      : out std_logic_vector(1 downto 0);
        o_UNSIGNED       : out std_logic;
        o_overflow       : out std_logic;
        o_imm_extend     : out std_logic;
        o_RS_SEL         : out std_logic_vector(4 downto 0);
        o_RT_SEL         : out std_logic_vector(4 downto 0);
        o_immediate      : out std_logic_vector(15 downto 0));
    end component;
    -- | Datapath
    component Datapath is
      port (i_clk          : in  std_logic;
            i_reset        : in  std_logic;
            i_ALU_SRC_A      : in  std_logic_vector(1 downto 0);
            i_ALU_SRC_B      : in  std_logic_vector(1 downto 0);
            i_ALU_INV_B      : in  std_logic;
            i_COMP_SEL       : in  std_logic_vector(2 downto 0);
            i_OVER_CTRL      : in  std_logic_vector(1 downto 0);
            i_ALU_OUTPUT     : in  std_logic_vector(3 downto 0);
            i_RS_SEL         : in  std_logic_vector(4 downto 0);
            i_RT_SEL         : in  std_logic_vector(4 downto 0);
            i_imm          : in  std_logic_vector(15 downto 0);
            i_UNSIGNED       : in  std_logic;
            i_overflow       : in  std_logic;
            i_imm_extend     : in  std_logic;
            i_STORE_FROM_MEM   : in  std_logic;
            i_REG_SEL      : in  std_logic_vector(1 downto 0);
            i_PC_EN          : in  std_logic;
            i_MEM_ADDRESS_SEL : in  std_logic;
            i_MEM_OUT      : in  std_logic_vector(31 downto 0);
            o_MEM_ADDRESS    : out std_logic_vector(31 downto 0);
            o_RT_DATA      : out std_logic_vector(31 downto 0));
    end component;

  --| Define Control Signals from Controller_FSM to Datapath
    signal w_ALU_SRC_A      : std_logic_vector(1 downto 0);
    signal w_ALU_SRC_B      : std_logic_vector(1 downto 0);
    signal w_ALU_INV_B      : std_logic;
    signal w_COMP_SEL       : std_logic_vector(2 downto 0);
```

```
122   signal w_OVER_CTRL       : std_logic_vector(1 downto 0);
      signal w_ALU_OUTPUT      : std_logic_vector(3 downto 0);
124   signal w_RS_SEL        : std_logic_vector(4 downto 0);
      signal w_RT_SEL        : std_logic_vector(4 downto 0);
126   signal w_immediate       : std_logic_vector(15 downto 0);
      signal w_UNSIGNED        : std_logic;
128   signal w_overflow        : std_logic;
      signal w_imm_extend      : std_logic;
130   signal w_STORE_FROM_MEM    : std_logic;
      signal w_REG_SEL         : std_logic_vector(1 downto 0);
132   signal w_PC_EN           : std_logic;
      signal w_MEM_ADDRESS_SEL   : std_logic;
134
    begin
136   --| Connect the Controller
      u_Controller_FSM : Controller_FSM
138   port map (i_clk => i_clk,
              i_reset => i_reset,
140           i_MEM_OUT => i_MEM_OUT,
              i_MEM_READY => i_MEM_READY,
142           o_MEM_READ => o_MEM_READ,
              o_MEM_WRITE => o_MEM_WRITE,
144           o_MEM_ADDRESS_SEL => w_MEM_ADDRESS_SEL,
              o_STORE_FROM_MEM => w_STORE_FROM_MEM,
146           o_PC_EN => w_PC_EN,
              o_ALU_SRC_A => w_ALU_SRC_A,
148           o_ALU_SRC_B => w_ALU_SRC_B,
              o_ALU_INV_B => w_ALU_INV_B,
150           o_COMP_SEL => w_COMP_SEL,
              o_OVER_CTRL => w_OVER_CTRL,
152           o_ALU_OUTPUT => w_ALU_OUTPUT,
              o_REG_SEL => w_REG_SEL,
154           o_UNSIGNED => w_UNSIGNED,
              o_overflow => w_overflow,
156           o_imm_extend => w_imm_extend,
              o_RS_SEL => w_RS_SEL,
158           o_RT_SEL => w_RT_SEL,
              o_immediate => w_immediate);
160
      --| Connect the Datapath
162   u_Datapath : Datapath
      port map (i_clk => i_clk,
164           i_reset => i_reset,
```

```
            i_ALU_SRC_A => w_ALU_SRC_A,
            i_ALU_SRC_B => w_ALU_SRC_B,
            i_ALU_INV_B => w_ALU_INV_B,
            i_COMP_SEL => w_COMP_SEL,
            i_OVER_CTRL => w_OVER_CTRL,
            i_ALU_OUTPUT => w_ALU_OUTPUT,
            i_RS_SEL => w_RS_SEL,
            i_RT_SEL => w_RT_SEL,
            i_imm => w_immediate,
            i_UNSIGNED => w_UNSIGNED,
            i_overflow => w_overflow,
            i_imm_extend => w_imm_extend,
            i_STORE_FROM_MEM => w_STORE_FROM_MEM,
            i_REG_SEL => w_REG_SEL,
            i_PC_EN => w_PC_EN,
            i_MEM_ADDRESS_SEL => w_MEM_ADDRESS_SEL,
            i_MEM_OUT => i_MEM_OUT,
            o_MEM_ADDRESS => o_MEM_ADDRESS,
            o_RT_DATA => o_MEM_IN);
end a_Basic_MIPS;
```

Listing A.48. BASIC_MIPS.vhd Code

Appendix B. Testbench VHDL Code

2.1 NAND

```
--| myNAND2_TB
--| Tests myNAND2
library IEEE;
use IEEE.std_logic_1164.all;

entity myNAND2_TB is
end myNAND2_TB;

architecture testbench of myNAND2_TB is
  -- Declare Components
  component myNAND2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare signals
  signal w_A : std_logic := '0';
  signal w_B : std_logic := '0';
  signal w_Z : std_logic;

begin
  -- Connect myNAND2
  u_myNAND2: myNAND2
  port map (i_A => w_A,
            i_B => w_B,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= '1';
    wait for 20 ns;
    w_B <= '1';
    wait for 20 ns;
    w_A <= '0';
    wait for 20 ns;
```

```
40      wait;
    end process stimulus;
42  end testbench;
```

Listing B.1. myNAND2_TB.vhd Code

2.2 Inverter

2.2.1 1-Bit Inverter

```
--| myINV_TB
--| Tests myINV
library IEEE;
use IEEE.std_logic_1164.all;

entity myINV_TB is
end myINV_TB;

architecture testbench of myINV_TB is
  component myINV is
    port (i_A : in  std_logic;
          o_Z : out std_logic
          );
  end component;
  -- Declare signals
  signal w_A : std_logic := '0';
  signal w_Z : std_logic;
begin
  -- Connect myINV
  u_myINV: myINV
  port map (i_A => w_A,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= '1';
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.2. myINV_TB.vhd Code

2.2.2 N-Bit Inverter

```
--| myINV_N_N_TB
--| Tests myINV_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myINV_N_TB is
end myINV_N_TB;

architecture testbench of myINV_N_TB is
  component myINV_N is
    generic (m_width : integer := 32);
    port (i_A : in  std_logic_vector(m_width-1 downto 0);
          o_Z : out std_logic_vector(m_width-1 downto 0)
          );
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_Z : std_logic_vector(31 downto 0);
begin
  -- Connect myNAND2
  u_myINV_N: myINV_N
  generic map(m_width => 32)
  port map (i_A => w_A,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"1111_0000_1111_0000_1111_0000_1111_0000";
    wait for 20 ns;
    w_A <= B"1111_1111_1111_1111_1111_1111_1111_1111";
    wait for 20 ns;
    w_A <= B"0000_0000_0000_0000_0000_0000_0000_0000";
    wait for 20 ns;
    w_A <= B"0000_0001_0010_0011_0100_0101_0110_0111";
    wait for 20 ns;
    w_A <= B"1000_1001_1010_1011_1100_1101_1110_1111";
    wait for 20 ns;
    wait;
```

```
  end process stimulus;
end testbench;
```

Listing B.3. myINV_N_TB.vhd Code

2.3 Multiplexers

2.3.1 2 Input, 1-Bit Multiplexer

```
--| myMUX2_1_TB
--| Tests myMUX2_1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX2_1_TB is
end myMUX2_1_TB;

architecture testbench of myMUX2_1_TB is
  -- Declare Components
  component myMUX2_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_S : in  std_logic;
          o_Z : out std_logic
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic := '0';
  signal w_1 : std_logic := '1';
  signal w_S : std_logic := '0';
  signal w_Z : std_logic;

begin
  -- Connect myMUX2_1
  u_myMUX2_1: myMUX2_1
  port map (i_0 => w_0,
            i_1 => w_1,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= '1';
    wait for 20 ns;
    w_0 <= '1';
    w_1 <= '0';
```

```
    wait for 20 ns;
42  w_S <= '0';
    wait for 20 ns;
44  wait;
  end process stimulus;
46 end testbench;
```

Listing B.4. myMUX2_1_TB.vhd Code

2.3.2 2 Input, N-Bit Multiplexer

```
--| myMUX2_N_TB
--| Tests myMUX2_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX2_N_TB is
end myMUX2_N_TB;

architecture testbench of myMUX2_N_TB is
  -- Declare Components
  component myMUX2_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
          i_S : in  std_logic;
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_1111_1111_1111_1111";
  signal w_1 : std_logic_vector(31 downto 0) := B"1111
   _1111_1111_1111_0000_0000_0000_0000";
  signal w_S : std_logic := '0';
  signal w_Z : std_logic_vector(31 downto 0);

begin
  -- Connect myMUX2_N
  u_myMUX2_N: myMUX2_N
  generic map (m_width => 32)
  port map (i_0 => w_0,
            i_1 => w_1,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= '1';
```

```
40      wait for 20 ns;
        w_0 <= B"1111_0000_1111_0000_1111_0000_1111_0000";
42      w_1 <= B"0000_1111_0000_1111_0000_1111_0000_1111";
        wait for 20 ns;
44      w_S <= '0';
        wait for 20 ns;
46      wait;
    end process stimulus;
48  end testbench;
```

Listing B.5. myMUX2_N_TB.vhd Code

2.3.3 4 Input, 1-Bit Multiplexer

```
--| myMUX4_1_TB
--| Tests myMUX4_1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX4_1_TB is
end myMUX4_1_TB;

architecture testbench of myMUX4_1_TB is
  -- Declare Components
  component myMUX4_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_S : in  std_logic_vector(1 downto 0);
          o_Z : out std_logic
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic := '0';
  signal w_1 : std_logic := '1';
  signal w_2 : std_logic := '1';
  signal w_3 : std_logic := '0';
  signal w_S : std_logic_vector(1 downto 0) := B"00";
  signal w_Z : std_logic;

begin
  -- Connect myMUX4_1
  u_myMUX4_1: myMUX4_1
  port map (i_0 => w_0,
            i_1 => w_1,
            i_2 => w_2,
            i_3 => w_3,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
```

```
42      wait for 20 ns;
        w_S <= B"01";
44      wait for 20 ns;
        w_S <= B"10";
46      wait for 20 ns;
        w_S <= B"11";
48      wait for 20 ns;
        wait;
50    end process stimulus;
  end testbench;
```

Listing B.6. myMUX4_1_TB.vhd Code

2.3.4 4 Input, N-Bit Multiplexer

```
--| myMUX4_N_TB
--| Tests myMUX4_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX4_N_TB is
end myMUX4_N_TB;

architecture testbench of myMUX4_N_TB is
  -- Declare Components
  component myMUX4_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
          i_2 : in  std_logic_vector(31 downto 0);
          i_3 : in  std_logic_vector(31 downto 0);
          i_S : in  std_logic_vector(1 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_1111_1111";
  signal w_1 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_1111_1111_0000_0000";
  signal w_2 : std_logic_vector(31 downto 0) := B"0000
   _0000_1111_1111_0000_0000_0000_0000";
  signal w_3 : std_logic_vector(31 downto 0) := B"1111
   _1111_0000_0000_0000_0000_0000_0000";
  signal w_S : std_logic_vector(1 downto 0) := (others =>
   '0');
  signal w_Z : std_logic_vector(31 downto 0);

begin
  -- Connect myMUX4_N
  u_myMUX4_N: myMUX4_N
  generic map (m_width => 32)
  port map (i_0 => w_0,
            i_1 => w_1,
            i_2 => w_2,
```

```
37            i_3 => w_3,
              i_S => w_S,
39            o_Z => w_Z);

41    -- Process for stimulating inputs
      stimulus: process is
43    begin
        wait for 20 ns;
45      w_S <= B"01";
        wait for 20 ns;
47      w_S <= B"10";
        wait for 20 ns;
49      w_S <= B"11";
        wait for 20 ns;
51      wait;
      end process stimulus;
53  end testbench;
```

Listing B.7. myMUX4_N_TB.vhd Code

2.3.5 8 Input, 1-Bit Multiplexer

```
--| myMUX8_1_TB
--| Tests myMUX8_1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX8_1_TB is
end myMUX8_1_TB;

architecture testbench of myMUX8_1_TB is
  -- Declare Components
  component myMUX8_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_S : in  std_logic_vector(2 downto 0);
          o_Z : out std_logic
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic := '0';
  signal w_1 : std_logic := '1';
  signal w_2 : std_logic := '1';
  signal w_3 : std_logic := '0';
  signal w_4 : std_logic := '1';
  signal w_5 : std_logic := '0';
  signal w_6 : std_logic := '1';
  signal w_7 : std_logic := '0';
  signal w_S : std_logic_vector(2 downto 0) := B"000";
  signal w_Z : std_logic;

begin
  -- Connect myMUX8_1
  u_myMUX8_1: myMUX8_1
  port map (i_0 => w_0,
            i_1 => w_1,
```

```
            i_2 => w_2,
            i_3 => w_3,
            i_4 => w_4,
            i_5 => w_5,
            i_6 => w_6,
            i_7 => w_7,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= B"001";
    wait for 20 ns;
    w_S <= B"010";
    wait for 20 ns;
    w_S <= B"011";
    wait for 20 ns;
    w_S <= B"100";
    wait for 20 ns;
    w_S <= B"101";
    wait for 20 ns;
    w_S <= B"110";
    wait for 20 ns;
    w_S <= B"111";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.8. myMUX8_1_TB.vhd Code

2.3.6 8 Input, N-Bit Multiplexer

```
--| myMUX8_N_TB
--| Tests myMUX8_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX8_N_TB is
end myMUX8_N_TB;

architecture testbench of myMUX8_N_TB is
  -- Declare Components
  component myMUX8_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
          i_2 : in  std_logic_vector(31 downto 0);
          i_3 : in  std_logic_vector(31 downto 0);
          i_4 : in  std_logic_vector(31 downto 0);
          i_5 : in  std_logic_vector(31 downto 0);
          i_6 : in  std_logic_vector(31 downto 0);
          i_7 : in  std_logic_vector(31 downto 0);
          i_S : in  std_logic_vector(2 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_0000_1111";
  signal w_1 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_1111_0000";
  signal w_2 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_1111_0000_0000";
  signal w_3 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_1111_0000_0000_0000";
  signal w_4 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_1111_0000_0000_0000_0000";
  signal w_5 : std_logic_vector(31 downto 0) := B"0000
   _0000_1111_0000_0000_0000_0000_0000";
  signal w_6 : std_logic_vector(31 downto 0) := B"0000
   _1111_0000_0000_0000_0000_0000_0000";
```

```
  signal w_7 : std_logic_vector(31 downto 0) := B"1111
   _0000_0000_0000_0000_0000_0000_0000";
  signal w_S : std_logic_vector(2 downto 0) := (others =>
   '0');
  signal w_Z : std_logic_vector(31 downto 0);

begin
  -- Connect myMUX8_N
  u_myMUX8_N: myMUX8_N
  generic map (m_width => 32)
  port map (i_0 => w_0,
            i_1 => w_1,
            i_2 => w_2,
            i_3 => w_3,
            i_4 => w_4,
            i_5 => w_5,
            i_6 => w_6,
            i_7 => w_7,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= B"001";
    wait for 20 ns;
    w_S <= B"010";
    wait for 20 ns;
    w_S <= B"011";
    wait for 20 ns;
    w_S <= B"100";
    wait for 20 ns;
    w_S <= B"101";
    wait for 20 ns;
    w_S <= B"110";
    wait for 20 ns;
    w_S <= B"111";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.9. myMUX8_N_TB.vhd Code

2.3.7 16 Input, 1-Bit Multiplexer

```
--| myMUX16_1_TB
--| Tests myMUX16_1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX16_1_TB is
end myMUX16_1_TB;

architecture testbench of myMUX16_1_TB is
  -- Declare Components
  component myMUX16_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          i_9 : in  std_logic;
          i_10  : in  std_logic;
          i_11  : in  std_logic;
          i_12  : in  std_logic;
          i_13  : in  std_logic;
          i_14  : in  std_logic;
          i_15  : in  std_logic;
          i_S : in  std_logic_vector(3 downto 0);
          o_Z : out std_logic
          );
  end component;

  -- Declare signals
  signal w_0  : std_logic := '0';
  signal w_1  : std_logic := '1';
  signal w_2  : std_logic := '1';
  signal w_3  : std_logic := '0';
  signal w_4  : std_logic := '1';
  signal w_5  : std_logic := '0';
  signal w_6  : std_logic := '1';
  signal w_7  : std_logic := '0';
```

```
    signal w_8  : std_logic := '0';
    signal w_9  : std_logic := '1';
    signal w_10 : std_logic := '0';
    signal w_11 : std_logic := '1';
    signal w_12 : std_logic := '1';
    signal w_13 : std_logic := '0';
    signal w_14 : std_logic := '0';
    signal w_15 : std_logic := '1';
    signal w_S  : std_logic_vector(3 downto 0) := B"0000";
    signal w_Z  : std_logic;

begin
  -- Connect myMUX16_1
  u_myMUX16_1: myMUX16_1
  port map (i_0 => w_0,
            i_1 => w_1,
            i_2 => w_2,
            i_3 => w_3,
            i_4 => w_4,
            i_5 => w_5,
            i_6 => w_6,
            i_7 => w_7,
            i_8 => w_8,
            i_9 => w_9,
            i_10 => w_10,
            i_11 => w_11,
            i_12 => w_12,
            i_13 => w_13,
            i_14 => w_14,
            i_15 => w_15,
            i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= B"0001";
    wait for 20 ns;
    w_S <= B"0010";
    wait for 20 ns;
    w_S <= B"0011";
    wait for 20 ns;
```

```
85      w_S <= B"0100";
        wait for 20 ns;
87      w_S <= B"0101";
        wait for 20 ns;
89      w_S <= B"0110";
        wait for 20 ns;
91      w_S <= B"0111";
        wait for 20 ns;
93      w_S <= B"1000";
        wait for 20 ns;
95      w_S <= B"1001";
        wait for 20 ns;
97      w_S <= B"1010";
        wait for 20 ns;
99      w_S <= B"1011";
        wait for 20 ns;
101     w_S <= B"1100";
        wait for 20 ns;
103     w_S <= B"1101";
        wait for 20 ns;
105     w_S <= B"1110";
        wait for 20 ns;
107     w_S <= B"1111";
        wait for 20 ns;
109     wait;
    end process stimulus;
111 end testbench;
```

Listing B.10. myMUX16_1_TB.vhd Code

2.3.8 16 Input, N-Bit Multiplexer

```
--| myMUX16_N_TB
--| Tests myMUX16_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX16_N_TB is
end myMUX16_N_TB;

architecture testbench of myMUX16_N_TB is
  -- Declare Components
  component myMUX16_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
          i_2 : in  std_logic_vector(31 downto 0);
          i_3 : in  std_logic_vector(31 downto 0);
          i_4 : in  std_logic_vector(31 downto 0);
          i_5 : in  std_logic_vector(31 downto 0);
          i_6 : in  std_logic_vector(31 downto 0);
          i_7 : in  std_logic_vector(31 downto 0);
          i_8 : in  std_logic_vector(31 downto 0);
          i_9 : in  std_logic_vector(31 downto 0);
          i_10 : in  std_logic_vector(31 downto 0);
          i_11 : in  std_logic_vector(31 downto 0);
          i_12 : in  std_logic_vector(31 downto 0);
          i_13 : in  std_logic_vector(31 downto 0);
          i_14 : in  std_logic_vector(31 downto 0);
          i_15 : in  std_logic_vector(31 downto 0);
          i_S : in  std_logic_vector(3 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_0 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_0000_0011";
  signal w_1 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_0000_1100";
  signal w_2 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_0011_0000";
```

```
  signal w_3 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_1100_0000";
  signal w_4 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0011_0000_0000";
  signal w_5 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_1100_0000_0000";
  signal w_6 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0011_0000_0000_0000";
  signal w_7 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_1100_0000_0000_0000";
  signal w_8 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0011_0000_0000_0000_0000";
  signal w_9 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_1100_0000_0000_0000_0000";
  signal w_10 : std_logic_vector(31 downto 0) := B"0000
   _0000_0011_0000_0000_0000_0000_0000";
  signal w_11 : std_logic_vector(31 downto 0) := B"0000
   _0000_1100_0000_0000_0000_0000_0000";
  signal w_12 : std_logic_vector(31 downto 0) := B"0000
   _0011_0000_0000_0000_0000_0000_0000";
  signal w_13 : std_logic_vector(31 downto 0) := B"0000
   _1100_0000_0000_0000_0000_0000_0000";
  signal w_14 : std_logic_vector(31 downto 0) := B"0011
   _0000_0000_0000_0000_0000_0000_0000";
  signal w_15 : std_logic_vector(31 downto 0) := B"1100
   _0000_0000_0000_0000_0000_0000_0000";
  signal w_S : std_logic_vector(3 downto 0) := (others =>
   '0');
  signal w_Z : std_logic_vector(31 downto 0);

begin
  -- Connect myMUX16_N
  u_myMUX16_N: myMUX16_N
  generic map (m_width => 32)
  port map (i_0 => w_0,
            i_1 => w_1,
            i_2 => w_2,
            i_3 => w_3,
            i_4 => w_4,
            i_5 => w_5,
            i_6 => w_6,
            i_7 => w_7,
            i_8 => w_8,
```

```
67            i_9 => w_9,
              i_10 => w_10,
69            i_11 => w_11,
              i_12 => w_12,
71            i_13 => w_13,
              i_14 => w_14,
73            i_15 => w_15,
              i_S => w_S,
75            o_Z => w_Z);

77    -- Process for stimulating inputs
      stimulus: process is
79    begin
        wait for 20 ns;
81      w_S <= B"0001";
        wait for 20 ns;
83      w_S <= B"0010";
        wait for 20 ns;
85      w_S <= B"0011";
        wait for 20 ns;
87      w_S <= B"0100";
        wait for 20 ns;
89      w_S <= B"0101";
        wait for 20 ns;
91      w_S <= B"0110";
        wait for 20 ns;
93      w_S <= B"0111";
        wait for 20 ns;
95      w_S <= B"1000";
        wait for 20 ns;
97      w_S <= B"1001";
        wait for 20 ns;
99      w_S <= B"1010";
        wait for 20 ns;
101     w_S <= B"1011";
        wait for 20 ns;
103     w_S <= B"1100";
        wait for 20 ns;
105     w_S <= B"1101";
        wait for 20 ns;
107     w_S <= B"1110";
        wait for 20 ns;
109     w_S <= B"1111";
```

```
    wait for 20 ns;
111     wait;
  end process stimulus;
113 end testbench;
```

Listing B.11. myMUX16_N_TB.vhd Code

2.3.9 32 Input, 1-Bit Multiplexer

```
--| myMUX32_1_TB
--| Tests myMUX32_1
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX32_1_TB is
end myMUX32_1_TB;

architecture testbench of myMUX32_1_TB is
  -- Declare Components
  component myMUX32_1 is
    port (i_0 : in  std_logic;
          i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          i_9 : in  std_logic;
          i_10  : in  std_logic;
          i_11  : in  std_logic;
          i_12  : in  std_logic;
          i_13  : in  std_logic;
          i_14  : in  std_logic;
          i_15  : in  std_logic;
          i_16  : in  std_logic;
          i_17  : in  std_logic;
          i_18  : in  std_logic;
          i_19  : in  std_logic;
          i_20  : in  std_logic;
          i_21  : in  std_logic;
          i_22  : in  std_logic;
          i_23  : in  std_logic;
          i_24  : in  std_logic;
          i_25  : in  std_logic;
          i_26  : in  std_logic;
          i_27  : in  std_logic;
          i_28  : in  std_logic;
          i_29  : in  std_logic;
```

```
        i_30  : in  std_logic;
        i_31  : in  std_logic;
        i_S : in  std_logic_vector (4 downto 0);
        o_Z : out std_logic
        );
end component;

-- Declare signals
signal w_0  : std_logic := '0';
signal w_1  : std_logic := '1';
signal w_2  : std_logic := '1';
signal w_3  : std_logic := '0';
signal w_4  : std_logic := '1';
signal w_5  : std_logic := '0';
signal w_6  : std_logic := '1';
signal w_7  : std_logic := '0';
signal w_8  : std_logic := '0';
signal w_9  : std_logic := '1';
signal w_10 : std_logic := '0';
signal w_11 : std_logic := '1';
signal w_12 : std_logic := '1';
signal w_13 : std_logic := '0';
signal w_14 : std_logic := '0';
signal w_15 : std_logic := '1';
signal w_16 : std_logic := '0';
signal w_17 : std_logic := '0';
signal w_18 : std_logic := '1';
signal w_19 : std_logic := '1';
signal w_20 : std_logic := '1';
signal w_21 : std_logic := '1';
signal w_22 : std_logic := '0';
signal w_23 : std_logic := '0';
signal w_24 : std_logic := '1';
signal w_25 : std_logic := '1';
signal w_26 : std_logic := '1';
signal w_27 : std_logic := '0';
signal w_28 : std_logic := '1';
signal w_29 : std_logic := '1';
signal w_30 : std_logic := '0';
signal w_31 : std_logic := '1';
signal w_S  : std_logic_vector(4 downto 0) := B"00000";
signal w_Z  : std_logic;
```

```
85  begin
      -- Connect myMUX32_1
87    u_myMUX32_1: myMUX32_1
      port map (i_0 => w_0,
89              i_1 => w_1,
                i_2 => w_2,
91              i_3 => w_3,
                i_4 => w_4,
93              i_5 => w_5,
                i_6 => w_6,
95              i_7 => w_7,
                i_8 => w_8,
97              i_9 => w_9,
                i_10 => w_10,
99              i_11 => w_11,
                i_12 => w_12,
101             i_13 => w_13,
                i_14 => w_14,
103             i_15 => w_15,
                i_16 => w_16,
105             i_17 => w_17,
                i_18 => w_18,
107             i_19 => w_19,
                i_20 => w_20,
109             i_21 => w_21,
                i_22 => w_22,
111             i_23 => w_23,
                i_24 => w_24,
113             i_25 => w_25,
                i_26 => w_26,
115             i_27 => w_27,
                i_28 => w_28,
117             i_29 => w_29,
                i_30 => w_30,
119             i_31 => w_31,
                i_S => w_S,
121             o_Z => w_Z);

123   -- Process for stimulating inputs
      stimulus: process is
125   begin
        wait for 20 ns;
127     w_S <= B"00001";
```

```
        wait for 20 ns;
129     w_S <= B"00010";
        wait for 20 ns;
131     w_S <= B"00011";
        wait for 20 ns;
133     w_S <= B"00100";
        wait for 20 ns;
135     w_S <= B"00101";
        wait for 20 ns;
137     w_S <= B"00110";
        wait for 20 ns;
139     w_S <= B"00111";
        wait for 20 ns;
141     w_S <= B"01000";
        wait for 20 ns;
143     w_S <= B"01001";
        wait for 20 ns;
145     w_S <= B"01010";
        wait for 20 ns;
147     w_S <= B"01011";
        wait for 20 ns;
149     w_S <= B"01100";
        wait for 20 ns;
151     w_S <= B"01101";
        wait for 20 ns;
153     w_S <= B"01110";
        wait for 20 ns;
155     w_S <= B"01111";
        wait for 20 ns;
157     w_S <= B"10000";
        wait for 20 ns;
159     w_S <= B"10001";
        wait for 20 ns;
161     w_S <= B"10010";
        wait for 20 ns;
163     w_S <= B"10011";
        wait for 20 ns;
165     w_S <= B"10100";
        wait for 20 ns;
167     w_S <= B"10101";
        wait for 20 ns;
169     w_S <= B"10110";
        wait for 20 ns;
```

```
171     w_S <= B"10111";
        wait for 20 ns;
173     w_S <= B"11000";
        wait for 20 ns;
175     w_S <= B"11001";
        wait for 20 ns;
177     w_S <= B"11010";
        wait for 20 ns;
179     w_S <= B"11011";
        wait for 20 ns;
181     w_S <= B"11100";
        wait for 20 ns;
183     w_S <= B"11101";
        wait for 20 ns;
185     w_S <= B"11110";
        wait for 20 ns;
187     w_S <= B"11111";
        wait for 20 ns;
189     wait;
    end process stimulus;
191 end testbench;
```

Listing B.12. myMUX32_1_TB.vhd Code

2.3.10 32 Input, N-Bit Multiplexer

```
--| myMUX32_N_TB
--| Tests myMUX32_N
library IEEE;
use IEEE.std_logic_1164.all;

entity myMUX32_N_TB is
end myMUX32_N_TB;

architecture testbench of myMUX32_N_TB is
  -- Declare Components
  component myMUX32_N is
    generic (m_width : integer := 32);
    port (i_0 : in  std_logic_vector(31 downto 0);
          i_1 : in  std_logic_vector(31 downto 0);
          i_2 : in  std_logic_vector(31 downto 0);
          i_3 : in  std_logic_vector(31 downto 0);
          i_4 : in  std_logic_vector(31 downto 0);
          i_5 : in  std_logic_vector(31 downto 0);
          i_6 : in  std_logic_vector(31 downto 0);
          i_7 : in  std_logic_vector(31 downto 0);
          i_8 : in  std_logic_vector(31 downto 0);
          i_9 : in  std_logic_vector(31 downto 0);
          i_10 : in  std_logic_vector(31 downto 0);
          i_11 : in  std_logic_vector(31 downto 0);
          i_12 : in  std_logic_vector(31 downto 0);
          i_13 : in  std_logic_vector(31 downto 0);
          i_14 : in  std_logic_vector(31 downto 0);
          i_15 : in  std_logic_vector(31 downto 0);
          i_16 : in  std_logic_vector(31 downto 0);
          i_17 : in  std_logic_vector(31 downto 0);
          i_18 : in  std_logic_vector(31 downto 0);
          i_19 : in  std_logic_vector(31 downto 0);
          i_20 : in  std_logic_vector(31 downto 0);
          i_21 : in  std_logic_vector(31 downto 0);
          i_22 : in  std_logic_vector(31 downto 0);
          i_23 : in  std_logic_vector(31 downto 0);
          i_24 : in  std_logic_vector(31 downto 0);
          i_25 : in  std_logic_vector(31 downto 0);
          i_26 : in  std_logic_vector(31 downto 0);
          i_27 : in  std_logic_vector(31 downto 0);
          i_28 : in  std_logic_vector(31 downto 0);
```

```
        i_29 : in  std_logic_vector(31 downto 0);
        i_30 : in  std_logic_vector(31 downto 0);
        i_31 : in  std_logic_vector(31 downto 0);
        i_S : in  std_logic_vector(4 downto 0);
        o_Z : out std_logic_vector(31 downto 0)
        );
end component;

-- Declare signals
signal w_0 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0000_0001";
signal w_1 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0000_0010";
signal w_2 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0000_0100";
signal w_3 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0000_1000";
signal w_4 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0001_0000";
signal w_5 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0010_0000";
signal w_6 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_0100_0000";
signal w_7 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0000_1000_0000";
signal w_8 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0001_0000_0000";
signal w_9 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0010_0000_0000";
signal w_10 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_0100_0000_0000";
signal w_11 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0000_1000_0000_0000";
signal w_12 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0001_0000_0000_0000";
signal w_13 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0010_0000_0000_0000";
signal w_14 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_0100_0000_0000_0000";
signal w_15 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0000_1000_0000_0000_0000";
signal w_16 : std_logic_vector(31 downto 0) := B"0000
 _0000_0000_0001_0000_0000_0000_0000";
```

```
  signal w_17 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0010_0000_0000_0000_0000";
69 signal w_18 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0100_0000_0000_0000_0000";
   signal w_19 : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_1000_0000_0000_0000_0000";
71 signal w_20 : std_logic_vector(31 downto 0) := B"0000
   _0000_0001_0000_0000_0000_0000_0000";
   signal w_21 : std_logic_vector(31 downto 0) := B"0000
   _0000_0010_0000_0000_0000_0000_0000";
73 signal w_22 : std_logic_vector(31 downto 0) := B"0000
   _0000_0100_0000_0000_0000_0000_0000";
   signal w_23 : std_logic_vector(31 downto 0) := B"0000
   _0000_1000_0000_0000_0000_0000_0000";
75 signal w_24 : std_logic_vector(31 downto 0) := B"0000
   _0001_0000_0000_0000_0000_0000_0000";
   signal w_25 : std_logic_vector(31 downto 0) := B"0000
   _0010_0000_0000_0000_0000_0000_0000";
77 signal w_26 : std_logic_vector(31 downto 0) := B"0000
   _0100_0000_0000_0000_0000_0000_0000";
   signal w_27 : std_logic_vector(31 downto 0) := B"0000
   _1000_0000_0000_0000_0000_0000_0000";
79 signal w_28 : std_logic_vector(31 downto 0) := B"0001
   _0000_0000_0000_0000_0000_0000_0000";
   signal w_29 : std_logic_vector(31 downto 0) := B"0010
   _0000_0000_0000_0000_0000_0000_0000";
81 signal w_30 : std_logic_vector(31 downto 0) := B"0100
   _0000_0000_0000_0000_0000_0000_0000";
   signal w_31 : std_logic_vector(31 downto 0) := B"1000
   _0000_0000_0000_0000_0000_0000_0000";
83 signal w_S : std_logic_vector(4 downto 0) := (others =>
   '0');
   signal w_Z : std_logic_vector(31 downto 0);
85
begin
87 -- Connect myMUX32_N
   u_myMUX32_N: myMUX32_N
89 generic map (m_width => 32)
   port map (i_0 => w_0,
91         i_1 => w_1,
          i_2 => w_2,
93        i_3 => w_3,
          i_4 => w_4,
```

```
95            i_5 => w_5,
              i_6 => w_6,
97            i_7 => w_7,
              i_8 => w_8,
99            i_9 => w_9,
              i_10 => w_10,
101           i_11 => w_11,
              i_12 => w_12,
103           i_13 => w_13,
              i_14 => w_14,
105           i_15 => w_15,
              i_16 => w_16,
107           i_17 => w_17,
              i_18 => w_18,
109           i_19 => w_19,
              i_20 => w_20,
111           i_21 => w_21,
              i_22 => w_22,
113           i_23 => w_23,
              i_24 => w_24,
115           i_25 => w_25,
              i_26 => w_26,
117           i_27 => w_27,
              i_28 => w_28,
119           i_29 => w_29,
              i_30 => w_30,
121           i_31 => w_31,
              i_S => w_S,
123           o_Z => w_Z);

125   -- Process for stimulating inputs
      stimulus: process is
127   begin
        wait for 20 ns;
129     w_S <= B"00001";
        wait for 20 ns;
131     w_S <= B"00010";
        wait for 20 ns;
133     w_S <= B"00011";
        wait for 20 ns;
135     w_S <= B"00100";
        wait for 20 ns;
137     w_S <= B"00101";
```

```
        wait for 20 ns;
139     w_S <= B"00110";
        wait for 20 ns;
141     w_S <= B"00111";
        wait for 20 ns;
143     w_S <= B"01000";
        wait for 20 ns;
145     w_S <= B"01001";
        wait for 20 ns;
147     w_S <= B"01010";
        wait for 20 ns;
149     w_S <= B"01011";
        wait for 20 ns;
151     w_S <= B"01100";
        wait for 20 ns;
153     w_S <= B"01101";
        wait for 20 ns;
155     w_S <= B"01110";
        wait for 20 ns;
157     w_S <= B"01111";
        wait for 20 ns;
159     w_S <= B"10000";
        wait for 20 ns;
161     w_S <= B"10001";
        wait for 20 ns;
163     w_S <= B"10010";
        wait for 20 ns;
165     w_S <= B"10011";
        wait for 20 ns;
167     w_S <= B"10100";
        wait for 20 ns;
169     w_S <= B"10101";
        wait for 20 ns;
171     w_S <= B"10110";
        wait for 20 ns;
173     w_S <= B"10111";
        wait for 20 ns;
175     w_S <= B"11000";
        wait for 20 ns;
177     w_S <= B"11001";
        wait for 20 ns;
179     w_S <= B"11010";
        wait for 20 ns;
```

```
181     w_S <= B"11011";
        wait for 20 ns;
183     w_S <= B"11100";
        wait for 20 ns;
185     w_S <= B"11101";
        wait for 20 ns;
187     w_S <= B"11110";
        wait for 20 ns;
189     w_S <= B"11111";
        wait for 20 ns;
191     wait;
    end process stimulus;
193 end testbench;
```

Listing B.13. myMUX32_N_TB.vhd Code

2.4 Left Shift

```
--| LEFT_SHIFT_TB
--| Tests LEFT_SHIFT
library IEEE;
use IEEE.std_logic_1164.all;

entity LEFT_SHIFT_TB is
end LEFT_SHIFT_TB;

architecture testbench of LEFT_SHIFT_TB is
  -- Declare Components
  component LEFT_SHIFT is
    port (i_A  : in  std_logic_vector(31 downto 0);
          i_SA : in  std_logic_vector(4 downto 0);
          o_Z  : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_A  : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_0000_0000_0000_0001";
  signal w_SA : std_logic_vector(4 downto 0) := (others =>
   '0');
  signal w_Z  : std_logic_vector(31 downto 0);

begin
  -- Connect LEFT_SHIFT
  u_LEFT_SHIFT: LEFT_SHIFT
  port map (i_A => w_A,
            i_SA => w_SA,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_SA <= B"00001";
    wait for 20 ns;
    w_SA <= B"00010";
    wait for 20 ns;
    w_SA <= B"00011";
    wait for 20 ns;
```

```
        w_SA <= B"00100";
41      wait for 20 ns;
        w_SA <= B"00101";
43      wait for 20 ns;
        w_SA <= B"00110";
45      wait for 20 ns;
        w_SA <= B"00111";
47      wait for 20 ns;
        w_SA <= B"01000";
49      wait for 20 ns;
        w_SA <= B"01001";
51      wait for 20 ns;
        w_SA <= B"01010";
53      wait for 20 ns;
        w_SA <= B"01011";
55      wait for 20 ns;
        w_SA <= B"01100";
57      wait for 20 ns;
        w_SA <= B"01101";
59      wait for 20 ns;
        w_SA <= B"01110";
61      wait for 20 ns;
        w_SA <= B"01111";
63      wait for 20 ns;
        w_SA <= B"10000";
65      wait for 20 ns;
        w_SA <= B"10001";
67      wait for 20 ns;
        w_SA <= B"10010";
69      wait for 20 ns;
        w_SA <= B"10011";
71      wait for 20 ns;
        w_SA <= B"10100";
73      wait for 20 ns;
        w_SA <= B"10101";
75      wait for 20 ns;
        w_SA <= B"10110";
77      wait for 20 ns;
        w_SA <= B"10111";
79      wait for 20 ns;
        w_SA <= B"11000";
81      wait for 20 ns;
        w_SA <= B"11001";
```

```
83      wait for 20 ns;
        w_SA <= B"11010";
85      wait for 20 ns;
        w_SA <= B"11011";
87      wait for 20 ns;
        w_SA <= B"11100";
89      wait for 20 ns;
        w_SA <= B"11101";
91      wait for 20 ns;
        w_SA <= B"11110";
93      wait for 20 ns;
        w_SA <= B"11111";
95      wait for 20 ns;
        wait;
97    end process stimulus;
  end testbench;
```

Listing B.14. LEFT_SHIFT_TB.vhd Code

2.5 Left Shift 2

```
--| SLL2_TB.vhd
--| Test the SLL2.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity SLL2_32_TB is
end SLL2_32_TB;

architecture testbench of SLL2_32_TB is
--| Declare components
  component SLL2_32 is
    port (i_A : in  std_logic_vector(31 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

--| Declare Signals
  signal w_A : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_1111_0000_1111_0101_1001";
  signal w_Z : std_logic_vector(31 downto 0);
begin
  u_SLL2_32: SLL2_32
  port map(i_A => w_A,
          o_Z => w_Z);

  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"1111_0000_1111_0000_1100_1111_0101_1001";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.15. SLL2_32_TB.vhd Code

2.6 Right Shift

```
--| RIGHT_SHIFT_TB
--| Tests RIGHT_SHIFT
library IEEE;
use IEEE.std_logic_1164.all;

entity RIGHT_SHIFT_TB is
end RIGHT_SHIFT_TB;

architecture testbench of RIGHT_SHIFT_TB is
  -- Declare Components
  component RIGHT_SHIFT is
    port (i_A    : in  std_logic_vector(31 downto 0);
          i_SA   : in  std_logic_vector(4 downto 0);
          i_ARITH : in std_logic;
          o_Z    : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_A  : std_logic_vector(31 downto 0) := B"1111
   _1111_1111_1111_1111_1111_1111_1111";
  signal w_SA : std_logic_vector(4 downto 0) := (others =>
   '0');
  signal w_ARITH : std_logic := '0';
  signal w_Z  : std_logic_vector(31 downto 0);

begin
  -- Connect RIGHT_SHIFT
  u_RIGHT_SHIFT: RIGHT_SHIFT
  port map (i_A => w_A,
            i_SA => w_SA,
            i_ARITH => w_ARITH,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_SA <= B"00001";
    wait for 20 ns;
    w_SA <= B"00010";
```

```
        wait for 20 ns;
41      w_SA <= B"00011";
        wait for 20 ns;
43      w_SA <= B"00100";
        wait for 20 ns;
45      w_SA <= B"00101";
        wait for 20 ns;
47      w_SA <= B"00110";
        wait for 20 ns;
49      w_SA <= B"00111";
        wait for 20 ns;
51      w_SA <= B"01000";
        wait for 20 ns;
53      w_SA <= B"01001";
        wait for 20 ns;
55      w_SA <= B"01010";
        wait for 20 ns;
57      w_SA <= B"01011";
        wait for 20 ns;
59      w_SA <= B"01100";
        wait for 20 ns;
61      w_SA <= B"01101";
        wait for 20 ns;
63      w_SA <= B"01110";
        wait for 20 ns;
65      w_SA <= B"01111";
        wait for 20 ns;
67      w_SA <= B"10000";
        wait for 20 ns;
69      w_SA <= B"10001";
        wait for 20 ns;
71      w_SA <= B"10010";
        wait for 20 ns;
73      w_SA <= B"10011";
        wait for 20 ns;
75      w_SA <= B"10100";
        wait for 20 ns;
77      w_SA <= B"10101";
        wait for 20 ns;
79      w_SA <= B"10110";
        wait for 20 ns;
81      w_SA <= B"10111";
        wait for 20 ns;
```

```
83      w_SA <= B"11000";
        wait for 20 ns;
85      w_SA <= B"11001";
        wait for 20 ns;
87      w_SA <= B"11010";
        wait for 20 ns;
89      w_SA <= B"11011";
        wait for 20 ns;
91      w_SA <= B"11100";
        wait for 20 ns;
93      w_SA <= B"11101";
        wait for 20 ns;
95      w_SA <= B"11110";
        wait for 20 ns;
97      w_SA <= B"11111";
        wait for 20 ns;
99      w_A <= B"1100_0000_0000_0000_0000_0000_0000_0000";
        w_SA <= B"00000";
101     wait for 20 ns;
        w_SA <= B"00001";
103     wait for 20 ns;
        w_SA <= B"00010";
105     wait for 20 ns;
        w_SA <= B"00011";
107     wait for 20 ns;
        w_SA <= B"00100";
109     wait for 20 ns;
        w_SA <= B"00101";
111     wait for 20 ns;
        w_SA <= B"00110";
113     wait for 20 ns;
        w_SA <= B"00111";
115     wait for 20 ns;
        w_SA <= B"01000";
117     wait for 20 ns;
        w_SA <= B"01001";
119     wait for 20 ns;
        w_SA <= B"01010";
121     wait for 20 ns;
        w_SA <= B"01011";
123     wait for 20 ns;
        w_SA <= B"01100";
125     wait for 20 ns;
```

```
    w_SA <= B"01101";
    wait for 20 ns;
    w_SA <= B"01110";
    wait for 20 ns;
    w_SA <= B"01111";
    wait for 20 ns;
    w_ARITH <= '1';
    w_SA <= B"10000";
    wait for 20 ns;
    w_SA <= B"10001";
    wait for 20 ns;
    w_SA <= B"10010";
    wait for 20 ns;
    w_SA <= B"10011";
    wait for 20 ns;
    w_SA <= B"10100";
    wait for 20 ns;
    w_SA <= B"10101";
    wait for 20 ns;
    w_SA <= B"10110";
    wait for 20 ns;
    w_SA <= B"10111";
    wait for 20 ns;
    w_SA <= B"11000";
    wait for 20 ns;
    w_SA <= B"11001";
    wait for 20 ns;
    w_SA <= B"11010";
    wait for 20 ns;
    w_SA <= B"11011";
    wait for 20 ns;
    w_SA <= B"11100";
    wait for 20 ns;
    w_SA <= B"11101";
    wait for 20 ns;
    w_SA <= B"11110";
    wait for 20 ns;
    w_SA <= B"11111";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.16. RIGHT_SHIFT_TB.vhd Code

2.7 Bitwise Operations

2.7.1 2-bit Bitwise Operations

```
--| Bitwise_Ops2_TB
--| Tests Bitwise_Ops2
library IEEE;
use IEEE.std_logic_1164.all;

entity Bitwise_Ops2_TB is
end Bitwise_Ops2_TB;

architecture testbench of Bitwise_Ops2_TB is
  component Bitwise_Ops2 is
    port (i_A : in  std_logic;
          i_B : in  std_logic;
          o_A_n : out std_logic;
          o_B_n : out std_logic;
          o_AND : out std_logic;
          o_NAND: out std_logic;
          o_OR  : out std_logic;
          o_XOR : out std_logic;
          o_XNOR  : out std_logic;
          o_NOR : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic := '0';
  signal w_A_n : std_logic := '0';
  signal w_B : std_logic := '0';
  signal w_B_n : std_logic := '0';
  signal w_AND : std_logic;
  signal w_NAND : std_logic;
  signal w_OR  : std_logic;
  signal w_XOR : std_logic;
  signal w_XNOR : std_logic;
  signal w_NOR : std_logic;
begin
  -- Connect Bitwise_Ops2
  u_Bitwise_Ops2: Bitwise_Ops2
  port map (i_A => w_A,
            i_B => w_B,
            o_A_n => w_A_n,
            o_B_n => w_B_n,
            o_AND => w_AND,
```

```
41            o_NAND => w_NAND,
              o_OR => w_OR,
43            o_XOR => w_XOR,
              o_XNOR => w_XNOR,
45            o_NOR => w_NOR);

47    -- Process for stimulating inputs
    stimulus: process is
49    begin
      wait for 20 ns;
51    w_A <= '1';
      wait for 20 ns;
53    w_B <= '1';
      wait for 20 ns;
55    w_A <= '0';
      wait for 20 ns;
57    wait;
    end process stimulus;
59  end testbench;
```

Listing B.17. Bitwise_Ops2_TB.vhd Code

2.7.2 2-bit Bitwise Operations

```
--| Bitwise_Ops2_32_TB
--| Tests Bitwise_Ops2_32
library IEEE;
use IEEE.std_logic_1164.all;

entity Bitwise_Ops2_32_TB is
end Bitwise_Ops2_32_TB;

architecture testbench of Bitwise_Ops2_32_TB is
  component Bitwise_Ops2_32 is
    port (i_A    : in  std_logic_vector(31 downto 0);
          i_B    : in  std_logic_vector(31 downto 0);
          o_A_n  : out std_logic_vector(31 downto 0);
          o_B_n  : out std_logic_vector(31 downto 0);
          o_AND  : out std_logic_vector(31 downto 0);
          o_NAND : out std_logic_vector(31 downto 0);
          o_OR   : out std_logic_vector(31 downto 0);
          o_XOR  : out std_logic_vector(31 downto 0);
          o_XNOR : out std_logic_vector(31 downto 0);
          o_NOR  : out std_logic_vector(31 downto 0));
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_A_n : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_B : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_B_n : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_AND : std_logic_vector(31 downto 0);
  signal w_NAND : std_logic_vector(31 downto 0);
  signal w_OR  : std_logic_vector(31 downto 0);
  signal w_XOR : std_logic_vector(31 downto 0);
  signal w_XNOR : std_logic_vector(31 downto 0);
  signal w_NOR : std_logic_vector(31 downto 0);
begin
  -- Connect Bitwise_Ops2_32
  u_Bitwise_Ops2_32: Bitwise_Ops2_32
  port map (i_A => w_A,
            i_B => w_B,
```

```
            o_A_n => w_A_n,
            o_B_n => w_B_n,
            o_AND => w_AND,
            o_NAND => w_NAND,
            o_OR => w_OR,
            o_XOR => w_XOR,
            o_XNOR => w_XNOR,
            o_NOR => w_NOR);

    -- Process for stimulating inputs
    stimulus: process is
    begin
      wait for 20 ns;
      w_A <= B"1111_1111_1111_1111_1111_1111_1111_1111";
      wait for 20 ns;
      w_B <= B"1111_1111_1111_1111_1111_1111_1111_1111";
      wait for 20 ns;
      w_A <= (others => '0');
      wait for 20 ns;
      w_A <= B"1010_1010_1010_1010_1010_1010_1010_1010";
      w_B <= B"0101_0101_0101_0101_0101_0101_0101_0101";
      wait for 20 ns;
      wait;
    end process stimulus;
end testbench;
```

Listing B.18. Bitwise_Ops2_32_TB.vhd Code

2.8 Adder

2.8.1 1-bit Full Adder

```
--| FULL_ADDER_1_TB
--| Tests FULL_ADDER_1
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_1_TB is
end FULL_ADDER_1_TB;

architecture testbench of FULL_ADDER_1_TB is
  component FULL_ADDER_1 is
    port (i_A      : in  std_logic;
          i_A_n    : in  std_logic;
          i_B      : in  std_logic;
          i_B_n    : in  std_logic;
          i_AB_AND : in  std_logic;
          i_AB_NAND : in  std_logic;
          i_AB_NOR : in  std_logic;
          i_C      : in  std_logic;
          o_S      : out std_logic;
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic := '0';
  signal w_A_n : std_logic;
  signal w_B : std_logic := '0';
  signal w_B_n : std_logic;
  signal w_AB_AND : std_logic;
  signal w_AB_NAND : std_logic;
  signal w_AB_NOR : std_logic;
  signal w_C : std_logic := '0';
  signal w_S : std_logic;
  signal w_Cout  : std_logic;
begin
  -- Connect input wires
  w_A_n <= NOT w_A;
  w_B_n <= NOT w_B;
  w_AB_AND <= w_A AND w_B;
  w_AB_NAND <= NOT w_AB_AND;
  w_AB_NOR <= NOT (w_A OR w_B);

```

```
  -- Connect FULL_ADDER_1
42 u_FULL_ADDER_1: FULL_ADDER_1
  port map (i_A => w_A,
44          i_A_n => w_A_n,
            i_B => w_B,
46          i_B_n => w_B_n,
            i_AB_AND => w_AB_AND,
48          i_AB_NAND => w_AB_NAND,
            i_AB_NOR => w_AB_NOR,
50          i_C => w_C,
            o_S => w_S,
52          o_C => w_Cout);

54 -- Process for stimulating inputs
  stimulus: process is
56 begin
    wait for 20 ns;
58   w_C <= '1';
    wait for 20 ns;
60   w_C <= '0';
    w_B <= '1';
62   wait for 20 ns;
    w_C <= '1';
64   wait for 20 ns;
    w_C <= '0';
66   w_B <= '0';
    w_A <= '1';
68   wait for 20 ns;
    w_C <= '1';
70   wait for 20 ns;
    w_C <= '0';
72   w_B <= '1';
    wait for 20 ns;
74   w_C <= '1';
    wait for 20 ns;
76   wait;
  end process stimulus;
78 end testbench;
```

Listing B.19. FULL_ADDER_1_TB.vhd Code

2.8.2 4-bit Full Adder

```
--| FULL_ADDER_4_TB
--| Tests FULL_ADDER_4
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_4_TB is
end FULL_ADDER_4_TB;

architecture testbench of FULL_ADDER_4_TB is
  component FULL_ADDER_4 is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_A_n    : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_B_n    : in  std_logic_vector(3 downto 0);
          i_AB_AND    : in  std_logic_vector(3 downto 0);
          i_AB_NAND : in  std_logic_vector(3 downto 0);
          i_AB_NOR    : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(3 downto 0) := (others =>'0')
   ;
  signal w_A_n : std_logic_vector(3 downto 0);
  signal w_B : std_logic_vector(3 downto 0) := (others =>'0')
   ;
  signal w_B_n : std_logic_vector(3 downto 0);
  signal w_AB_AND : std_logic_vector(3 downto 0);
  signal w_AB_NAND : std_logic_vector(3 downto 0);
  signal w_AB_NOR : std_logic_vector(3 downto 0);
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(3 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect input wires

  w_A_n <= NOT w_A;
  w_B_n <= NOT w_B;
  w_AB_AND <= w_A AND w_B;
  w_AB_NAND <= NOT w_AB_AND;
```

```
40    w_AB_NOR <= NOT (w_A OR w_B);
      -- Connect FULL_ADDER_4
42    u_FULL_ADDER_4: FULL_ADDER_4
      port map (i_A => w_A,
44              i_A_n => w_A_n,
                i_B => w_B,
46              i_B_n => w_B_n,
                i_AB_AND => w_AB_AND,
48              i_AB_NAND => w_AB_NAND,
                i_AB_NOR => w_AB_NOR,
50              i_C => w_C,
                o_S => w_S,
52              o_C => w_Cout);

54    -- Process for stimulating inputs
      stimulus: process is
56    begin
        wait for 20 ns;
58      w_A <= B"0001";
        wait for 20 ns;
60      w_A <= B"0010";
        wait for 20 ns;
62      w_A <= B"0011";
        wait for 20 ns;
64      w_A <= B"0100";
        wait for 20 ns;
66      w_A <= B"0101";
        wait for 20 ns;
68      w_A <= B"0110";
        wait for 20 ns;
70      w_A <= B"0111";
        wait for 20 ns;
72      w_A <= B"1000";
        wait for 20 ns;
74      w_A <= B"1001";
        wait for 20 ns;
76      w_A <= B"1010";
        wait for 20 ns;
78      w_A <= B"1011";
        wait for 20 ns;
80      w_A <= B"1100";
        wait for 20 ns;
82      w_A <= B"1101";
```

```
        wait for 20 ns;
84      w_A <= B"1110";
        wait for 20 ns;
86      w_A <= B"1111";
        wait for 20 ns;
88      w_B <= B"0001";
        w_A <= B"0000";
90      wait for 20 ns;
        w_A <= B"0001";
92      wait for 20 ns;
        w_A <= B"0010";
94      wait for 20 ns;
        w_A <= B"0011";
96      wait for 20 ns;
        w_A <= B"0100";
98      wait for 20 ns;
        w_A <= B"0101";
100     wait for 20 ns;
        w_A <= B"0110";
102     wait for 20 ns;
        w_A <= B"0111";
104     wait for 20 ns;
        w_A <= B"1000";
106     wait for 20 ns;
        w_A <= B"1001";
108     wait for 20 ns;
        w_A <= B"1010";
110     wait for 20 ns;
        w_A <= B"1011";
112     wait for 20 ns;
        w_A <= B"1100";
114     wait for 20 ns;
        w_A <= B"1101";
116     wait for 20 ns;
        w_A <= B"1110";
118     wait for 20 ns;
        w_A <= B"1111";
120     wait for 20 ns;
        w_B <= B"0010";
122     w_A <= B"0000";
        wait for 20 ns;
124     w_A <= B"0001";
        wait for 20 ns;
```

```
126         w_A <= B"0010";
            wait for 20 ns;
128         w_A <= B"0011";
            wait for 20 ns;
130         w_A <= B"0100";
            wait for 20 ns;
132         w_A <= B"0101";
            wait for 20 ns;
134         w_A <= B"0110";
            wait for 20 ns;
136         w_A <= B"0111";
            wait for 20 ns;
138         w_A <= B"1000";
            wait for 20 ns;
140         w_A <= B"1001";
            wait for 20 ns;
142         w_A <= B"1010";
            wait for 20 ns;
144         w_A <= B"1011";
            wait for 20 ns;
146         w_A <= B"1100";
            wait for 20 ns;
148         w_A <= B"1101";
            wait for 20 ns;
150         w_A <= B"1110";
            wait for 20 ns;
152         w_A <= B"1111";
            wait for 20 ns;
154         w_B <= B"0011";
            w_A <= B"0000";
156         wait for 20 ns;
            w_A <= B"0001";
158         wait for 20 ns;
            w_A <= B"0010";
160         wait for 20 ns;
            w_A <= B"0011";
162         wait for 20 ns;
            w_A <= B"0100";
164         wait for 20 ns;
            w_A <= B"0101";
166         wait for 20 ns;
            w_A <= B"0110";
168         wait for 20 ns;
```

```
        w_A <= B"0111";
170     wait for 20 ns;
        w_A <= B"1000";
172     wait for 20 ns;
        w_A <= B"1001";
174     wait for 20 ns;
        w_A <= B"1010";
176     wait for 20 ns;
        w_A <= B"1011";
178     wait for 20 ns;
        w_A <= B"1100";
180     wait for 20 ns;
        w_A <= B"1101";
182     wait for 20 ns;
        w_A <= B"1110";
184     wait for 20 ns;
        w_A <= B"1111";
186     wait for 20 ns;
        w_B <= B"0100";
188     w_A <= B"0000";
        wait for 20 ns;
190     w_A <= B"0001";
        wait for 20 ns;
192     w_A <= B"0010";
        wait for 20 ns;
194     w_A <= B"0011";
        wait for 20 ns;
196     w_A <= B"0100";
        wait for 20 ns;
198     w_A <= B"0101";
        wait for 20 ns;
200     w_A <= B"0110";
        wait for 20 ns;
202     w_A <= B"0111";
        wait for 20 ns;
204     w_A <= B"1000";
        wait for 20 ns;
206     w_A <= B"1001";
        wait for 20 ns;
208     w_A <= B"1010";
        wait for 20 ns;
210     w_A <= B"1011";
        wait for 20 ns;
```

```
212     w_A <= B"1100";
        wait for 20 ns;
214     w_A <= B"1101";
        wait for 20 ns;
216     w_A <= B"1110";
        wait for 20 ns;
218     w_A <= B"1111";
        wait for 20 ns;
220     w_B <= B"0101";
        w_A <= B"0000";
222     wait for 20 ns;
        w_A <= B"0001";
224     wait for 20 ns;
        w_A <= B"0010";
226     wait for 20 ns;
        w_A <= B"0011";
228     wait for 20 ns;
        w_A <= B"0100";
230     wait for 20 ns;
        w_A <= B"0101";
232     wait for 20 ns;
        w_A <= B"0110";
234     wait for 20 ns;
        w_A <= B"0111";
236     wait for 20 ns;
        w_A <= B"1000";
238     wait for 20 ns;
        w_A <= B"1001";
240     wait for 20 ns;
        w_A <= B"1010";
242     wait for 20 ns;
        w_A <= B"1011";
244     wait for 20 ns;
        w_A <= B"1100";
246     wait for 20 ns;
        w_A <= B"1101";
248     wait for 20 ns;
        w_A <= B"1110";
250     wait for 20 ns;
        w_A <= B"1111";
252     wait for 20 ns;
        w_B <= B"0110";
254     w_A <= B"0000";
```

```
        wait for 20 ns;
256     w_A <= B"0001";
        wait for 20 ns;
258     w_A <= B"0010";
        wait for 20 ns;
260     w_A <= B"0011";
        wait for 20 ns;
262     w_A <= B"0100";
        wait for 20 ns;
264     w_A <= B"0101";
        wait for 20 ns;
266     w_A <= B"0110";
        wait for 20 ns;
268     w_A <= B"0111";
        wait for 20 ns;
270     w_A <= B"1000";
        wait for 20 ns;
272     w_A <= B"1001";
        wait for 20 ns;
274     w_A <= B"1010";
        wait for 20 ns;
276     w_A <= B"1011";
        wait for 20 ns;
278     w_A <= B"1100";
        wait for 20 ns;
280     w_A <= B"1101";
        wait for 20 ns;
282     w_A <= B"1110";
        wait for 20 ns;
284     w_A <= B"1111";
        wait for 20 ns;
286     w_B <= B"0111";
        w_A <= B"0000";
288     wait for 20 ns;
        w_A <= B"0001";
290     wait for 20 ns;
        w_A <= B"0010";
292     wait for 20 ns;
        w_A <= B"0011";
294     wait for 20 ns;
        w_A <= B"0100";
296     wait for 20 ns;
        w_A <= B"0101";
```

```
298         wait for 20 ns;
            w_A <= B"0110";
300         wait for 20 ns;
            w_A <= B"0111";
302         wait for 20 ns;
            w_A <= B"1000";
304         wait for 20 ns;
            w_A <= B"1001";
306         wait for 20 ns;
            w_A <= B"1010";
308         wait for 20 ns;
            w_A <= B"1011";
310         wait for 20 ns;
            w_A <= B"1100";
312         wait for 20 ns;
            w_A <= B"1101";
314         wait for 20 ns;
            w_A <= B"1110";
316         wait for 20 ns;
            w_A <= B"1111";
318         wait for 20 ns;
            w_B <= B"1000";
320         w_A <= B"0000";
            wait for 20 ns;
322         w_A <= B"0001";
            wait for 20 ns;
324         w_A <= B"0010";
            wait for 20 ns;
326         w_A <= B"0011";
            wait for 20 ns;
328         w_A <= B"0100";
            wait for 20 ns;
330         w_A <= B"0101";
            wait for 20 ns;
332         w_A <= B"0110";
            wait for 20 ns;
334         w_A <= B"0111";
            wait for 20 ns;
336         w_A <= B"1000";
            wait for 20 ns;
338         w_A <= B"1001";
            wait for 20 ns;
340         w_A <= B"1010";
```

```
        wait for 20 ns;
342     w_A <= B"1011";
        wait for 20 ns;
344     w_A <= B"1100";
        wait for 20 ns;
346     w_A <= B"1101";
        wait for 20 ns;
348     w_A <= B"1110";
        wait for 20 ns;
350     w_A <= B"1111";
        wait for 20 ns;
352     w_B <= B"1001";
        w_A <= B"0000";
354     wait for 20 ns;
        w_A <= B"0001";
356     wait for 20 ns;
        w_A <= B"0010";
358     wait for 20 ns;
        w_A <= B"0011";
360     wait for 20 ns;
        w_A <= B"0100";
362     wait for 20 ns;
        w_A <= B"0101";
364     wait for 20 ns;
        w_A <= B"0110";
366     wait for 20 ns;
        w_A <= B"0111";
368     wait for 20 ns;
        w_A <= B"1000";
370     wait for 20 ns;
        w_A <= B"1001";
372     wait for 20 ns;
        w_A <= B"1010";
374     wait for 20 ns;
        w_A <= B"1011";
376     wait for 20 ns;
        w_A <= B"1100";
378     wait for 20 ns;
        w_A <= B"1101";
380     wait for 20 ns;
        w_A <= B"1110";
382     wait for 20 ns;
        w_A <= B"1111";
```

```
384         wait for 20 ns;
            w_B <= B"1010";
386         w_A <= B"0000";
            wait for 20 ns;
388         w_A <= B"0001";
            wait for 20 ns;
390         w_A <= B"0010";
            wait for 20 ns;
392         w_A <= B"0011";
            wait for 20 ns;
394         w_A <= B"0100";
            wait for 20 ns;
396         w_A <= B"0101";
            wait for 20 ns;
398         w_A <= B"0110";
            wait for 20 ns;
400         w_A <= B"0111";
            wait for 20 ns;
402         w_A <= B"1000";
            wait for 20 ns;
404         w_A <= B"1001";
            wait for 20 ns;
406         w_A <= B"1010";
            wait for 20 ns;
408         w_A <= B"1011";
            wait for 20 ns;
410         w_A <= B"1100";
            wait for 20 ns;
412         w_A <= B"1101";
            wait for 20 ns;
414         w_A <= B"1110";
            wait for 20 ns;
416         w_A <= B"1111";
            wait for 20 ns;
418         w_B <= B"1011";
            w_A <= B"0000";
420         wait for 20 ns;
            w_A <= B"0001";
422         wait for 20 ns;
            w_A <= B"0010";
424         wait for 20 ns;
            w_A <= B"0011";
426         wait for 20 ns;
```

```
        w_A <= B"0100";
428     wait for 20 ns;
        w_A <= B"0101";
430     wait for 20 ns;
        w_A <= B"0110";
432     wait for 20 ns;
        w_A <= B"0111";
434     wait for 20 ns;
        w_A <= B"1000";
436     wait for 20 ns;
        w_A <= B"1001";
438     wait for 20 ns;
        w_A <= B"1010";
440     wait for 20 ns;
        w_A <= B"1011";
442     wait for 20 ns;
        w_A <= B"1100";
444     wait for 20 ns;
        w_A <= B"1101";
446     wait for 20 ns;
        w_A <= B"1110";
448     wait for 20 ns;
        w_A <= B"1111";
450     wait for 20 ns;
        w_B <= B"1100";
452     w_A <= B"0000";
        wait for 20 ns;
454     w_A <= B"0001";
        wait for 20 ns;
456     w_A <= B"0010";
        wait for 20 ns;
458     w_A <= B"0011";
        wait for 20 ns;
460     w_A <= B"0100";
        wait for 20 ns;
462     w_A <= B"0101";
        wait for 20 ns;
464     w_A <= B"0110";
        wait for 20 ns;
466     w_A <= B"0111";
        wait for 20 ns;
468     w_A <= B"1000";
        wait for 20 ns;
```

```
470         w_A <= B"1001";
            wait for 20 ns;
472         w_A <= B"1010";
            wait for 20 ns;
474         w_A <= B"1011";
            wait for 20 ns;
476         w_A <= B"1100";
            wait for 20 ns;
478         w_A <= B"1101";
            wait for 20 ns;
480         w_A <= B"1110";
            wait for 20 ns;
482         w_A <= B"1111";
            wait for 20 ns;
484         w_B <= B"1101";
            w_A <= B"0000";
486         wait for 20 ns;
            w_A <= B"0001";
488         wait for 20 ns;
            w_A <= B"0010";
490         wait for 20 ns;
            w_A <= B"0011";
492         wait for 20 ns;
            w_A <= B"0100";
494         wait for 20 ns;
            w_A <= B"0101";
496         wait for 20 ns;
            w_A <= B"0110";
498         wait for 20 ns;
            w_A <= B"0111";
500         wait for 20 ns;
            w_A <= B"1000";
502         wait for 20 ns;
            w_A <= B"1001";
504         wait for 20 ns;
            w_A <= B"1010";
506         wait for 20 ns;
            w_A <= B"1011";
508         wait for 20 ns;
            w_A <= B"1100";
510         wait for 20 ns;
            w_A <= B"1101";
512         wait for 20 ns;
```

```
        w_A <= B"1110";
514     wait for 20 ns;
        w_A <= B"1111";
516     wait for 20 ns;
        w_B <= B"1110";
518     w_A <= B"0000";
        wait for 20 ns;
520     w_A <= B"0001";
        wait for 20 ns;
522     w_A <= B"0010";
        wait for 20 ns;
524     w_A <= B"0011";
        wait for 20 ns;
526     w_A <= B"0100";
        wait for 20 ns;
528     w_A <= B"0101";
        wait for 20 ns;
530     w_A <= B"0110";
        wait for 20 ns;
532     w_A <= B"0111";
        wait for 20 ns;
534     w_A <= B"1000";
        wait for 20 ns;
536     w_A <= B"1001";
        wait for 20 ns;
538     w_A <= B"1010";
        wait for 20 ns;
540     w_A <= B"1011";
        wait for 20 ns;
542     w_A <= B"1100";
        wait for 20 ns;
544     w_A <= B"1101";
        wait for 20 ns;
546     w_A <= B"1110";
        wait for 20 ns;
548     w_A <= B"1111";
        wait for 20 ns;
550     w_B <= B"1111";
        w_A <= B"0000";
552     wait for 20 ns;
        w_A <= B"0001";
554     wait for 20 ns;
        w_A <= B"0010";
```

```
556         wait for 20 ns;
            w_A <= B"0011";
558         wait for 20 ns;
            w_A <= B"0100";
560         wait for 20 ns;
            w_A <= B"0101";
562         wait for 20 ns;
            w_A <= B"0110";
564         wait for 20 ns;
            w_A <= B"0111";
566         wait for 20 ns;
            w_A <= B"1000";
568         wait for 20 ns;
            w_A <= B"1001";
570         wait for 20 ns;
            w_A <= B"1010";
572         wait for 20 ns;
            w_A <= B"1011";
574         wait for 20 ns;
            w_A <= B"1100";
576         wait for 20 ns;
            w_A <= B"1101";
578         wait for 20 ns;
            w_A <= B"1110";
580         wait for 20 ns;
            w_A <= B"1111";
582         wait for 20 ns;
            w_c <= '1';
584         w_B <= B"0000";
            w_A <= B"0000";
586         wait for 20 ns;
            w_A <= B"0001";
588         wait for 20 ns;
            w_A <= B"0010";
590         wait for 20 ns;
            w_A <= B"0011";
592         wait for 20 ns;
            w_A <= B"0100";
594         wait for 20 ns;
            w_A <= B"0101";
596         wait for 20 ns;
            w_A <= B"0110";
598         wait for 20 ns;
```

```
        w_A <= B"0111";
600     wait for 20 ns;
        w_A <= B"1000";
602     wait for 20 ns;
        w_A <= B"1001";
604     wait for 20 ns;
        w_A <= B"1010";
606     wait for 20 ns;
        w_A <= B"1011";
608     wait for 20 ns;
        w_A <= B"1100";
610     wait for 20 ns;
        w_A <= B"1101";
612     wait for 20 ns;
        w_A <= B"1110";
614     wait for 20 ns;
        w_A <= B"1111";
616     wait for 20 ns;
        w_B <= B"0001";
618     w_A <= B"0000";
        wait for 20 ns;
620     w_A <= B"0001";
        wait for 20 ns;
622     w_A <= B"0010";
        wait for 20 ns;
624     w_A <= B"0011";
        wait for 20 ns;
626     w_A <= B"0100";
        wait for 20 ns;
628     w_A <= B"0101";
        wait for 20 ns;
630     w_A <= B"0110";
        wait for 20 ns;
632     w_A <= B"0111";
        wait for 20 ns;
634     w_A <= B"1000";
        wait for 20 ns;
636     w_A <= B"1001";
        wait for 20 ns;
638     w_A <= B"1010";
        wait for 20 ns;
640     w_A <= B"1011";
        wait for 20 ns;
```

```
642         w_A <= B"1100";
            wait for 20 ns;
644         w_A <= B"1101";
            wait for 20 ns;
646         w_A <= B"1110";
            wait for 20 ns;
648         w_A <= B"1111";
            wait for 20 ns;
650         w_B <= B"0010";
            w_A <= B"0000";
652         wait for 20 ns;
            w_A <= B"0001";
654         wait for 20 ns;
            w_A <= B"0010";
656         wait for 20 ns;
            w_A <= B"0011";
658         wait for 20 ns;
            w_A <= B"0100";
660         wait for 20 ns;
            w_A <= B"0101";
662         wait for 20 ns;
            w_A <= B"0110";
664         wait for 20 ns;
            w_A <= B"0111";
666         wait for 20 ns;
            w_A <= B"1000";
668         wait for 20 ns;
            w_A <= B"1001";
670         wait for 20 ns;
            w_A <= B"1010";
672         wait for 20 ns;
            w_A <= B"1011";
674         wait for 20 ns;
            w_A <= B"1100";
676         wait for 20 ns;
            w_A <= B"1101";
678         wait for 20 ns;
            w_A <= B"1110";
680         wait for 20 ns;
            w_A <= B"1111";
682         wait for 20 ns;
            w_B <= B"0011";
684         w_A <= B"0000";
```

```
        wait for 20 ns;
686     w_A <= B"0001";
        wait for 20 ns;
688     w_A <= B"0010";
        wait for 20 ns;
690     w_A <= B"0011";
        wait for 20 ns;
692     w_A <= B"0100";
        wait for 20 ns;
694     w_A <= B"0101";
        wait for 20 ns;
696     w_A <= B"0110";
        wait for 20 ns;
698     w_A <= B"0111";
        wait for 20 ns;
700     w_A <= B"1000";
        wait for 20 ns;
702     w_A <= B"1001";
        wait for 20 ns;
704     w_A <= B"1010";
        wait for 20 ns;
706     w_A <= B"1011";
        wait for 20 ns;
708     w_A <= B"1100";
        wait for 20 ns;
710     w_A <= B"1101";
        wait for 20 ns;
712     w_A <= B"1110";
        wait for 20 ns;
714     w_A <= B"1111";
        wait for 20 ns;
716     w_B <= B"0100";
        w_A <= B"0000";
718     wait for 20 ns;
        w_A <= B"0001";
720     wait for 20 ns;
        w_A <= B"0010";
722     wait for 20 ns;
        w_A <= B"0011";
724     wait for 20 ns;
        w_A <= B"0100";
726     wait for 20 ns;
        w_A <= B"0101";
```

```
728         wait for 20 ns;
            w_A <= B"0110";
730         wait for 20 ns;
            w_A <= B"0111";
732         wait for 20 ns;
            w_A <= B"1000";
734         wait for 20 ns;
            w_A <= B"1001";
736         wait for 20 ns;
            w_A <= B"1010";
738         wait for 20 ns;
            w_A <= B"1011";
740         wait for 20 ns;
            w_A <= B"1100";
742         wait for 20 ns;
            w_A <= B"1101";
744         wait for 20 ns;
            w_A <= B"1110";
746         wait for 20 ns;
            w_A <= B"1111";
748         wait for 20 ns;
            w_B <= B"0101";
750         w_A <= B"0000";
            wait for 20 ns;
752         w_A <= B"0001";
            wait for 20 ns;
754         w_A <= B"0010";
            wait for 20 ns;
756         w_A <= B"0011";
            wait for 20 ns;
758         w_A <= B"0100";
            wait for 20 ns;
760         w_A <= B"0101";
            wait for 20 ns;
762         w_A <= B"0110";
            wait for 20 ns;
764         w_A <= B"0111";
            wait for 20 ns;
766         w_A <= B"1000";
            wait for 20 ns;
768         w_A <= B"1001";
            wait for 20 ns;
770         w_A <= B"1010";
```

```
        wait for 20 ns;
772     w_A <= B"1011";
        wait for 20 ns;
774     w_A <= B"1100";
        wait for 20 ns;
776     w_A <= B"1101";
        wait for 20 ns;
778     w_A <= B"1110";
        wait for 20 ns;
780     w_A <= B"1111";
        wait for 20 ns;
782     w_B <= B"0110";
        w_A <= B"0000";
784     wait for 20 ns;
        w_A <= B"0001";
786     wait for 20 ns;
        w_A <= B"0010";
788     wait for 20 ns;
        w_A <= B"0011";
790     wait for 20 ns;
        w_A <= B"0100";
792     wait for 20 ns;
        w_A <= B"0101";
794     wait for 20 ns;
        w_A <= B"0110";
796     wait for 20 ns;
        w_A <= B"0111";
798     wait for 20 ns;
        w_A <= B"1000";
800     wait for 20 ns;
        w_A <= B"1001";
802     wait for 20 ns;
        w_A <= B"1010";
804     wait for 20 ns;
        w_A <= B"1011";
806     wait for 20 ns;
        w_A <= B"1100";
808     wait for 20 ns;
        w_A <= B"1101";
810     wait for 20 ns;
        w_A <= B"1110";
812     wait for 20 ns;
        w_A <= B"1111";
```

```
814     wait for 20 ns;
        w_B <= B"0111";
816     w_A <= B"0000";
        wait for 20 ns;
818     w_A <= B"0001";
        wait for 20 ns;
820     w_A <= B"0010";
        wait for 20 ns;
822     w_A <= B"0011";
        wait for 20 ns;
824     w_A <= B"0100";
        wait for 20 ns;
826     w_A <= B"0101";
        wait for 20 ns;
828     w_A <= B"0110";
        wait for 20 ns;
830     w_A <= B"0111";
        wait for 20 ns;
832     w_A <= B"1000";
        wait for 20 ns;
834     w_A <= B"1001";
        wait for 20 ns;
836     w_A <= B"1010";
        wait for 20 ns;
838     w_A <= B"1011";
        wait for 20 ns;
840     w_A <= B"1100";
        wait for 20 ns;
842     w_A <= B"1101";
        wait for 20 ns;
844     w_A <= B"1110";
        wait for 20 ns;
846     w_A <= B"1111";
        wait for 20 ns;
848     w_B <= B"1000";
        w_A <= B"0000";
850     wait for 20 ns;
        w_A <= B"0001";
852     wait for 20 ns;
        w_A <= B"0010";
854     wait for 20 ns;
        w_A <= B"0011";
856     wait for 20 ns;
```

```
        w_A <= B"0100";
858     wait for 20 ns;
        w_A <= B"0101";
860     wait for 20 ns;
        w_A <= B"0110";
862     wait for 20 ns;
        w_A <= B"0111";
864     wait for 20 ns;
        w_A <= B"1000";
866     wait for 20 ns;
        w_A <= B"1001";
868     wait for 20 ns;
        w_A <= B"1010";
870     wait for 20 ns;
        w_A <= B"1011";
872     wait for 20 ns;
        w_A <= B"1100";
874     wait for 20 ns;
        w_A <= B"1101";
876     wait for 20 ns;
        w_A <= B"1110";
878     wait for 20 ns;
        w_A <= B"1111";
880     wait for 20 ns;
        w_B <= B"1001";
882     w_A <= B"0000";
        wait for 20 ns;
884     w_A <= B"0001";
        wait for 20 ns;
886     w_A <= B"0010";
        wait for 20 ns;
888     w_A <= B"0011";
        wait for 20 ns;
890     w_A <= B"0100";
        wait for 20 ns;
892     w_A <= B"0101";
        wait for 20 ns;
894     w_A <= B"0110";
        wait for 20 ns;
896     w_A <= B"0111";
        wait for 20 ns;
898     w_A <= B"1000";
        wait for 20 ns;
```

```
900         w_A <= B"1001";
            wait for 20 ns;
902         w_A <= B"1010";
            wait for 20 ns;
904         w_A <= B"1011";
            wait for 20 ns;
906         w_A <= B"1100";
            wait for 20 ns;
908         w_A <= B"1101";
            wait for 20 ns;
910         w_A <= B"1110";
            wait for 20 ns;
912         w_A <= B"1111";
            wait for 20 ns;
914         w_B <= B"1010";
            w_A <= B"0000";
916         wait for 20 ns;
            w_A <= B"0001";
918         wait for 20 ns;
            w_A <= B"0010";
920         wait for 20 ns;
            w_A <= B"0011";
922         wait for 20 ns;
            w_A <= B"0100";
924         wait for 20 ns;
            w_A <= B"0101";
926         wait for 20 ns;
            w_A <= B"0110";
928         wait for 20 ns;
            w_A <= B"0111";
930         wait for 20 ns;
            w_A <= B"1000";
932         wait for 20 ns;
            w_A <= B"1001";
934         wait for 20 ns;
            w_A <= B"1010";
936         wait for 20 ns;
            w_A <= B"1011";
938         wait for 20 ns;
            w_A <= B"1100";
940         wait for 20 ns;
            w_A <= B"1101";
942         wait for 20 ns;
```

```
        w_A <= B"1110";
944     wait for 20 ns;
        w_A <= B"1111";
946     wait for 20 ns;
        w_B <= B"1011";
948     w_A <= B"0000";
        wait for 20 ns;
950     w_A <= B"0001";
        wait for 20 ns;
952     w_A <= B"0010";
        wait for 20 ns;
954     w_A <= B"0011";
        wait for 20 ns;
956     w_A <= B"0100";
        wait for 20 ns;
958     w_A <= B"0101";
        wait for 20 ns;
960     w_A <= B"0110";
        wait for 20 ns;
962     w_A <= B"0111";
        wait for 20 ns;
964     w_A <= B"1000";
        wait for 20 ns;
966     w_A <= B"1001";
        wait for 20 ns;
968     w_A <= B"1010";
        wait for 20 ns;
970     w_A <= B"1011";
        wait for 20 ns;
972     w_A <= B"1100";
        wait for 20 ns;
974     w_A <= B"1101";
        wait for 20 ns;
976     w_A <= B"1110";
        wait for 20 ns;
978     w_A <= B"1111";
        wait for 20 ns;
980     w_B <= B"1100";
        w_A <= B"0000";
982     wait for 20 ns;
        w_A <= B"0001";
984     wait for 20 ns;
        w_A <= B"0010";
```

```
986         wait for 20 ns;
            w_A <= B"0011";
988         wait for 20 ns;
            w_A <= B"0100";
990         wait for 20 ns;
            w_A <= B"0101";
992         wait for 20 ns;
            w_A <= B"0110";
994         wait for 20 ns;
            w_A <= B"0111";
996         wait for 20 ns;
            w_A <= B"1000";
998         wait for 20 ns;
            w_A <= B"1001";
1000        wait for 20 ns;
            w_A <= B"1010";
1002        wait for 20 ns;
            w_A <= B"1011";
1004        wait for 20 ns;
            w_A <= B"1100";
1006        wait for 20 ns;
            w_A <= B"1101";
1008        wait for 20 ns;
            w_A <= B"1110";
1010        wait for 20 ns;
            w_A <= B"1111";
1012        wait for 20 ns;
            w_B <= B"1101";
1014        w_A <= B"0000";
            wait for 20 ns;
1016        w_A <= B"0001";
            wait for 20 ns;
1018        w_A <= B"0010";
            wait for 20 ns;
1020        w_A <= B"0011";
            wait for 20 ns;
1022        w_A <= B"0100";
            wait for 20 ns;
1024        w_A <= B"0101";
            wait for 20 ns;
1026        w_A <= B"0110";
            wait for 20 ns;
1028        w_A <= B"0111";
```

```
        wait for 20 ns;
1030    w_A <= B"1000";
        wait for 20 ns;
1032    w_A <= B"1001";
        wait for 20 ns;
1034    w_A <= B"1010";
        wait for 20 ns;
1036    w_A <= B"1011";
        wait for 20 ns;
1038    w_A <= B"1100";
        wait for 20 ns;
1040    w_A <= B"1101";
        wait for 20 ns;
1042    w_A <= B"1110";
        wait for 20 ns;
1044    w_A <= B"1111";
        wait for 20 ns;
1046    w_B <= B"1110";
        w_A <= B"0000";
1048    wait for 20 ns;
        w_A <= B"0001";
1050    wait for 20 ns;
        w_A <= B"0010";
1052    wait for 20 ns;
        w_A <= B"0011";
1054    wait for 20 ns;
        w_A <= B"0100";
1056    wait for 20 ns;
        w_A <= B"0101";
1058    wait for 20 ns;
        w_A <= B"0110";
1060    wait for 20 ns;
        w_A <= B"0111";
1062    wait for 20 ns;
        w_A <= B"1000";
1064    wait for 20 ns;
        w_A <= B"1001";
1066    wait for 20 ns;
        w_A <= B"1010";
1068    wait for 20 ns;
        w_A <= B"1011";
1070    wait for 20 ns;
        w_A <= B"1100";
```

```
1072        wait for 20 ns;
            w_A <= B"1101";
1074        wait for 20 ns;
            w_A <= B"1110";
1076        wait for 20 ns;
            w_A <= B"1111";
1078        wait for 20 ns;
            w_B <= B"1111";
1080        w_A <= B"0000";
            wait for 20 ns;
1082        w_A <= B"0001";
            wait for 20 ns;
1084        w_A <= B"0010";
            wait for 20 ns;
1086        w_A <= B"0011";
            wait for 20 ns;
1088        w_A <= B"0100";
            wait for 20 ns;
1090        w_A <= B"0101";
            wait for 20 ns;
1092        w_A <= B"0110";
            wait for 20 ns;
1094        w_A <= B"0111";
            wait for 20 ns;
1096        w_A <= B"1000";
            wait for 20 ns;
1098        w_A <= B"1001";
            wait for 20 ns;
1100        w_A <= B"1010";
            wait for 20 ns;
1102        w_A <= B"1011";
            wait for 20 ns;
1104        w_A <= B"1100";
            wait for 20 ns;
1106        w_A <= B"1101";
            wait for 20 ns;
1108        w_A <= B"1110";
            wait for 20 ns;
1110        w_A <= B"1111";
            wait for 20 ns;
1112        wait;
        end process stimulus;
1114    end testbench;
```

Listing B.20. FULL_ADDER_4_TB.vhd Code

2.8.3 4-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_4_TB
--| Tests CARRY_SELECT_ADDER_4
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_4_TB is
end CARRY_SELECT_ADDER_4_TB;

architecture testbench of CARRY_SELECT_ADDER_4_TB is
  component CARRY_SELECT_ADDER_4 is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_A_n    : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_B_n    : in  std_logic_vector(3 downto 0);
          i_AB_AND    : in  std_logic_vector(3 downto 0);
          i_AB_NAND : in  std_logic_vector(3 downto 0);
          i_AB_NOR    : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(3 downto 0) := (others =>'0')
   ;
  signal w_A_n : std_logic_vector(3 downto 0);
  signal w_B : std_logic_vector(3 downto 0) := (others =>'0')
   ;
  signal w_B_n : std_logic_vector(3 downto 0);
  signal w_AB_AND : std_logic_vector(3 downto 0);
  signal w_AB_NAND : std_logic_vector(3 downto 0);
  signal w_AB_NOR : std_logic_vector(3 downto 0);
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(3 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect input wires

  w_A_n <= NOT w_A;
  w_B_n <= NOT w_B;
  w_AB_AND <= w_A AND w_B;
  w_AB_NAND <= NOT w_AB_AND;
```

```
40    w_AB_NOR <= NOT (w_A OR w_B);
      -- Connect CARRY_SELECT_ADDER_4
42    u_CARRY_SELECT_ADDER_4: CARRY_SELECT_ADDER_4
      port map (i_A => w_A,
44              i_A_n => w_A_n,
                i_B => w_B,
46              i_B_n => w_B_n,
                i_AB_AND => w_AB_AND,
48              i_AB_NAND => w_AB_NAND,
                i_AB_NOR => w_AB_NOR,
50              i_C => w_C,
                o_S => w_S,
52              o_C => w_Cout);

54    -- Process for stimulating inputs
      stimulus: process is
56    begin
        wait for 20 ns;
58      w_A <= B"0001";
        wait for 20 ns;
60      w_A <= B"0010";
        wait for 20 ns;
62      w_A <= B"0011";
        wait for 20 ns;
64      w_A <= B"0100";
        wait for 20 ns;
66      w_A <= B"0101";
        wait for 20 ns;
68      w_A <= B"0110";
        wait for 20 ns;
70      w_A <= B"0111";
        wait for 20 ns;
72      w_A <= B"1000";
        wait for 20 ns;
74      w_A <= B"1001";
        wait for 20 ns;
76      w_A <= B"1010";
        wait for 20 ns;
78      w_A <= B"1011";
        wait for 20 ns;
80      w_A <= B"1100";
        wait for 20 ns;
82      w_A <= B"1101";
```

```
        wait for 20 ns;
84      w_A <= B"1110";
        wait for 20 ns;
86      w_A <= B"1111";
        wait for 20 ns;
88      w_B <= B"0001";
        w_A <= B"0000";
90      wait for 20 ns;
        w_A <= B"0001";
92      wait for 20 ns;
        w_A <= B"0010";
94      wait for 20 ns;
        w_A <= B"0011";
96      wait for 20 ns;
        w_A <= B"0100";
98      wait for 20 ns;
        w_A <= B"0101";
100     wait for 20 ns;
        w_A <= B"0110";
102     wait for 20 ns;
        w_A <= B"0111";
104     wait for 20 ns;
        w_A <= B"1000";
106     wait for 20 ns;
        w_A <= B"1001";
108     wait for 20 ns;
        w_A <= B"1010";
110     wait for 20 ns;
        w_A <= B"1011";
112     wait for 20 ns;
        w_A <= B"1100";
114     wait for 20 ns;
        w_A <= B"1101";
116     wait for 20 ns;
        w_A <= B"1110";
118     wait for 20 ns;
        w_A <= B"1111";
120     wait for 20 ns;
        w_B <= B"0010";
122     w_A <= B"0000";
        wait for 20 ns;
124     w_A <= B"0001";
        wait for 20 ns;
```

```
126         w_A <= B"0010";
            wait for 20 ns;
128         w_A <= B"0011";
            wait for 20 ns;
130         w_A <= B"0100";
            wait for 20 ns;
132         w_A <= B"0101";
            wait for 20 ns;
134         w_A <= B"0110";
            wait for 20 ns;
136         w_A <= B"0111";
            wait for 20 ns;
138         w_A <= B"1000";
            wait for 20 ns;
140         w_A <= B"1001";
            wait for 20 ns;
142         w_A <= B"1010";
            wait for 20 ns;
144         w_A <= B"1011";
            wait for 20 ns;
146         w_A <= B"1100";
            wait for 20 ns;
148         w_A <= B"1101";
            wait for 20 ns;
150         w_A <= B"1110";
            wait for 20 ns;
152         w_A <= B"1111";
            wait for 20 ns;
154         w_B <= B"0011";
            w_A <= B"0000";
156         wait for 20 ns;
            w_A <= B"0001";
158         wait for 20 ns;
            w_A <= B"0010";
160         wait for 20 ns;
            w_A <= B"0011";
162         wait for 20 ns;
            w_A <= B"0100";
164         wait for 20 ns;
            w_A <= B"0101";
166         wait for 20 ns;
            w_A <= B"0110";
168         wait for 20 ns;
```

```
        w_A <= B"0111";
170     wait for 20 ns;
        w_A <= B"1000";
172     wait for 20 ns;
        w_A <= B"1001";
174     wait for 20 ns;
        w_A <= B"1010";
176     wait for 20 ns;
        w_A <= B"1011";
178     wait for 20 ns;
        w_A <= B"1100";
180     wait for 20 ns;
        w_A <= B"1101";
182     wait for 20 ns;
        w_A <= B"1110";
184     wait for 20 ns;
        w_A <= B"1111";
186     wait for 20 ns;
        w_B <= B"0100";
188     w_A <= B"0000";
        wait for 20 ns;
190     w_A <= B"0001";
        wait for 20 ns;
192     w_A <= B"0010";
        wait for 20 ns;
194     w_A <= B"0011";
        wait for 20 ns;
196     w_A <= B"0100";
        wait for 20 ns;
198     w_A <= B"0101";
        wait for 20 ns;
200     w_A <= B"0110";
        wait for 20 ns;
202     w_A <= B"0111";
        wait for 20 ns;
204     w_A <= B"1000";
        wait for 20 ns;
206     w_A <= B"1001";
        wait for 20 ns;
208     w_A <= B"1010";
        wait for 20 ns;
210     w_A <= B"1011";
        wait for 20 ns;
```

```
212     w_A <= B"1100";
        wait for 20 ns;
214     w_A <= B"1101";
        wait for 20 ns;
216     w_A <= B"1110";
        wait for 20 ns;
218     w_A <= B"1111";
        wait for 20 ns;
220     w_B <= B"0101";
        w_A <= B"0000";
222     wait for 20 ns;
        w_A <= B"0001";
224     wait for 20 ns;
        w_A <= B"0010";
226     wait for 20 ns;
        w_A <= B"0011";
228     wait for 20 ns;
        w_A <= B"0100";
230     wait for 20 ns;
        w_A <= B"0101";
232     wait for 20 ns;
        w_A <= B"0110";
234     wait for 20 ns;
        w_A <= B"0111";
236     wait for 20 ns;
        w_A <= B"1000";
238     wait for 20 ns;
        w_A <= B"1001";
240     wait for 20 ns;
        w_A <= B"1010";
242     wait for 20 ns;
        w_A <= B"1011";
244     wait for 20 ns;
        w_A <= B"1100";
246     wait for 20 ns;
        w_A <= B"1101";
248     wait for 20 ns;
        w_A <= B"1110";
250     wait for 20 ns;
        w_A <= B"1111";
252     wait for 20 ns;
        w_B <= B"0110";
254     w_A <= B"0000";
```

```
        wait for 20 ns;
256     w_A <= B"0001";
        wait for 20 ns;
258     w_A <= B"0010";
        wait for 20 ns;
260     w_A <= B"0011";
        wait for 20 ns;
262     w_A <= B"0100";
        wait for 20 ns;
264     w_A <= B"0101";
        wait for 20 ns;
266     w_A <= B"0110";
        wait for 20 ns;
268     w_A <= B"0111";
        wait for 20 ns;
270     w_A <= B"1000";
        wait for 20 ns;
272     w_A <= B"1001";
        wait for 20 ns;
274     w_A <= B"1010";
        wait for 20 ns;
276     w_A <= B"1011";
        wait for 20 ns;
278     w_A <= B"1100";
        wait for 20 ns;
280     w_A <= B"1101";
        wait for 20 ns;
282     w_A <= B"1110";
        wait for 20 ns;
284     w_A <= B"1111";
        wait for 20 ns;
286     w_B <= B"0111";
        w_A <= B"0000";
288     wait for 20 ns;
        w_A <= B"0001";
290     wait for 20 ns;
        w_A <= B"0010";
292     wait for 20 ns;
        w_A <= B"0011";
294     wait for 20 ns;
        w_A <= B"0100";
296     wait for 20 ns;
        w_A <= B"0101";
```

```
298         wait for 20 ns;
            w_A <= B"0110";
300         wait for 20 ns;
            w_A <= B"0111";
302         wait for 20 ns;
            w_A <= B"1000";
304         wait for 20 ns;
            w_A <= B"1001";
306         wait for 20 ns;
            w_A <= B"1010";
308         wait for 20 ns;
            w_A <= B"1011";
310         wait for 20 ns;
            w_A <= B"1100";
312         wait for 20 ns;
            w_A <= B"1101";
314         wait for 20 ns;
            w_A <= B"1110";
316         wait for 20 ns;
            w_A <= B"1111";
318         wait for 20 ns;
            w_B <= B"1000";
320         w_A <= B"0000";
            wait for 20 ns;
322         w_A <= B"0001";
            wait for 20 ns;
324         w_A <= B"0010";
            wait for 20 ns;
326         w_A <= B"0011";
            wait for 20 ns;
328         w_A <= B"0100";
            wait for 20 ns;
330         w_A <= B"0101";
            wait for 20 ns;
332         w_A <= B"0110";
            wait for 20 ns;
334         w_A <= B"0111";
            wait for 20 ns;
336         w_A <= B"1000";
            wait for 20 ns;
338         w_A <= B"1001";
            wait for 20 ns;
340         w_A <= B"1010";
```

```
        wait for 20 ns;
342     w_A <= B"1011";
        wait for 20 ns;
344     w_A <= B"1100";
        wait for 20 ns;
346     w_A <= B"1101";
        wait for 20 ns;
348     w_A <= B"1110";
        wait for 20 ns;
350     w_A <= B"1111";
        wait for 20 ns;
352     w_B <= B"1001";
        w_A <= B"0000";
354     wait for 20 ns;
        w_A <= B"0001";
356     wait for 20 ns;
        w_A <= B"0010";
358     wait for 20 ns;
        w_A <= B"0011";
360     wait for 20 ns;
        w_A <= B"0100";
362     wait for 20 ns;
        w_A <= B"0101";
364     wait for 20 ns;
        w_A <= B"0110";
366     wait for 20 ns;
        w_A <= B"0111";
368     wait for 20 ns;
        w_A <= B"1000";
370     wait for 20 ns;
        w_A <= B"1001";
372     wait for 20 ns;
        w_A <= B"1010";
374     wait for 20 ns;
        w_A <= B"1011";
376     wait for 20 ns;
        w_A <= B"1100";
378     wait for 20 ns;
        w_A <= B"1101";
380     wait for 20 ns;
        w_A <= B"1110";
382     wait for 20 ns;
        w_A <= B"1111";
```

```
384         wait for 20 ns;
            w_B <= B"1010";
386         w_A <= B"0000";
            wait for 20 ns;
388         w_A <= B"0001";
            wait for 20 ns;
390         w_A <= B"0010";
            wait for 20 ns;
392         w_A <= B"0011";
            wait for 20 ns;
394         w_A <= B"0100";
            wait for 20 ns;
396         w_A <= B"0101";
            wait for 20 ns;
398         w_A <= B"0110";
            wait for 20 ns;
400         w_A <= B"0111";
            wait for 20 ns;
402         w_A <= B"1000";
            wait for 20 ns;
404         w_A <= B"1001";
            wait for 20 ns;
406         w_A <= B"1010";
            wait for 20 ns;
408         w_A <= B"1011";
            wait for 20 ns;
410         w_A <= B"1100";
            wait for 20 ns;
412         w_A <= B"1101";
            wait for 20 ns;
414         w_A <= B"1110";
            wait for 20 ns;
416         w_A <= B"1111";
            wait for 20 ns;
418         w_B <= B"1011";
            w_A <= B"0000";
420         wait for 20 ns;
            w_A <= B"0001";
422         wait for 20 ns;
            w_A <= B"0010";
424         wait for 20 ns;
            w_A <= B"0011";
426         wait for 20 ns;
```

```
        w_A <= B"0100";
        wait for 20 ns;
        w_A <= B"0101";
        wait for 20 ns;
        w_A <= B"0110";
        wait for 20 ns;
        w_A <= B"0111";
        wait for 20 ns;
        w_A <= B"1000";
        wait for 20 ns;
        w_A <= B"1001";
        wait for 20 ns;
        w_A <= B"1010";
        wait for 20 ns;
        w_A <= B"1011";
        wait for 20 ns;
        w_A <= B"1100";
        wait for 20 ns;
        w_A <= B"1101";
        wait for 20 ns;
        w_A <= B"1110";
        wait for 20 ns;
        w_A <= B"1111";
        wait for 20 ns;
        w_B <= B"1100";
        w_A <= B"0000";
        wait for 20 ns;
        w_A <= B"0001";
        wait for 20 ns;
        w_A <= B"0010";
        wait for 20 ns;
        w_A <= B"0011";
        wait for 20 ns;
        w_A <= B"0100";
        wait for 20 ns;
        w_A <= B"0101";
        wait for 20 ns;
        w_A <= B"0110";
        wait for 20 ns;
        w_A <= B"0111";
        wait for 20 ns;
        w_A <= B"1000";
        wait for 20 ns;
```

```
470         w_A <= B"1001";
            wait for 20 ns;
472         w_A <= B"1010";
            wait for 20 ns;
474         w_A <= B"1011";
            wait for 20 ns;
476         w_A <= B"1100";
            wait for 20 ns;
478         w_A <= B"1101";
            wait for 20 ns;
480         w_A <= B"1110";
            wait for 20 ns;
482         w_A <= B"1111";
            wait for 20 ns;
484         w_B <= B"1101";
            w_A <= B"0000";
486         wait for 20 ns;
            w_A <= B"0001";
488         wait for 20 ns;
            w_A <= B"0010";
490         wait for 20 ns;
            w_A <= B"0011";
492         wait for 20 ns;
            w_A <= B"0100";
494         wait for 20 ns;
            w_A <= B"0101";
496         wait for 20 ns;
            w_A <= B"0110";
498         wait for 20 ns;
            w_A <= B"0111";
500         wait for 20 ns;
            w_A <= B"1000";
502         wait for 20 ns;
            w_A <= B"1001";
504         wait for 20 ns;
            w_A <= B"1010";
506         wait for 20 ns;
            w_A <= B"1011";
508         wait for 20 ns;
            w_A <= B"1100";
510         wait for 20 ns;
            w_A <= B"1101";
512         wait for 20 ns;
```

```
        w_A <= B"1110";
514     wait for 20 ns;
        w_A <= B"1111";
516     wait for 20 ns;
        w_B <= B"1110";
518     w_A <= B"0000";
        wait for 20 ns;
520     w_A <= B"0001";
        wait for 20 ns;
522     w_A <= B"0010";
        wait for 20 ns;
524     w_A <= B"0011";
        wait for 20 ns;
526     w_A <= B"0100";
        wait for 20 ns;
528     w_A <= B"0101";
        wait for 20 ns;
530     w_A <= B"0110";
        wait for 20 ns;
532     w_A <= B"0111";
        wait for 20 ns;
534     w_A <= B"1000";
        wait for 20 ns;
536     w_A <= B"1001";
        wait for 20 ns;
538     w_A <= B"1010";
        wait for 20 ns;
540     w_A <= B"1011";
        wait for 20 ns;
542     w_A <= B"1100";
        wait for 20 ns;
544     w_A <= B"1101";
        wait for 20 ns;
546     w_A <= B"1110";
        wait for 20 ns;
548     w_A <= B"1111";
        wait for 20 ns;
550     w_B <= B"1111";
        w_A <= B"0000";
552     wait for 20 ns;
        w_A <= B"0001";
554     wait for 20 ns;
        w_A <= B"0010";
```

```
556         wait for 20 ns;
            w_A <= B"0011";
558         wait for 20 ns;
            w_A <= B"0100";
560         wait for 20 ns;
            w_A <= B"0101";
562         wait for 20 ns;
            w_A <= B"0110";
564         wait for 20 ns;
            w_A <= B"0111";
566         wait for 20 ns;
            w_A <= B"1000";
568         wait for 20 ns;
            w_A <= B"1001";
570         wait for 20 ns;
            w_A <= B"1010";
572         wait for 20 ns;
            w_A <= B"1011";
574         wait for 20 ns;
            w_A <= B"1100";
576         wait for 20 ns;
            w_A <= B"1101";
578         wait for 20 ns;
            w_A <= B"1110";
580         wait for 20 ns;
            w_A <= B"1111";
582         wait for 20 ns;
            w_c <= '1';
584         w_B <= B"0000";
            w_A <= B"0000";
586         wait for 20 ns;
            w_A <= B"0001";
588         wait for 20 ns;
            w_A <= B"0010";
590         wait for 20 ns;
            w_A <= B"0011";
592         wait for 20 ns;
            w_A <= B"0100";
594         wait for 20 ns;
            w_A <= B"0101";
596         wait for 20 ns;
            w_A <= B"0110";
598         wait for 20 ns;
```

```
        w_A <= B"0111";
600     wait for 20 ns;
        w_A <= B"1000";
602     wait for 20 ns;
        w_A <= B"1001";
604     wait for 20 ns;
        w_A <= B"1010";
606     wait for 20 ns;
        w_A <= B"1011";
608     wait for 20 ns;
        w_A <= B"1100";
610     wait for 20 ns;
        w_A <= B"1101";
612     wait for 20 ns;
        w_A <= B"1110";
614     wait for 20 ns;
        w_A <= B"1111";
616     wait for 20 ns;
        w_B <= B"0001";
618     w_A <= B"0000";
        wait for 20 ns;
620     w_A <= B"0001";
        wait for 20 ns;
622     w_A <= B"0010";
        wait for 20 ns;
624     w_A <= B"0011";
        wait for 20 ns;
626     w_A <= B"0100";
        wait for 20 ns;
628     w_A <= B"0101";
        wait for 20 ns;
630     w_A <= B"0110";
        wait for 20 ns;
632     w_A <= B"0111";
        wait for 20 ns;
634     w_A <= B"1000";
        wait for 20 ns;
636     w_A <= B"1001";
        wait for 20 ns;
638     w_A <= B"1010";
        wait for 20 ns;
640     w_A <= B"1011";
        wait for 20 ns;
```

```
642         w_A <= B"1100";
            wait for 20 ns;
644         w_A <= B"1101";
            wait for 20 ns;
646         w_A <= B"1110";
            wait for 20 ns;
648         w_A <= B"1111";
            wait for 20 ns;
650         w_B <= B"0010";
            w_A <= B"0000";
652         wait for 20 ns;
            w_A <= B"0001";
654         wait for 20 ns;
            w_A <= B"0010";
656         wait for 20 ns;
            w_A <= B"0011";
658         wait for 20 ns;
            w_A <= B"0100";
660         wait for 20 ns;
            w_A <= B"0101";
662         wait for 20 ns;
            w_A <= B"0110";
664         wait for 20 ns;
            w_A <= B"0111";
666         wait for 20 ns;
            w_A <= B"1000";
668         wait for 20 ns;
            w_A <= B"1001";
670         wait for 20 ns;
            w_A <= B"1010";
672         wait for 20 ns;
            w_A <= B"1011";
674         wait for 20 ns;
            w_A <= B"1100";
676         wait for 20 ns;
            w_A <= B"1101";
678         wait for 20 ns;
            w_A <= B"1110";
680         wait for 20 ns;
            w_A <= B"1111";
682         wait for 20 ns;
            w_B <= B"0011";
684         w_A <= B"0000";
```

```
        wait for 20 ns;
686     w_A <= B"0001";
        wait for 20 ns;
688     w_A <= B"0010";
        wait for 20 ns;
690     w_A <= B"0011";
        wait for 20 ns;
692     w_A <= B"0100";
        wait for 20 ns;
694     w_A <= B"0101";
        wait for 20 ns;
696     w_A <= B"0110";
        wait for 20 ns;
698     w_A <= B"0111";
        wait for 20 ns;
700     w_A <= B"1000";
        wait for 20 ns;
702     w_A <= B"1001";
        wait for 20 ns;
704     w_A <= B"1010";
        wait for 20 ns;
706     w_A <= B"1011";
        wait for 20 ns;
708     w_A <= B"1100";
        wait for 20 ns;
710     w_A <= B"1101";
        wait for 20 ns;
712     w_A <= B"1110";
        wait for 20 ns;
714     w_A <= B"1111";
        wait for 20 ns;
716     w_B <= B"0100";
        w_A <= B"0000";
718     wait for 20 ns;
        w_A <= B"0001";
720     wait for 20 ns;
        w_A <= B"0010";
722     wait for 20 ns;
        w_A <= B"0011";
724     wait for 20 ns;
        w_A <= B"0100";
726     wait for 20 ns;
        w_A <= B"0101";
```

```
728         wait for 20 ns;
            w_A <= B"0110";
730         wait for 20 ns;
            w_A <= B"0111";
732         wait for 20 ns;
            w_A <= B"1000";
734         wait for 20 ns;
            w_A <= B"1001";
736         wait for 20 ns;
            w_A <= B"1010";
738         wait for 20 ns;
            w_A <= B"1011";
740         wait for 20 ns;
            w_A <= B"1100";
742         wait for 20 ns;
            w_A <= B"1101";
744         wait for 20 ns;
            w_A <= B"1110";
746         wait for 20 ns;
            w_A <= B"1111";
748         wait for 20 ns;
            w_B <= B"0101";
750         w_A <= B"0000";
            wait for 20 ns;
752         w_A <= B"0001";
            wait for 20 ns;
754         w_A <= B"0010";
            wait for 20 ns;
756         w_A <= B"0011";
            wait for 20 ns;
758         w_A <= B"0100";
            wait for 20 ns;
760         w_A <= B"0101";
            wait for 20 ns;
762         w_A <= B"0110";
            wait for 20 ns;
764         w_A <= B"0111";
            wait for 20 ns;
766         w_A <= B"1000";
            wait for 20 ns;
768         w_A <= B"1001";
            wait for 20 ns;
770         w_A <= B"1010";
```

```
        wait for 20 ns;
772     w_A <= B"1011";
        wait for 20 ns;
774     w_A <= B"1100";
        wait for 20 ns;
776     w_A <= B"1101";
        wait for 20 ns;
778     w_A <= B"1110";
        wait for 20 ns;
780     w_A <= B"1111";
        wait for 20 ns;
782     w_B <= B"0110";
        w_A <= B"0000";
784     wait for 20 ns;
        w_A <= B"0001";
786     wait for 20 ns;
        w_A <= B"0010";
788     wait for 20 ns;
        w_A <= B"0011";
790     wait for 20 ns;
        w_A <= B"0100";
792     wait for 20 ns;
        w_A <= B"0101";
794     wait for 20 ns;
        w_A <= B"0110";
796     wait for 20 ns;
        w_A <= B"0111";
798     wait for 20 ns;
        w_A <= B"1000";
800     wait for 20 ns;
        w_A <= B"1001";
802     wait for 20 ns;
        w_A <= B"1010";
804     wait for 20 ns;
        w_A <= B"1011";
806     wait for 20 ns;
        w_A <= B"1100";
808     wait for 20 ns;
        w_A <= B"1101";
810     wait for 20 ns;
        w_A <= B"1110";
812     wait for 20 ns;
        w_A <= B"1111";
```

```
814     wait for 20 ns;
        w_B <= B"0111";
816     w_A <= B"0000";
        wait for 20 ns;
818     w_A <= B"0001";
        wait for 20 ns;
820     w_A <= B"0010";
        wait for 20 ns;
822     w_A <= B"0011";
        wait for 20 ns;
824     w_A <= B"0100";
        wait for 20 ns;
826     w_A <= B"0101";
        wait for 20 ns;
828     w_A <= B"0110";
        wait for 20 ns;
830     w_A <= B"0111";
        wait for 20 ns;
832     w_A <= B"1000";
        wait for 20 ns;
834     w_A <= B"1001";
        wait for 20 ns;
836     w_A <= B"1010";
        wait for 20 ns;
838     w_A <= B"1011";
        wait for 20 ns;
840     w_A <= B"1100";
        wait for 20 ns;
842     w_A <= B"1101";
        wait for 20 ns;
844     w_A <= B"1110";
        wait for 20 ns;
846     w_A <= B"1111";
        wait for 20 ns;
848     w_B <= B"1000";
        w_A <= B"0000";
850     wait for 20 ns;
        w_A <= B"0001";
852     wait for 20 ns;
        w_A <= B"0010";
854     wait for 20 ns;
        w_A <= B"0011";
856     wait for 20 ns;
```

```
        w_A <= B"0100";
858     wait for 20 ns;
        w_A <= B"0101";
860     wait for 20 ns;
        w_A <= B"0110";
862     wait for 20 ns;
        w_A <= B"0111";
864     wait for 20 ns;
        w_A <= B"1000";
866     wait for 20 ns;
        w_A <= B"1001";
868     wait for 20 ns;
        w_A <= B"1010";
870     wait for 20 ns;
        w_A <= B"1011";
872     wait for 20 ns;
        w_A <= B"1100";
874     wait for 20 ns;
        w_A <= B"1101";
876     wait for 20 ns;
        w_A <= B"1110";
878     wait for 20 ns;
        w_A <= B"1111";
880     wait for 20 ns;
        w_B <= B"1001";
882     w_A <= B"0000";
        wait for 20 ns;
884     w_A <= B"0001";
        wait for 20 ns;
886     w_A <= B"0010";
        wait for 20 ns;
888     w_A <= B"0011";
        wait for 20 ns;
890     w_A <= B"0100";
        wait for 20 ns;
892     w_A <= B"0101";
        wait for 20 ns;
894     w_A <= B"0110";
        wait for 20 ns;
896     w_A <= B"0111";
        wait for 20 ns;
898     w_A <= B"1000";
        wait for 20 ns;
```

```
900         w_A <= B"1001";
            wait for 20 ns;
902         w_A <= B"1010";
            wait for 20 ns;
904         w_A <= B"1011";
            wait for 20 ns;
906         w_A <= B"1100";
            wait for 20 ns;
908         w_A <= B"1101";
            wait for 20 ns;
910         w_A <= B"1110";
            wait for 20 ns;
912         w_A <= B"1111";
            wait for 20 ns;
914         w_B <= B"1010";
            w_A <= B"0000";
916         wait for 20 ns;
            w_A <= B"0001";
918         wait for 20 ns;
            w_A <= B"0010";
920         wait for 20 ns;
            w_A <= B"0011";
922         wait for 20 ns;
            w_A <= B"0100";
924         wait for 20 ns;
            w_A <= B"0101";
926         wait for 20 ns;
            w_A <= B"0110";
928         wait for 20 ns;
            w_A <= B"0111";
930         wait for 20 ns;
            w_A <= B"1000";
932         wait for 20 ns;
            w_A <= B"1001";
934         wait for 20 ns;
            w_A <= B"1010";
936         wait for 20 ns;
            w_A <= B"1011";
938         wait for 20 ns;
            w_A <= B"1100";
940         wait for 20 ns;
            w_A <= B"1101";
942         wait for 20 ns;
```

```
        w_A <= B"1110";
944     wait for 20 ns;
        w_A <= B"1111";
946     wait for 20 ns;
        w_B <= B"1011";
948     w_A <= B"0000";
        wait for 20 ns;
950     w_A <= B"0001";
        wait for 20 ns;
952     w_A <= B"0010";
        wait for 20 ns;
954     w_A <= B"0011";
        wait for 20 ns;
956     w_A <= B"0100";
        wait for 20 ns;
958     w_A <= B"0101";
        wait for 20 ns;
960     w_A <= B"0110";
        wait for 20 ns;
962     w_A <= B"0111";
        wait for 20 ns;
964     w_A <= B"1000";
        wait for 20 ns;
966     w_A <= B"1001";
        wait for 20 ns;
968     w_A <= B"1010";
        wait for 20 ns;
970     w_A <= B"1011";
        wait for 20 ns;
972     w_A <= B"1100";
        wait for 20 ns;
974     w_A <= B"1101";
        wait for 20 ns;
976     w_A <= B"1110";
        wait for 20 ns;
978     w_A <= B"1111";
        wait for 20 ns;
980     w_B <= B"1100";
        w_A <= B"0000";
982     wait for 20 ns;
        w_A <= B"0001";
984     wait for 20 ns;
        w_A <= B"0010";
```

```
986         wait for 20 ns;
            w_A <= B"0011";
988         wait for 20 ns;
            w_A <= B"0100";
990         wait for 20 ns;
            w_A <= B"0101";
992         wait for 20 ns;
            w_A <= B"0110";
994         wait for 20 ns;
            w_A <= B"0111";
996         wait for 20 ns;
            w_A <= B"1000";
998         wait for 20 ns;
            w_A <= B"1001";
1000        wait for 20 ns;
            w_A <= B"1010";
1002        wait for 20 ns;
            w_A <= B"1011";
1004        wait for 20 ns;
            w_A <= B"1100";
1006        wait for 20 ns;
            w_A <= B"1101";
1008        wait for 20 ns;
            w_A <= B"1110";
1010        wait for 20 ns;
            w_A <= B"1111";
1012        wait for 20 ns;
            w_B <= B"1101";
1014        w_A <= B"0000";
            wait for 20 ns;
1016        w_A <= B"0001";
            wait for 20 ns;
1018        w_A <= B"0010";
            wait for 20 ns;
1020        w_A <= B"0011";
            wait for 20 ns;
1022        w_A <= B"0100";
            wait for 20 ns;
1024        w_A <= B"0101";
            wait for 20 ns;
1026        w_A <= B"0110";
            wait for 20 ns;
1028        w_A <= B"0111";
```

```
        wait for 20 ns;
1030    w_A <= B"1000";
        wait for 20 ns;
1032    w_A <= B"1001";
        wait for 20 ns;
1034    w_A <= B"1010";
        wait for 20 ns;
1036    w_A <= B"1011";
        wait for 20 ns;
1038    w_A <= B"1100";
        wait for 20 ns;
1040    w_A <= B"1101";
        wait for 20 ns;
1042    w_A <= B"1110";
        wait for 20 ns;
1044    w_A <= B"1111";
        wait for 20 ns;
1046    w_B <= B"1110";
        w_A <= B"0000";
1048    wait for 20 ns;
        w_A <= B"0001";
1050    wait for 20 ns;
        w_A <= B"0010";
1052    wait for 20 ns;
        w_A <= B"0011";
1054    wait for 20 ns;
        w_A <= B"0100";
1056    wait for 20 ns;
        w_A <= B"0101";
1058    wait for 20 ns;
        w_A <= B"0110";
1060    wait for 20 ns;
        w_A <= B"0111";
1062    wait for 20 ns;
        w_A <= B"1000";
1064    wait for 20 ns;
        w_A <= B"1001";
1066    wait for 20 ns;
        w_A <= B"1010";
1068    wait for 20 ns;
        w_A <= B"1011";
1070    wait for 20 ns;
        w_A <= B"1100";
```

```
1072        wait for 20 ns;
            w_A <= B"1101";
1074        wait for 20 ns;
            w_A <= B"1110";
1076        wait for 20 ns;
            w_A <= B"1111";
1078        wait for 20 ns;
            w_B <= B"1111";
1080        w_A <= B"0000";
            wait for 20 ns;
1082        w_A <= B"0001";
            wait for 20 ns;
1084        w_A <= B"0010";
            wait for 20 ns;
1086        w_A <= B"0011";
            wait for 20 ns;
1088        w_A <= B"0100";
            wait for 20 ns;
1090        w_A <= B"0101";
            wait for 20 ns;
1092        w_A <= B"0110";
            wait for 20 ns;
1094        w_A <= B"0111";
            wait for 20 ns;
1096        w_A <= B"1000";
            wait for 20 ns;
1098        w_A <= B"1001";
            wait for 20 ns;
1100        w_A <= B"1010";
            wait for 20 ns;
1102        w_A <= B"1011";
            wait for 20 ns;
1104        w_A <= B"1100";
            wait for 20 ns;
1106        w_A <= B"1101";
            wait for 20 ns;
1108        w_A <= B"1110";
            wait for 20 ns;
1110        w_A <= B"1111";
            wait for 20 ns;
1112        wait;
        end process stimulus;
1114    end testbench;
```

Listing B.21. CARRY_SELECT_ADDER_4_TB.vhd Code

2.8.4 32-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_32_TB
--| Tests CARRY_SELECT_ADDER_32
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_32_TB is
end CARRY_SELECT_ADDER_32_TB;

architecture testbench of CARRY_SELECT_ADDER_32_TB is
  component CARRY_SELECT_ADDER_32 is
    port (i_A      : in  std_logic_vector(31 downto 0);
          i_A_n    : in  std_logic_vector(31 downto 0);
          i_B      : in  std_logic_vector(31 downto 0);
          i_B_n    : in  std_logic_vector(31 downto 0);
          i_AB_AND    : in  std_logic_vector(31 downto 0);
          i_AB_NAND : in  std_logic_vector(31 downto 0);
          i_AB_NOR    : in  std_logic_vector(31 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(31 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(31 downto 0) := (others
   =>'0');
  signal w_A_n : std_logic_vector(31 downto 0);
  signal w_B : std_logic_vector(31 downto 0) := (others
   =>'0');
  signal w_B_n : std_logic_vector(31 downto 0);
  signal w_AB_AND : std_logic_vector(31 downto 0);
  signal w_AB_NAND : std_logic_vector(31 downto 0);
  signal w_AB_NOR : std_logic_vector(31 downto 0);
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(31 downto 0);
  signal w_Cout  : std_logic;

begin
  -- Connect input wires

  w_A_n <= NOT w_A;
  w_B_n <= NOT w_B;
  w_AB_AND <= w_A AND w_B;
```

```
40    w_AB_NAND <= NOT w_AB_AND;
      w_AB_NOR <= NOT (w_A OR w_B);
42    -- Connect CARRY_SELECT_ADDER_32
      u_CARRY_SELECT_ADDER_32: CARRY_SELECT_ADDER_32
44    port map (i_A => w_A,
              i_A_n => w_A_n,
46            i_B => w_B,
              i_B_n => w_B_n,
48            i_AB_AND => w_AB_AND,
              i_AB_NAND => w_AB_NAND,
50            i_AB_NOR => w_AB_NOR,
              i_C => w_C,
52            o_S => w_S,
              o_C => w_Cout);
54
      -- Process for stimulating inputs
56    stimulus: process is
      begin
58      wait for 20 ns;
        w_A <= B"0000_0000_0000_0000_0000_0000_0000_1111";
60      w_B <= B"0000_0000_0000_0000_0000_0000_0000_0001";
        wait for 20 ns;
62      w_C <= '1';
        wait for 20 ns;
64      w_A <= B"1111_1111_1111_1111_1111_1111_1111_1111";
        wait for 20 ns;
66      w_C <= '0';
        wait for 20 ns;
68      w_B <= B"0000_0000_0000_0000_0000_0000_0000_1111";
        wait for 20 ns;
70      w_C <= '1';
        wait for 20 ns;
72      w_A <= B"0000_0000_0000_0000_0000_0000_0000_1111";
        w_B <= B"0000_0000_0000_0000_0000_0000_1111_0000";
74      wait;
      end process stimulus;
76  end testbench;
```

Listing B.22. CARRY_SELECT_ADDER_32_TB.vhd Code

2.8.5 Self-Contained 1-bit Full Adder

```
--| FULL_ADDER_1_SC_TB
--| Tests FULL_ADDER_1_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_1_SC_TB is
end FULL_ADDER_1_SC_TB;

architecture testbench of FULL_ADDER_1_SC_TB is
  component FULL_ADDER_1_SC is
    port (i_A      : in  std_logic;
          i_B      : in  std_logic;
          i_C      : in  std_logic;
          o_S      : out std_logic;
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic := '0';
  signal w_B : std_logic := '0';
  signal w_C : std_logic := '0';
  signal w_S : std_logic;
  signal w_Cout  : std_logic;
begin
  -- Connect FULL_ADDER_1_SC
  u_FULL_ADDER_1_SC: FULL_ADDER_1_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_C <= '1';
    wait for 20 ns;
    w_C <= '0';
    w_B <= '1';
    wait for 20 ns;
    w_C <= '1';
```

```
42      wait for 20 ns;
        w_C <= '0';
44      w_B <= '0';
        w_A <= '1';
46      wait for 20 ns;
        w_C <= '1';
48      wait for 20 ns;
        w_C <= '0';
50      w_B <= '1';
        wait for 20 ns;
52      w_C <= '1';
        wait for 20 ns;
54      wait;
    end process stimulus;
56  end testbench;
```

Listing B.23. FULL_ADDER_1_SC_TB.vhd Code

2.8.6 Self-Contained 2-bit Full Adder

```
--| FULL_ADDER_2_SC_TB
--| Tests FULL_ADDER_2_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_2_SC_TB is
end FULL_ADDER_2_SC_TB;

architecture testbench of FULL_ADDER_2_SC_TB is
  component FULL_ADDER_2_SC is
    port (i_A      : in  std_logic_vector(1 downto 0);
          i_B      : in  std_logic_vector(1 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(1 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(1 downto 0) := (others =>'0')
   ;
  signal w_B : std_logic_vector(1 downto 0) := (others =>'0')
   ;
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(1 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect FULL_ADDER_2_SC
  u_FULL_ADDER_2_SC: FULL_ADDER_2_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"01";
    wait for 20 ns;
    w_A <= B"10";
    wait for 20 ns;
```

```
40      w_A <= B"11";
        wait for 20 ns;
42      w_A <= B"00";
        w_B <= B"01";
44      wait for 20 ns;
        w_A <= B"01";
46      wait for 20 ns;
        w_A <= B"10";
48      wait for 20 ns;
        w_A <= B"11";
50      wait for 20 ns;
        w_A <= B"00";
52      w_B <= B"10";
        wait for 20 ns;
54      w_A <= B"01";
        wait for 20 ns;
56      w_A <= B"10";
        wait for 20 ns;
58      w_A <= B"11";
        wait for 20 ns;
60      w_A <= B"00";
        w_B <= B"11";
62      wait for 20 ns;
        w_A <= B"01";
64      wait for 20 ns;
        w_A <= B"10";
66      wait for 20 ns;
        w_A <= B"11";
68      wait for 20 ns;
        w_A <= B"00";
70      w_B <= B"00";
        w_C <= '1';
72      wait for 20 ns;
        w_A <= B"01";
74      wait for 20 ns;
        w_A <= B"10";
76      wait for 20 ns;
        w_A <= B"11";
78      wait for 20 ns;
        w_A <= B"00";
80      w_B <= B"01";
        wait for 20 ns;
82      w_A <= B"01";
```

```
    wait for 20 ns;
84  w_A <= B"10";
    wait for 20 ns;
86  w_A <= B"11";
    wait for 20 ns;
88  w_A <= B"00";
    w_B <= B"10";
90  wait for 20 ns;
    w_A <= B"01";
92  wait for 20 ns;
    w_A <= B"10";
94  wait for 20 ns;
    w_A <= B"11";
96  wait for 20 ns;
    w_A <= B"00";
98  w_B <= B"11";
    wait for 20 ns;
100 w_A <= B"01";
    wait for 20 ns;
102 w_A <= B"10";
    wait for 20 ns;
104 w_A <= B"11";
    wait for 20 ns;
106 wait;
  end process stimulus;
108 end testbench;
```

Listing B.24. FULL_ADDER_2_SC_TB.vhd Code

2.8.7 Self-Contained 4-bit Full Adder

```
--| FULL_ADDER_4_SC_TB
--| Tests FULL_ADDER_4_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity FULL_ADDER_4_SC_TB is
end FULL_ADDER_4_SC_TB;

architecture testbench of FULL_ADDER_4_SC_TB is
  component FULL_ADDER_4_SC is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(3 downto 0) := (others =>'0');
  signal w_B : std_logic_vector(3 downto 0) := (others =>'0');
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(3 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect FULL_ADDER_4_SC
  u_FULL_ADDER_4_SC: FULL_ADDER_4_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"0001";
    wait for 20 ns;
    w_A <= B"0010";
    wait for 20 ns;
```

```
40      w_A <= B"0011";
        wait for 20 ns;
42      w_A <= B"0100";
        wait for 20 ns;
44      w_A <= B"0101";
        wait for 20 ns;
46      w_A <= B"0110";
        wait for 20 ns;
48      w_A <= B"0111";
        wait for 20 ns;
50      w_A <= B"1000";
        wait for 20 ns;
52      w_A <= B"1001";
        wait for 20 ns;
54      w_A <= B"1010";
        wait for 20 ns;
56      w_A <= B"1011";
        wait for 20 ns;
58      w_A <= B"1100";
        wait for 20 ns;
60      w_A <= B"1101";
        wait for 20 ns;
62      w_A <= B"1110";
        wait for 20 ns;
64      w_A <= B"1111";
        wait for 20 ns;
66      w_B <= B"0001";
        w_A <= B"0000";
68      wait for 20 ns;
        w_A <= B"0001";
70      wait for 20 ns;
        w_A <= B"0010";
72      wait for 20 ns;
        w_A <= B"0011";
74      wait for 20 ns;
        w_A <= B"0100";
76      wait for 20 ns;
        w_A <= B"0101";
78      wait for 20 ns;
        w_A <= B"0110";
80      wait for 20 ns;
        w_A <= B"0111";
82      wait for 20 ns;
```

```
        w_A <= B"1000";
84      wait for 20 ns;
        w_A <= B"1001";
86      wait for 20 ns;
        w_A <= B"1010";
88      wait for 20 ns;
        w_A <= B"1011";
90      wait for 20 ns;
        w_A <= B"1100";
92      wait for 20 ns;
        w_A <= B"1101";
94      wait for 20 ns;
        w_A <= B"1110";
96      wait for 20 ns;
        w_A <= B"1111";
98      wait for 20 ns;
        w_B <= B"0010";
100     w_A <= B"0000";
        wait for 20 ns;
102     w_A <= B"0001";
        wait for 20 ns;
104     w_A <= B"0010";
        wait for 20 ns;
106     w_A <= B"0011";
        wait for 20 ns;
108     w_A <= B"0100";
        wait for 20 ns;
110     w_A <= B"0101";
        wait for 20 ns;
112     w_A <= B"0110";
        wait for 20 ns;
114     w_A <= B"0111";
        wait for 20 ns;
116     w_A <= B"1000";
        wait for 20 ns;
118     w_A <= B"1001";
        wait for 20 ns;
120     w_A <= B"1010";
        wait for 20 ns;
122     w_A <= B"1011";
        wait for 20 ns;
124     w_A <= B"1100";
        wait for 20 ns;
```

```
126         w_A <= B"1101";
            wait for 20 ns;
128         w_A <= B"1110";
            wait for 20 ns;
130         w_A <= B"1111";
            wait for 20 ns;
132         w_B <= B"0011";
            w_A <= B"0000";
134         wait for 20 ns;
            w_A <= B"0001";
136         wait for 20 ns;
            w_A <= B"0010";
138         wait for 20 ns;
            w_A <= B"0011";
140         wait for 20 ns;
            w_A <= B"0100";
142         wait for 20 ns;
            w_A <= B"0101";
144         wait for 20 ns;
            w_A <= B"0110";
146         wait for 20 ns;
            w_A <= B"0111";
148         wait for 20 ns;
            w_A <= B"1000";
150         wait for 20 ns;
            w_A <= B"1001";
152         wait for 20 ns;
            w_A <= B"1010";
154         wait for 20 ns;
            w_A <= B"1011";
156         wait for 20 ns;
            w_A <= B"1100";
158         wait for 20 ns;
            w_A <= B"1101";
160         wait for 20 ns;
            w_A <= B"1110";
162         wait for 20 ns;
            w_A <= B"1111";
164         wait for 20 ns;
            w_B <= B"0100";
166         w_A <= B"0000";
            wait for 20 ns;
168         w_A <= B"0001";
```

```
        wait for 20 ns;
170     w_A <= B"0010";
        wait for 20 ns;
172     w_A <= B"0011";
        wait for 20 ns;
174     w_A <= B"0100";
        wait for 20 ns;
176     w_A <= B"0101";
        wait for 20 ns;
178     w_A <= B"0110";
        wait for 20 ns;
180     w_A <= B"0111";
        wait for 20 ns;
182     w_A <= B"1000";
        wait for 20 ns;
184     w_A <= B"1001";
        wait for 20 ns;
186     w_A <= B"1010";
        wait for 20 ns;
188     w_A <= B"1011";
        wait for 20 ns;
190     w_A <= B"1100";
        wait for 20 ns;
192     w_A <= B"1101";
        wait for 20 ns;
194     w_A <= B"1110";
        wait for 20 ns;
196     w_A <= B"1111";
        wait for 20 ns;
198     w_B <= B"0101";
        w_A <= B"0000";
200     wait for 20 ns;
        w_A <= B"0001";
202     wait for 20 ns;
        w_A <= B"0010";
204     wait for 20 ns;
        w_A <= B"0011";
206     wait for 20 ns;
        w_A <= B"0100";
208     wait for 20 ns;
        w_A <= B"0101";
210     wait for 20 ns;
        w_A <= B"0110";
```

```
212         wait for 20 ns;
            w_A <= B"0111";
214         wait for 20 ns;
            w_A <= B"1000";
216         wait for 20 ns;
            w_A <= B"1001";
218         wait for 20 ns;
            w_A <= B"1010";
220         wait for 20 ns;
            w_A <= B"1011";
222         wait for 20 ns;
            w_A <= B"1100";
224         wait for 20 ns;
            w_A <= B"1101";
226         wait for 20 ns;
            w_A <= B"1110";
228         wait for 20 ns;
            w_A <= B"1111";
230         wait for 20 ns;
            w_B <= B"0110";
232         w_A <= B"0000";
            wait for 20 ns;
234         w_A <= B"0001";
            wait for 20 ns;
236         w_A <= B"0010";
            wait for 20 ns;
238         w_A <= B"0011";
            wait for 20 ns;
240         w_A <= B"0100";
            wait for 20 ns;
242         w_A <= B"0101";
            wait for 20 ns;
244         w_A <= B"0110";
            wait for 20 ns;
246         w_A <= B"0111";
            wait for 20 ns;
248         w_A <= B"1000";
            wait for 20 ns;
250         w_A <= B"1001";
            wait for 20 ns;
252         w_A <= B"1010";
            wait for 20 ns;
254         w_A <= B"1011";
```

```
        wait for 20 ns;
256     w_A <= B"1100";
        wait for 20 ns;
258     w_A <= B"1101";
        wait for 20 ns;
260     w_A <= B"1110";
        wait for 20 ns;
262     w_A <= B"1111";
        wait for 20 ns;
264     w_B <= B"0111";
        w_A <= B"0000";
266     wait for 20 ns;
        w_A <= B"0001";
268     wait for 20 ns;
        w_A <= B"0010";
270     wait for 20 ns;
        w_A <= B"0011";
272     wait for 20 ns;
        w_A <= B"0100";
274     wait for 20 ns;
        w_A <= B"0101";
276     wait for 20 ns;
        w_A <= B"0110";
278     wait for 20 ns;
        w_A <= B"0111";
280     wait for 20 ns;
        w_A <= B"1000";
282     wait for 20 ns;
        w_A <= B"1001";
284     wait for 20 ns;
        w_A <= B"1010";
286     wait for 20 ns;
        w_A <= B"1011";
288     wait for 20 ns;
        w_A <= B"1100";
290     wait for 20 ns;
        w_A <= B"1101";
292     wait for 20 ns;
        w_A <= B"1110";
294     wait for 20 ns;
        w_A <= B"1111";
296     wait for 20 ns;
        w_B <= B"1000";
```

```
298         w_A <= B"0000";
            wait for 20 ns;
300         w_A <= B"0001";
            wait for 20 ns;
302         w_A <= B"0010";
            wait for 20 ns;
304         w_A <= B"0011";
            wait for 20 ns;
306         w_A <= B"0100";
            wait for 20 ns;
308         w_A <= B"0101";
            wait for 20 ns;
310         w_A <= B"0110";
            wait for 20 ns;
312         w_A <= B"0111";
            wait for 20 ns;
314         w_A <= B"1000";
            wait for 20 ns;
316         w_A <= B"1001";
            wait for 20 ns;
318         w_A <= B"1010";
            wait for 20 ns;
320         w_A <= B"1011";
            wait for 20 ns;
322         w_A <= B"1100";
            wait for 20 ns;
324         w_A <= B"1101";
            wait for 20 ns;
326         w_A <= B"1110";
            wait for 20 ns;
328         w_A <= B"1111";
            wait for 20 ns;
330         w_B <= B"1001";
            w_A <= B"0000";
332         wait for 20 ns;
            w_A <= B"0001";
334         wait for 20 ns;
            w_A <= B"0010";
336         wait for 20 ns;
            w_A <= B"0011";
338         wait for 20 ns;
            w_A <= B"0100";
340         wait for 20 ns;
```

```
        w_A <= B"0101";
342     wait for 20 ns;
        w_A <= B"0110";
344     wait for 20 ns;
        w_A <= B"0111";
346     wait for 20 ns;
        w_A <= B"1000";
348     wait for 20 ns;
        w_A <= B"1001";
350     wait for 20 ns;
        w_A <= B"1010";
352     wait for 20 ns;
        w_A <= B"1011";
354     wait for 20 ns;
        w_A <= B"1100";
356     wait for 20 ns;
        w_A <= B"1101";
358     wait for 20 ns;
        w_A <= B"1110";
360     wait for 20 ns;
        w_A <= B"1111";
362     wait for 20 ns;
        w_B <= B"1010";
364     w_A <= B"0000";
        wait for 20 ns;
366     w_A <= B"0001";
        wait for 20 ns;
368     w_A <= B"0010";
        wait for 20 ns;
370     w_A <= B"0011";
        wait for 20 ns;
372     w_A <= B"0100";
        wait for 20 ns;
374     w_A <= B"0101";
        wait for 20 ns;
376     w_A <= B"0110";
        wait for 20 ns;
378     w_A <= B"0111";
        wait for 20 ns;
380     w_A <= B"1000";
        wait for 20 ns;
382     w_A <= B"1001";
        wait for 20 ns;
```

```
384         w_A <= B"1010";
            wait for 20 ns;
386         w_A <= B"1011";
            wait for 20 ns;
388         w_A <= B"1100";
            wait for 20 ns;
390         w_A <= B"1101";
            wait for 20 ns;
392         w_A <= B"1110";
            wait for 20 ns;
394         w_A <= B"1111";
            wait for 20 ns;
396         w_B <= B"1011";
            w_A <= B"0000";
398         wait for 20 ns;
            w_A <= B"0001";
400         wait for 20 ns;
            w_A <= B"0010";
402         wait for 20 ns;
            w_A <= B"0011";
404         wait for 20 ns;
            w_A <= B"0100";
406         wait for 20 ns;
            w_A <= B"0101";
408         wait for 20 ns;
            w_A <= B"0110";
410         wait for 20 ns;
            w_A <= B"0111";
412         wait for 20 ns;
            w_A <= B"1000";
414         wait for 20 ns;
            w_A <= B"1001";
416         wait for 20 ns;
            w_A <= B"1010";
418         wait for 20 ns;
            w_A <= B"1011";
420         wait for 20 ns;
            w_A <= B"1100";
422         wait for 20 ns;
            w_A <= B"1101";
424         wait for 20 ns;
            w_A <= B"1110";
426         wait for 20 ns;
```

```
        w_A <= B"1111";
428     wait for 20 ns;
        w_B <= B"1100";
430     w_A <= B"0000";
        wait for 20 ns;
432     w_A <= B"0001";
        wait for 20 ns;
434     w_A <= B"0010";
        wait for 20 ns;
436     w_A <= B"0011";
        wait for 20 ns;
438     w_A <= B"0100";
        wait for 20 ns;
440     w_A <= B"0101";
        wait for 20 ns;
442     w_A <= B"0110";
        wait for 20 ns;
444     w_A <= B"0111";
        wait for 20 ns;
446     w_A <= B"1000";
        wait for 20 ns;
448     w_A <= B"1001";
        wait for 20 ns;
450     w_A <= B"1010";
        wait for 20 ns;
452     w_A <= B"1011";
        wait for 20 ns;
454     w_A <= B"1100";
        wait for 20 ns;
456     w_A <= B"1101";
        wait for 20 ns;
458     w_A <= B"1110";
        wait for 20 ns;
460     w_A <= B"1111";
        wait for 20 ns;
462     w_B <= B"1101";
        w_A <= B"0000";
464     wait for 20 ns;
        w_A <= B"0001";
466     wait for 20 ns;
        w_A <= B"0010";
468     wait for 20 ns;
        w_A <= B"0011";
```

```
470         wait for 20 ns;
            w_A <= B"0100";
472         wait for 20 ns;
            w_A <= B"0101";
474         wait for 20 ns;
            w_A <= B"0110";
476         wait for 20 ns;
            w_A <= B"0111";
478         wait for 20 ns;
            w_A <= B"1000";
480         wait for 20 ns;
            w_A <= B"1001";
482         wait for 20 ns;
            w_A <= B"1010";
484         wait for 20 ns;
            w_A <= B"1011";
486         wait for 20 ns;
            w_A <= B"1100";
488         wait for 20 ns;
            w_A <= B"1101";
490         wait for 20 ns;
            w_A <= B"1110";
492         wait for 20 ns;
            w_A <= B"1111";
494         wait for 20 ns;
            w_B <= B"1110";
496         w_A <= B"0000";
            wait for 20 ns;
498         w_A <= B"0001";
            wait for 20 ns;
500         w_A <= B"0010";
            wait for 20 ns;
502         w_A <= B"0011";
            wait for 20 ns;
504         w_A <= B"0100";
            wait for 20 ns;
506         w_A <= B"0101";
            wait for 20 ns;
508         w_A <= B"0110";
            wait for 20 ns;
510         w_A <= B"0111";
            wait for 20 ns;
512         w_A <= B"1000";
```

```
        wait for 20 ns;
514     w_A <= B"1001";
        wait for 20 ns;
516     w_A <= B"1010";
        wait for 20 ns;
518     w_A <= B"1011";
        wait for 20 ns;
520     w_A <= B"1100";
        wait for 20 ns;
522     w_A <= B"1101";
        wait for 20 ns;
524     w_A <= B"1110";
        wait for 20 ns;
526     w_A <= B"1111";
        wait for 20 ns;
528     w_B <= B"1111";
        w_A <= B"0000";
530     wait for 20 ns;
        w_A <= B"0001";
532     wait for 20 ns;
        w_A <= B"0010";
534     wait for 20 ns;
        w_A <= B"0011";
536     wait for 20 ns;
        w_A <= B"0100";
538     wait for 20 ns;
        w_A <= B"0101";
540     wait for 20 ns;
        w_A <= B"0110";
542     wait for 20 ns;
        w_A <= B"0111";
544     wait for 20 ns;
        w_A <= B"1000";
546     wait for 20 ns;
        w_A <= B"1001";
548     wait for 20 ns;
        w_A <= B"1010";
550     wait for 20 ns;
        w_A <= B"1011";
552     wait for 20 ns;
        w_A <= B"1100";
554     wait for 20 ns;
        w_A <= B"1101";
```

```
556         wait for 20 ns;
            w_A <= B"1110";
558         wait for 20 ns;
            w_A <= B"1111";
560         wait for 20 ns;
            w_c <= '1';
562         w_B <= B"0000";
            w_A <= B"0000";
564         wait for 20 ns;
            w_A <= B"0001";
566         wait for 20 ns;
            w_A <= B"0010";
568         wait for 20 ns;
            w_A <= B"0011";
570         wait for 20 ns;
            w_A <= B"0100";
572         wait for 20 ns;
            w_A <= B"0101";
574         wait for 20 ns;
            w_A <= B"0110";
576         wait for 20 ns;
            w_A <= B"0111";
578         wait for 20 ns;
            w_A <= B"1000";
580         wait for 20 ns;
            w_A <= B"1001";
582         wait for 20 ns;
            w_A <= B"1010";
584         wait for 20 ns;
            w_A <= B"1011";
586         wait for 20 ns;
            w_A <= B"1100";
588         wait for 20 ns;
            w_A <= B"1101";
590         wait for 20 ns;
            w_A <= B"1110";
592         wait for 20 ns;
            w_A <= B"1111";
594         wait for 20 ns;
            w_B <= B"0001";
596         w_A <= B"0000";
            wait for 20 ns;
598         w_A <= B"0001";
```

```
        wait for 20 ns;
600     w_A <= B"0010";
        wait for 20 ns;
602     w_A <= B"0011";
        wait for 20 ns;
604     w_A <= B"0100";
        wait for 20 ns;
606     w_A <= B"0101";
        wait for 20 ns;
608     w_A <= B"0110";
        wait for 20 ns;
610     w_A <= B"0111";
        wait for 20 ns;
612     w_A <= B"1000";
        wait for 20 ns;
614     w_A <= B"1001";
        wait for 20 ns;
616     w_A <= B"1010";
        wait for 20 ns;
618     w_A <= B"1011";
        wait for 20 ns;
620     w_A <= B"1100";
        wait for 20 ns;
622     w_A <= B"1101";
        wait for 20 ns;
624     w_A <= B"1110";
        wait for 20 ns;
626     w_A <= B"1111";
        wait for 20 ns;
628     w_B <= B"0010";
        w_A <= B"0000";
630     wait for 20 ns;
        w_A <= B"0001";
632     wait for 20 ns;
        w_A <= B"0010";
634     wait for 20 ns;
        w_A <= B"0011";
636     wait for 20 ns;
        w_A <= B"0100";
638     wait for 20 ns;
        w_A <= B"0101";
640     wait for 20 ns;
        w_A <= B"0110";
```

```
642         wait for 20 ns;
            w_A <= B"0111";
644         wait for 20 ns;
            w_A <= B"1000";
646         wait for 20 ns;
            w_A <= B"1001";
648         wait for 20 ns;
            w_A <= B"1010";
650         wait for 20 ns;
            w_A <= B"1011";
652         wait for 20 ns;
            w_A <= B"1100";
654         wait for 20 ns;
            w_A <= B"1101";
656         wait for 20 ns;
            w_A <= B"1110";
658         wait for 20 ns;
            w_A <= B"1111";
660         wait for 20 ns;
            w_B <= B"0011";
662         w_A <= B"0000";
            wait for 20 ns;
664         w_A <= B"0001";
            wait for 20 ns;
666         w_A <= B"0010";
            wait for 20 ns;
668         w_A <= B"0011";
            wait for 20 ns;
670         w_A <= B"0100";
            wait for 20 ns;
672         w_A <= B"0101";
            wait for 20 ns;
674         w_A <= B"0110";
            wait for 20 ns;
676         w_A <= B"0111";
            wait for 20 ns;
678         w_A <= B"1000";
            wait for 20 ns;
680         w_A <= B"1001";
            wait for 20 ns;
682         w_A <= B"1010";
            wait for 20 ns;
684         w_A <= B"1011";
```

```
        wait for 20 ns;
686     w_A <= B"1100";
        wait for 20 ns;
688     w_A <= B"1101";
        wait for 20 ns;
690     w_A <= B"1110";
        wait for 20 ns;
692     w_A <= B"1111";
        wait for 20 ns;
694     w_B <= B"0100";
        w_A <= B"0000";
696     wait for 20 ns;
        w_A <= B"0001";
698     wait for 20 ns;
        w_A <= B"0010";
700     wait for 20 ns;
        w_A <= B"0011";
702     wait for 20 ns;
        w_A <= B"0100";
704     wait for 20 ns;
        w_A <= B"0101";
706     wait for 20 ns;
        w_A <= B"0110";
708     wait for 20 ns;
        w_A <= B"0111";
710     wait for 20 ns;
        w_A <= B"1000";
712     wait for 20 ns;
        w_A <= B"1001";
714     wait for 20 ns;
        w_A <= B"1010";
716     wait for 20 ns;
        w_A <= B"1011";
718     wait for 20 ns;
        w_A <= B"1100";
720     wait for 20 ns;
        w_A <= B"1101";
722     wait for 20 ns;
        w_A <= B"1110";
724     wait for 20 ns;
        w_A <= B"1111";
726     wait for 20 ns;
        w_B <= B"0101";
```

```
728         w_A <= B"0000";
            wait for 20 ns;
730         w_A <= B"0001";
            wait for 20 ns;
732         w_A <= B"0010";
            wait for 20 ns;
734         w_A <= B"0011";
            wait for 20 ns;
736         w_A <= B"0100";
            wait for 20 ns;
738         w_A <= B"0101";
            wait for 20 ns;
740         w_A <= B"0110";
            wait for 20 ns;
742         w_A <= B"0111";
            wait for 20 ns;
744         w_A <= B"1000";
            wait for 20 ns;
746         w_A <= B"1001";
            wait for 20 ns;
748         w_A <= B"1010";
            wait for 20 ns;
750         w_A <= B"1011";
            wait for 20 ns;
752         w_A <= B"1100";
            wait for 20 ns;
754         w_A <= B"1101";
            wait for 20 ns;
756         w_A <= B"1110";
            wait for 20 ns;
758         w_A <= B"1111";
            wait for 20 ns;
760         w_B <= B"0110";
            w_A <= B"0000";
762         wait for 20 ns;
            w_A <= B"0001";
764         wait for 20 ns;
            w_A <= B"0010";
766         wait for 20 ns;
            w_A <= B"0011";
768         wait for 20 ns;
            w_A <= B"0100";
770         wait for 20 ns;
```

```
        w_A <= B"0101";
772     wait for 20 ns;
        w_A <= B"0110";
774     wait for 20 ns;
        w_A <= B"0111";
776     wait for 20 ns;
        w_A <= B"1000";
778     wait for 20 ns;
        w_A <= B"1001";
780     wait for 20 ns;
        w_A <= B"1010";
782     wait for 20 ns;
        w_A <= B"1011";
784     wait for 20 ns;
        w_A <= B"1100";
786     wait for 20 ns;
        w_A <= B"1101";
788     wait for 20 ns;
        w_A <= B"1110";
790     wait for 20 ns;
        w_A <= B"1111";
792     wait for 20 ns;
        w_B <= B"0111";
794     w_A <= B"0000";
        wait for 20 ns;
796     w_A <= B"0001";
        wait for 20 ns;
798     w_A <= B"0010";
        wait for 20 ns;
800     w_A <= B"0011";
        wait for 20 ns;
802     w_A <= B"0100";
        wait for 20 ns;
804     w_A <= B"0101";
        wait for 20 ns;
806     w_A <= B"0110";
        wait for 20 ns;
808     w_A <= B"0111";
        wait for 20 ns;
810     w_A <= B"1000";
        wait for 20 ns;
812     w_A <= B"1001";
        wait for 20 ns;
```

```
814         w_A <= B"1010";
            wait for 20 ns;
816         w_A <= B"1011";
            wait for 20 ns;
818         w_A <= B"1100";
            wait for 20 ns;
820         w_A <= B"1101";
            wait for 20 ns;
822         w_A <= B"1110";
            wait for 20 ns;
824         w_A <= B"1111";
            wait for 20 ns;
826         w_B <= B"1000";
            w_A <= B"0000";
828         wait for 20 ns;
            w_A <= B"0001";
830         wait for 20 ns;
            w_A <= B"0010";
832         wait for 20 ns;
            w_A <= B"0011";
834         wait for 20 ns;
            w_A <= B"0100";
836         wait for 20 ns;
            w_A <= B"0101";
838         wait for 20 ns;
            w_A <= B"0110";
840         wait for 20 ns;
            w_A <= B"0111";
842         wait for 20 ns;
            w_A <= B"1000";
844         wait for 20 ns;
            w_A <= B"1001";
846         wait for 20 ns;
            w_A <= B"1010";
848         wait for 20 ns;
            w_A <= B"1011";
850         wait for 20 ns;
            w_A <= B"1100";
852         wait for 20 ns;
            w_A <= B"1101";
854         wait for 20 ns;
            w_A <= B"1110";
856         wait for 20 ns;
```

```
        w_A <= B"1111";
858     wait for 20 ns;
        w_B <= B"1001";
860     w_A <= B"0000";
        wait for 20 ns;
862     w_A <= B"0001";
        wait for 20 ns;
864     w_A <= B"0010";
        wait for 20 ns;
866     w_A <= B"0011";
        wait for 20 ns;
868     w_A <= B"0100";
        wait for 20 ns;
870     w_A <= B"0101";
        wait for 20 ns;
872     w_A <= B"0110";
        wait for 20 ns;
874     w_A <= B"0111";
        wait for 20 ns;
876     w_A <= B"1000";
        wait for 20 ns;
878     w_A <= B"1001";
        wait for 20 ns;
880     w_A <= B"1010";
        wait for 20 ns;
882     w_A <= B"1011";
        wait for 20 ns;
884     w_A <= B"1100";
        wait for 20 ns;
886     w_A <= B"1101";
        wait for 20 ns;
888     w_A <= B"1110";
        wait for 20 ns;
890     w_A <= B"1111";
        wait for 20 ns;
892     w_B <= B"1010";
        w_A <= B"0000";
894     wait for 20 ns;
        w_A <= B"0001";
896     wait for 20 ns;
        w_A <= B"0010";
898     wait for 20 ns;
        w_A <= B"0011";
```

```
900         wait for 20 ns;
            w_A <= B"0100";
902         wait for 20 ns;
            w_A <= B"0101";
904         wait for 20 ns;
            w_A <= B"0110";
906         wait for 20 ns;
            w_A <= B"0111";
908         wait for 20 ns;
            w_A <= B"1000";
910         wait for 20 ns;
            w_A <= B"1001";
912         wait for 20 ns;
            w_A <= B"1010";
914         wait for 20 ns;
            w_A <= B"1011";
916         wait for 20 ns;
            w_A <= B"1100";
918         wait for 20 ns;
            w_A <= B"1101";
920         wait for 20 ns;
            w_A <= B"1110";
922         wait for 20 ns;
            w_A <= B"1111";
924         wait for 20 ns;
            w_B <= B"1011";
926         w_A <= B"0000";
            wait for 20 ns;
928         w_A <= B"0001";
            wait for 20 ns;
930         w_A <= B"0010";
            wait for 20 ns;
932         w_A <= B"0011";
            wait for 20 ns;
934         w_A <= B"0100";
            wait for 20 ns;
936         w_A <= B"0101";
            wait for 20 ns;
938         w_A <= B"0110";
            wait for 20 ns;
940         w_A <= B"0111";
            wait for 20 ns;
942         w_A <= B"1000";
```

```
        wait for 20 ns;
944     w_A <= B"1001";
        wait for 20 ns;
946     w_A <= B"1010";
        wait for 20 ns;
948     w_A <= B"1011";
        wait for 20 ns;
950     w_A <= B"1100";
        wait for 20 ns;
952     w_A <= B"1101";
        wait for 20 ns;
954     w_A <= B"1110";
        wait for 20 ns;
956     w_A <= B"1111";
        wait for 20 ns;
958     w_B <= B"1100";
        w_A <= B"0000";
960     wait for 20 ns;
        w_A <= B"0001";
962     wait for 20 ns;
        w_A <= B"0010";
964     wait for 20 ns;
        w_A <= B"0011";
966     wait for 20 ns;
        w_A <= B"0100";
968     wait for 20 ns;
        w_A <= B"0101";
970     wait for 20 ns;
        w_A <= B"0110";
972     wait for 20 ns;
        w_A <= B"0111";
974     wait for 20 ns;
        w_A <= B"1000";
976     wait for 20 ns;
        w_A <= B"1001";
978     wait for 20 ns;
        w_A <= B"1010";
980     wait for 20 ns;
        w_A <= B"1011";
982     wait for 20 ns;
        w_A <= B"1100";
984     wait for 20 ns;
        w_A <= B"1101";
```

```
986         wait for 20 ns;
            w_A <= B"1110";
988         wait for 20 ns;
            w_A <= B"1111";
990         wait for 20 ns;
            w_B <= B"1101";
992         w_A <= B"0000";
            wait for 20 ns;
994         w_A <= B"0001";
            wait for 20 ns;
996         w_A <= B"0010";
            wait for 20 ns;
998         w_A <= B"0011";
            wait for 20 ns;
1000        w_A <= B"0100";
            wait for 20 ns;
1002        w_A <= B"0101";
            wait for 20 ns;
1004        w_A <= B"0110";
            wait for 20 ns;
1006        w_A <= B"0111";
            wait for 20 ns;
1008        w_A <= B"1000";
            wait for 20 ns;
1010        w_A <= B"1001";
            wait for 20 ns;
1012        w_A <= B"1010";
            wait for 20 ns;
1014        w_A <= B"1011";
            wait for 20 ns;
1016        w_A <= B"1100";
            wait for 20 ns;
1018        w_A <= B"1101";
            wait for 20 ns;
1020        w_A <= B"1110";
            wait for 20 ns;
1022        w_A <= B"1111";
            wait for 20 ns;
1024        w_B <= B"1110";
            w_A <= B"0000";
1026        wait for 20 ns;
            w_A <= B"0001";
1028        wait for 20 ns;
```

```
        w_A <= B"0010";
1030    wait for 20 ns;
        w_A <= B"0011";
1032    wait for 20 ns;
        w_A <= B"0100";
1034    wait for 20 ns;
        w_A <= B"0101";
1036    wait for 20 ns;
        w_A <= B"0110";
1038    wait for 20 ns;
        w_A <= B"0111";
1040    wait for 20 ns;
        w_A <= B"1000";
1042    wait for 20 ns;
        w_A <= B"1001";
1044    wait for 20 ns;
        w_A <= B"1010";
1046    wait for 20 ns;
        w_A <= B"1011";
1048    wait for 20 ns;
        w_A <= B"1100";
1050    wait for 20 ns;
        w_A <= B"1101";
1052    wait for 20 ns;
        w_A <= B"1110";
1054    wait for 20 ns;
        w_A <= B"1111";
1056    wait for 20 ns;
        w_B <= B"1111";
1058    w_A <= B"0000";
        wait for 20 ns;
1060    w_A <= B"0001";
        wait for 20 ns;
1062    w_A <= B"0010";
        wait for 20 ns;
1064    w_A <= B"0011";
        wait for 20 ns;
1066    w_A <= B"0100";
        wait for 20 ns;
1068    w_A <= B"0101";
        wait for 20 ns;
1070    w_A <= B"0110";
        wait for 20 ns;
```

```
1072        w_A <= B"0111";
            wait for 20 ns;
1074        w_A <= B"1000";
            wait for 20 ns;
1076        w_A <= B"1001";
            wait for 20 ns;
1078        w_A <= B"1010";
            wait for 20 ns;
1080        w_A <= B"1011";
            wait for 20 ns;
1082        w_A <= B"1100";
            wait for 20 ns;
1084        w_A <= B"1101";
            wait for 20 ns;
1086        w_A <= B"1110";
            wait for 20 ns;
1088        w_A <= B"1111";
            wait for 20 ns;
1090        wait;
        end process stimulus;
1092 end testbench;
```

Listing B.25. FULL_ADDER_4_SC_TB.vhd Code

2.8.8 Self-Contained 2-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_2_SC_TB
--| Tests CARRY_SELECT_ADDER_2_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_2_SC_TB is
end CARRY_SELECT_ADDER_2_SC_TB;

architecture testbench of CARRY_SELECT_ADDER_2_SC_TB is
  component CARRY_SELECT_ADDER_2_SC is
    port (i_A     : in  std_logic_vector(1 downto 0);
          i_B     : in  std_logic_vector(1 downto 0);
          i_C     : in  std_logic;
          o_S     : out std_logic_vector(1 downto 0);
          o_C     : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(1 downto 0) := (others =>'0');
  signal w_B : std_logic_vector(1 downto 0) := (others =>'0');
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(1 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect CARRY_SELECT_ADDER_2_SC
  u_CARRY_SELECT_ADDER_2_SC: CARRY_SELECT_ADDER_2_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"01";
    wait for 20 ns;
    w_A <= B"10";
    wait for 20 ns;
```

```
40      w_A <= B"11";
        wait for 20 ns;
42      w_A <= B"00";
        w_B <= B"01";
44      wait for 20 ns;
        w_A <= B"01";
46      wait for 20 ns;
        w_A <= B"10";
48      wait for 20 ns;
        w_A <= B"11";
50      wait for 20 ns;
        w_A <= B"00";
52      w_B <= B"10";
        wait for 20 ns;
54      w_A <= B"01";
        wait for 20 ns;
56      w_A <= B"10";
        wait for 20 ns;
58      w_A <= B"11";
        wait for 20 ns;
60      w_A <= B"00";
        w_B <= B"11";
62      wait for 20 ns;
        w_A <= B"01";
64      wait for 20 ns;
        w_A <= B"10";
66      wait for 20 ns;
        w_A <= B"11";
68      wait for 20 ns;
        w_A <= B"00";
70      w_B <= B"00";
        w_C <= '1';
72      wait for 20 ns;
        w_A <= B"01";
74      wait for 20 ns;
        w_A <= B"10";
76      wait for 20 ns;
        w_A <= B"11";
78      wait for 20 ns;
        w_A <= B"00";
80      w_B <= B"01";
        wait for 20 ns;
82      w_A <= B"01";
```

```
    wait for 20 ns;
    w_A <= B"10";
    wait for 20 ns;
    w_A <= B"11";
    wait for 20 ns;
    w_A <= B"00";
    w_B <= B"10";
    wait for 20 ns;
    w_A <= B"01";
    wait for 20 ns;
    w_A <= B"10";
    wait for 20 ns;
    w_A <= B"11";
    wait for 20 ns;
    w_A <= B"00";
    w_B <= B"11";
    wait for 20 ns;
    w_A <= B"01";
    wait for 20 ns;
    w_A <= B"10";
    wait for 20 ns;
    w_A <= B"11";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.26. CARRY_SELECT_ADDER_2_SC_TB.vhd Code

2.8.9 Self-Contained 4-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_4_SC_TB
--| Tests CARRY_SELECT_ADDER_4_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_4_SC_TB is
end CARRY_SELECT_ADDER_4_SC_TB;

architecture testbench of CARRY_SELECT_ADDER_4_SC_TB is
  component CARRY_SELECT_ADDER_4_SC is
    port (i_A      : in  std_logic_vector(3 downto 0);
          i_B      : in  std_logic_vector(3 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(3 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(3 downto 0) := (others =>'0');
  signal w_B : std_logic_vector(3 downto 0) := (others =>'0');
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(3 downto 0);
  signal w_Cout  : std_logic;
begin
  -- Connect CARRY_SELECT_ADDER_4_SC
  u_CARRY_SELECT_ADDER_4_SC: CARRY_SELECT_ADDER_4_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"0001";
    wait for 20 ns;
    w_A <= B"0010";
    wait for 20 ns;
```

```
40      w_A <= B"0011";
        wait for 20 ns;
42      w_A <= B"0100";
        wait for 20 ns;
44      w_A <= B"0101";
        wait for 20 ns;
46      w_A <= B"0110";
        wait for 20 ns;
48      w_A <= B"0111";
        wait for 20 ns;
50      w_A <= B"1000";
        wait for 20 ns;
52      w_A <= B"1001";
        wait for 20 ns;
54      w_A <= B"1010";
        wait for 20 ns;
56      w_A <= B"1011";
        wait for 20 ns;
58      w_A <= B"1100";
        wait for 20 ns;
60      w_A <= B"1101";
        wait for 20 ns;
62      w_A <= B"1110";
        wait for 20 ns;
64      w_A <= B"1111";
        wait for 20 ns;
66      w_B <= B"0001";
        w_A <= B"0000";
68      wait for 20 ns;
        w_A <= B"0001";
70      wait for 20 ns;
        w_A <= B"0010";
72      wait for 20 ns;
        w_A <= B"0011";
74      wait for 20 ns;
        w_A <= B"0100";
76      wait for 20 ns;
        w_A <= B"0101";
78      wait for 20 ns;
        w_A <= B"0110";
80      wait for 20 ns;
        w_A <= B"0111";
82      wait for 20 ns;
```

```
        w_A <= B"1000";
84      wait for 20 ns;
        w_A <= B"1001";
86      wait for 20 ns;
        w_A <= B"1010";
88      wait for 20 ns;
        w_A <= B"1011";
90      wait for 20 ns;
        w_A <= B"1100";
92      wait for 20 ns;
        w_A <= B"1101";
94      wait for 20 ns;
        w_A <= B"1110";
96      wait for 20 ns;
        w_A <= B"1111";
98      wait for 20 ns;
        w_B <= B"0010";
100     w_A <= B"0000";
        wait for 20 ns;
102     w_A <= B"0001";
        wait for 20 ns;
104     w_A <= B"0010";
        wait for 20 ns;
106     w_A <= B"0011";
        wait for 20 ns;
108     w_A <= B"0100";
        wait for 20 ns;
110     w_A <= B"0101";
        wait for 20 ns;
112     w_A <= B"0110";
        wait for 20 ns;
114     w_A <= B"0111";
        wait for 20 ns;
116     w_A <= B"1000";
        wait for 20 ns;
118     w_A <= B"1001";
        wait for 20 ns;
120     w_A <= B"1010";
        wait for 20 ns;
122     w_A <= B"1011";
        wait for 20 ns;
124     w_A <= B"1100";
        wait for 20 ns;
```

```
126         w_A <= B"1101";
            wait for 20 ns;
128         w_A <= B"1110";
            wait for 20 ns;
130         w_A <= B"1111";
            wait for 20 ns;
132         w_B <= B"0011";
            w_A <= B"0000";
134         wait for 20 ns;
            w_A <= B"0001";
136         wait for 20 ns;
            w_A <= B"0010";
138         wait for 20 ns;
            w_A <= B"0011";
140         wait for 20 ns;
            w_A <= B"0100";
142         wait for 20 ns;
            w_A <= B"0101";
144         wait for 20 ns;
            w_A <= B"0110";
146         wait for 20 ns;
            w_A <= B"0111";
148         wait for 20 ns;
            w_A <= B"1000";
150         wait for 20 ns;
            w_A <= B"1001";
152         wait for 20 ns;
            w_A <= B"1010";
154         wait for 20 ns;
            w_A <= B"1011";
156         wait for 20 ns;
            w_A <= B"1100";
158         wait for 20 ns;
            w_A <= B"1101";
160         wait for 20 ns;
            w_A <= B"1110";
162         wait for 20 ns;
            w_A <= B"1111";
164         wait for 20 ns;
            w_B <= B"0100";
166         w_A <= B"0000";
            wait for 20 ns;
168         w_A <= B"0001";
```

```
        wait for 20 ns;
170     w_A <= B"0010";
        wait for 20 ns;
172     w_A <= B"0011";
        wait for 20 ns;
174     w_A <= B"0100";
        wait for 20 ns;
176     w_A <= B"0101";
        wait for 20 ns;
178     w_A <= B"0110";
        wait for 20 ns;
180     w_A <= B"0111";
        wait for 20 ns;
182     w_A <= B"1000";
        wait for 20 ns;
184     w_A <= B"1001";
        wait for 20 ns;
186     w_A <= B"1010";
        wait for 20 ns;
188     w_A <= B"1011";
        wait for 20 ns;
190     w_A <= B"1100";
        wait for 20 ns;
192     w_A <= B"1101";
        wait for 20 ns;
194     w_A <= B"1110";
        wait for 20 ns;
196     w_A <= B"1111";
        wait for 20 ns;
198     w_B <= B"0101";
        w_A <= B"0000";
200     wait for 20 ns;
        w_A <= B"0001";
202     wait for 20 ns;
        w_A <= B"0010";
204     wait for 20 ns;
        w_A <= B"0011";
206     wait for 20 ns;
        w_A <= B"0100";
208     wait for 20 ns;
        w_A <= B"0101";
210     wait for 20 ns;
        w_A <= B"0110";
```

```
212     wait for 20 ns;
        w_A <= B"0111";
214     wait for 20 ns;
        w_A <= B"1000";
216     wait for 20 ns;
        w_A <= B"1001";
218     wait for 20 ns;
        w_A <= B"1010";
220     wait for 20 ns;
        w_A <= B"1011";
222     wait for 20 ns;
        w_A <= B"1100";
224     wait for 20 ns;
        w_A <= B"1101";
226     wait for 20 ns;
        w_A <= B"1110";
228     wait for 20 ns;
        w_A <= B"1111";
230     wait for 20 ns;
        w_B <= B"0110";
232     w_A <= B"0000";
        wait for 20 ns;
234     w_A <= B"0001";
        wait for 20 ns;
236     w_A <= B"0010";
        wait for 20 ns;
238     w_A <= B"0011";
        wait for 20 ns;
240     w_A <= B"0100";
        wait for 20 ns;
242     w_A <= B"0101";
        wait for 20 ns;
244     w_A <= B"0110";
        wait for 20 ns;
246     w_A <= B"0111";
        wait for 20 ns;
248     w_A <= B"1000";
        wait for 20 ns;
250     w_A <= B"1001";
        wait for 20 ns;
252     w_A <= B"1010";
        wait for 20 ns;
254     w_A <= B"1011";
```

```
        wait for 20 ns;
256     w_A <= B"1100";
        wait for 20 ns;
258     w_A <= B"1101";
        wait for 20 ns;
260     w_A <= B"1110";
        wait for 20 ns;
262     w_A <= B"1111";
        wait for 20 ns;
264     w_B <= B"0111";
        w_A <= B"0000";
266     wait for 20 ns;
        w_A <= B"0001";
268     wait for 20 ns;
        w_A <= B"0010";
270     wait for 20 ns;
        w_A <= B"0011";
272     wait for 20 ns;
        w_A <= B"0100";
274     wait for 20 ns;
        w_A <= B"0101";
276     wait for 20 ns;
        w_A <= B"0110";
278     wait for 20 ns;
        w_A <= B"0111";
280     wait for 20 ns;
        w_A <= B"1000";
282     wait for 20 ns;
        w_A <= B"1001";
284     wait for 20 ns;
        w_A <= B"1010";
286     wait for 20 ns;
        w_A <= B"1011";
288     wait for 20 ns;
        w_A <= B"1100";
290     wait for 20 ns;
        w_A <= B"1101";
292     wait for 20 ns;
        w_A <= B"1110";
294     wait for 20 ns;
        w_A <= B"1111";
296     wait for 20 ns;
        w_B <= B"1000";
```

```
298         w_A <= B"0000";
            wait for 20 ns;
300         w_A <= B"0001";
            wait for 20 ns;
302         w_A <= B"0010";
            wait for 20 ns;
304         w_A <= B"0011";
            wait for 20 ns;
306         w_A <= B"0100";
            wait for 20 ns;
308         w_A <= B"0101";
            wait for 20 ns;
310         w_A <= B"0110";
            wait for 20 ns;
312         w_A <= B"0111";
            wait for 20 ns;
314         w_A <= B"1000";
            wait for 20 ns;
316         w_A <= B"1001";
            wait for 20 ns;
318         w_A <= B"1010";
            wait for 20 ns;
320         w_A <= B"1011";
            wait for 20 ns;
322         w_A <= B"1100";
            wait for 20 ns;
324         w_A <= B"1101";
            wait for 20 ns;
326         w_A <= B"1110";
            wait for 20 ns;
328         w_A <= B"1111";
            wait for 20 ns;
330         w_B <= B"1001";
            w_A <= B"0000";
332         wait for 20 ns;
            w_A <= B"0001";
334         wait for 20 ns;
            w_A <= B"0010";
336         wait for 20 ns;
            w_A <= B"0011";
338         wait for 20 ns;
            w_A <= B"0100";
340         wait for 20 ns;
```

```
        w_A <= B"0101";
342     wait for 20 ns;
        w_A <= B"0110";
344     wait for 20 ns;
        w_A <= B"0111";
346     wait for 20 ns;
        w_A <= B"1000";
348     wait for 20 ns;
        w_A <= B"1001";
350     wait for 20 ns;
        w_A <= B"1010";
352     wait for 20 ns;
        w_A <= B"1011";
354     wait for 20 ns;
        w_A <= B"1100";
356     wait for 20 ns;
        w_A <= B"1101";
358     wait for 20 ns;
        w_A <= B"1110";
360     wait for 20 ns;
        w_A <= B"1111";
362     wait for 20 ns;
        w_B <= B"1010";
364     w_A <= B"0000";
        wait for 20 ns;
366     w_A <= B"0001";
        wait for 20 ns;
368     w_A <= B"0010";
        wait for 20 ns;
370     w_A <= B"0011";
        wait for 20 ns;
372     w_A <= B"0100";
        wait for 20 ns;
374     w_A <= B"0101";
        wait for 20 ns;
376     w_A <= B"0110";
        wait for 20 ns;
378     w_A <= B"0111";
        wait for 20 ns;
380     w_A <= B"1000";
        wait for 20 ns;
382     w_A <= B"1001";
        wait for 20 ns;
```

```
384         w_A <= B"1010";
            wait for 20 ns;
386         w_A <= B"1011";
            wait for 20 ns;
388         w_A <= B"1100";
            wait for 20 ns;
390         w_A <= B"1101";
            wait for 20 ns;
392         w_A <= B"1110";
            wait for 20 ns;
394         w_A <= B"1111";
            wait for 20 ns;
396         w_B <= B"1011";
            w_A <= B"0000";
398         wait for 20 ns;
            w_A <= B"0001";
400         wait for 20 ns;
            w_A <= B"0010";
402         wait for 20 ns;
            w_A <= B"0011";
404         wait for 20 ns;
            w_A <= B"0100";
406         wait for 20 ns;
            w_A <= B"0101";
408         wait for 20 ns;
            w_A <= B"0110";
410         wait for 20 ns;
            w_A <= B"0111";
412         wait for 20 ns;
            w_A <= B"1000";
414         wait for 20 ns;
            w_A <= B"1001";
416         wait for 20 ns;
            w_A <= B"1010";
418         wait for 20 ns;
            w_A <= B"1011";
420         wait for 20 ns;
            w_A <= B"1100";
422         wait for 20 ns;
            w_A <= B"1101";
424         wait for 20 ns;
            w_A <= B"1110";
426         wait for 20 ns;
```

```
        w_A <= B"1111";
428     wait for 20 ns;
        w_B <= B"1100";
430     w_A <= B"0000";
        wait for 20 ns;
432     w_A <= B"0001";
        wait for 20 ns;
434     w_A <= B"0010";
        wait for 20 ns;
436     w_A <= B"0011";
        wait for 20 ns;
438     w_A <= B"0100";
        wait for 20 ns;
440     w_A <= B"0101";
        wait for 20 ns;
442     w_A <= B"0110";
        wait for 20 ns;
444     w_A <= B"0111";
        wait for 20 ns;
446     w_A <= B"1000";
        wait for 20 ns;
448     w_A <= B"1001";
        wait for 20 ns;
450     w_A <= B"1010";
        wait for 20 ns;
452     w_A <= B"1011";
        wait for 20 ns;
454     w_A <= B"1100";
        wait for 20 ns;
456     w_A <= B"1101";
        wait for 20 ns;
458     w_A <= B"1110";
        wait for 20 ns;
460     w_A <= B"1111";
        wait for 20 ns;
462     w_B <= B"1101";
        w_A <= B"0000";
464     wait for 20 ns;
        w_A <= B"0001";
466     wait for 20 ns;
        w_A <= B"0010";
468     wait for 20 ns;
        w_A <= B"0011";
```

```
470         wait for 20 ns;
            w_A <= B"0100";
472         wait for 20 ns;
            w_A <= B"0101";
474         wait for 20 ns;
            w_A <= B"0110";
476         wait for 20 ns;
            w_A <= B"0111";
478         wait for 20 ns;
            w_A <= B"1000";
480         wait for 20 ns;
            w_A <= B"1001";
482         wait for 20 ns;
            w_A <= B"1010";
484         wait for 20 ns;
            w_A <= B"1011";
486         wait for 20 ns;
            w_A <= B"1100";
488         wait for 20 ns;
            w_A <= B"1101";
490         wait for 20 ns;
            w_A <= B"1110";
492         wait for 20 ns;
            w_A <= B"1111";
494         wait for 20 ns;
            w_B <= B"1110";
496         w_A <= B"0000";
            wait for 20 ns;
498         w_A <= B"0001";
            wait for 20 ns;
500         w_A <= B"0010";
            wait for 20 ns;
502         w_A <= B"0011";
            wait for 20 ns;
504         w_A <= B"0100";
            wait for 20 ns;
506         w_A <= B"0101";
            wait for 20 ns;
508         w_A <= B"0110";
            wait for 20 ns;
510         w_A <= B"0111";
            wait for 20 ns;
512         w_A <= B"1000";
```

```
        wait for 20 ns;
514     w_A <= B"1001";
        wait for 20 ns;
516     w_A <= B"1010";
        wait for 20 ns;
518     w_A <= B"1011";
        wait for 20 ns;
520     w_A <= B"1100";
        wait for 20 ns;
522     w_A <= B"1101";
        wait for 20 ns;
524     w_A <= B"1110";
        wait for 20 ns;
526     w_A <= B"1111";
        wait for 20 ns;
528     w_B <= B"1111";
        w_A <= B"0000";
530     wait for 20 ns;
        w_A <= B"0001";
532     wait for 20 ns;
        w_A <= B"0010";
534     wait for 20 ns;
        w_A <= B"0011";
536     wait for 20 ns;
        w_A <= B"0100";
538     wait for 20 ns;
        w_A <= B"0101";
540     wait for 20 ns;
        w_A <= B"0110";
542     wait for 20 ns;
        w_A <= B"0111";
544     wait for 20 ns;
        w_A <= B"1000";
546     wait for 20 ns;
        w_A <= B"1001";
548     wait for 20 ns;
        w_A <= B"1010";
550     wait for 20 ns;
        w_A <= B"1011";
552     wait for 20 ns;
        w_A <= B"1100";
554     wait for 20 ns;
        w_A <= B"1101";
```

```
556         wait for 20 ns;
            w_A <= B"1110";
558         wait for 20 ns;
            w_A <= B"1111";
560         wait for 20 ns;
            w_c <= '1';
562         w_B <= B"0000";
            w_A <= B"0000";
564         wait for 20 ns;
            w_A <= B"0001";
566         wait for 20 ns;
            w_A <= B"0010";
568         wait for 20 ns;
            w_A <= B"0011";
570         wait for 20 ns;
            w_A <= B"0100";
572         wait for 20 ns;
            w_A <= B"0101";
574         wait for 20 ns;
            w_A <= B"0110";
576         wait for 20 ns;
            w_A <= B"0111";
578         wait for 20 ns;
            w_A <= B"1000";
580         wait for 20 ns;
            w_A <= B"1001";
582         wait for 20 ns;
            w_A <= B"1010";
584         wait for 20 ns;
            w_A <= B"1011";
586         wait for 20 ns;
            w_A <= B"1100";
588         wait for 20 ns;
            w_A <= B"1101";
590         wait for 20 ns;
            w_A <= B"1110";
592         wait for 20 ns;
            w_A <= B"1111";
594         wait for 20 ns;
            w_B <= B"0001";
596         w_A <= B"0000";
            wait for 20 ns;
598         w_A <= B"0001";
```

```
        wait for 20 ns;
600     w_A <= B"0010";
        wait for 20 ns;
602     w_A <= B"0011";
        wait for 20 ns;
604     w_A <= B"0100";
        wait for 20 ns;
606     w_A <= B"0101";
        wait for 20 ns;
608     w_A <= B"0110";
        wait for 20 ns;
610     w_A <= B"0111";
        wait for 20 ns;
612     w_A <= B"1000";
        wait for 20 ns;
614     w_A <= B"1001";
        wait for 20 ns;
616     w_A <= B"1010";
        wait for 20 ns;
618     w_A <= B"1011";
        wait for 20 ns;
620     w_A <= B"1100";
        wait for 20 ns;
622     w_A <= B"1101";
        wait for 20 ns;
624     w_A <= B"1110";
        wait for 20 ns;
626     w_A <= B"1111";
        wait for 20 ns;
628     w_B <= B"0010";
        w_A <= B"0000";
630     wait for 20 ns;
        w_A <= B"0001";
632     wait for 20 ns;
        w_A <= B"0010";
634     wait for 20 ns;
        w_A <= B"0011";
636     wait for 20 ns;
        w_A <= B"0100";
638     wait for 20 ns;
        w_A <= B"0101";
640     wait for 20 ns;
        w_A <= B"0110";
```

```
642         wait for 20 ns;
            w_A <= B"0111";
644         wait for 20 ns;
            w_A <= B"1000";
646         wait for 20 ns;
            w_A <= B"1001";
648         wait for 20 ns;
            w_A <= B"1010";
650         wait for 20 ns;
            w_A <= B"1011";
652         wait for 20 ns;
            w_A <= B"1100";
654         wait for 20 ns;
            w_A <= B"1101";
656         wait for 20 ns;
            w_A <= B"1110";
658         wait for 20 ns;
            w_A <= B"1111";
660         wait for 20 ns;
            w_B <= B"0011";
662         w_A <= B"0000";
            wait for 20 ns;
664         w_A <= B"0001";
            wait for 20 ns;
666         w_A <= B"0010";
            wait for 20 ns;
668         w_A <= B"0011";
            wait for 20 ns;
670         w_A <= B"0100";
            wait for 20 ns;
672         w_A <= B"0101";
            wait for 20 ns;
674         w_A <= B"0110";
            wait for 20 ns;
676         w_A <= B"0111";
            wait for 20 ns;
678         w_A <= B"1000";
            wait for 20 ns;
680         w_A <= B"1001";
            wait for 20 ns;
682         w_A <= B"1010";
            wait for 20 ns;
684         w_A <= B"1011";
```

```
        wait for 20 ns;
686     w_A <= B"1100";
        wait for 20 ns;
688     w_A <= B"1101";
        wait for 20 ns;
690     w_A <= B"1110";
        wait for 20 ns;
692     w_A <= B"1111";
        wait for 20 ns;
694     w_B <= B"0100";
        w_A <= B"0000";
696     wait for 20 ns;
        w_A <= B"0001";
698     wait for 20 ns;
        w_A <= B"0010";
700     wait for 20 ns;
        w_A <= B"0011";
702     wait for 20 ns;
        w_A <= B"0100";
704     wait for 20 ns;
        w_A <= B"0101";
706     wait for 20 ns;
        w_A <= B"0110";
708     wait for 20 ns;
        w_A <= B"0111";
710     wait for 20 ns;
        w_A <= B"1000";
712     wait for 20 ns;
        w_A <= B"1001";
714     wait for 20 ns;
        w_A <= B"1010";
716     wait for 20 ns;
        w_A <= B"1011";
718     wait for 20 ns;
        w_A <= B"1100";
720     wait for 20 ns;
        w_A <= B"1101";
722     wait for 20 ns;
        w_A <= B"1110";
724     wait for 20 ns;
        w_A <= B"1111";
726     wait for 20 ns;
        w_B <= B"0101";
```

```
728         w_A <= B"0000";
            wait for 20 ns;
730         w_A <= B"0001";
            wait for 20 ns;
732         w_A <= B"0010";
            wait for 20 ns;
734         w_A <= B"0011";
            wait for 20 ns;
736         w_A <= B"0100";
            wait for 20 ns;
738         w_A <= B"0101";
            wait for 20 ns;
740         w_A <= B"0110";
            wait for 20 ns;
742         w_A <= B"0111";
            wait for 20 ns;
744         w_A <= B"1000";
            wait for 20 ns;
746         w_A <= B"1001";
            wait for 20 ns;
748         w_A <= B"1010";
            wait for 20 ns;
750         w_A <= B"1011";
            wait for 20 ns;
752         w_A <= B"1100";
            wait for 20 ns;
754         w_A <= B"1101";
            wait for 20 ns;
756         w_A <= B"1110";
            wait for 20 ns;
758         w_A <= B"1111";
            wait for 20 ns;
760         w_B <= B"0110";
            w_A <= B"0000";
762         wait for 20 ns;
            w_A <= B"0001";
764         wait for 20 ns;
            w_A <= B"0010";
766         wait for 20 ns;
            w_A <= B"0011";
768         wait for 20 ns;
            w_A <= B"0100";
770         wait for 20 ns;
```

```
        w_A <= B"0101";
772     wait for 20 ns;
        w_A <= B"0110";
774     wait for 20 ns;
        w_A <= B"0111";
776     wait for 20 ns;
        w_A <= B"1000";
778     wait for 20 ns;
        w_A <= B"1001";
780     wait for 20 ns;
        w_A <= B"1010";
782     wait for 20 ns;
        w_A <= B"1011";
784     wait for 20 ns;
        w_A <= B"1100";
786     wait for 20 ns;
        w_A <= B"1101";
788     wait for 20 ns;
        w_A <= B"1110";
790     wait for 20 ns;
        w_A <= B"1111";
792     wait for 20 ns;
        w_B <= B"0111";
794     w_A <= B"0000";
        wait for 20 ns;
796     w_A <= B"0001";
        wait for 20 ns;
798     w_A <= B"0010";
        wait for 20 ns;
800     w_A <= B"0011";
        wait for 20 ns;
802     w_A <= B"0100";
        wait for 20 ns;
804     w_A <= B"0101";
        wait for 20 ns;
806     w_A <= B"0110";
        wait for 20 ns;
808     w_A <= B"0111";
        wait for 20 ns;
810     w_A <= B"1000";
        wait for 20 ns;
812     w_A <= B"1001";
        wait for 20 ns;
```

```
814     w_A <= B"1010";
        wait for 20 ns;
816     w_A <= B"1011";
        wait for 20 ns;
818     w_A <= B"1100";
        wait for 20 ns;
820     w_A <= B"1101";
        wait for 20 ns;
822     w_A <= B"1110";
        wait for 20 ns;
824     w_A <= B"1111";
        wait for 20 ns;
826     w_B <= B"1000";
        w_A <= B"0000";
828     wait for 20 ns;
        w_A <= B"0001";
830     wait for 20 ns;
        w_A <= B"0010";
832     wait for 20 ns;
        w_A <= B"0011";
834     wait for 20 ns;
        w_A <= B"0100";
836     wait for 20 ns;
        w_A <= B"0101";
838     wait for 20 ns;
        w_A <= B"0110";
840     wait for 20 ns;
        w_A <= B"0111";
842     wait for 20 ns;
        w_A <= B"1000";
844     wait for 20 ns;
        w_A <= B"1001";
846     wait for 20 ns;
        w_A <= B"1010";
848     wait for 20 ns;
        w_A <= B"1011";
850     wait for 20 ns;
        w_A <= B"1100";
852     wait for 20 ns;
        w_A <= B"1101";
854     wait for 20 ns;
        w_A <= B"1110";
856     wait for 20 ns;
```

```
        w_A <= B"1111";
858     wait for 20 ns;
        w_B <= B"1001";
860     w_A <= B"0000";
        wait for 20 ns;
862     w_A <= B"0001";
        wait for 20 ns;
864     w_A <= B"0010";
        wait for 20 ns;
866     w_A <= B"0011";
        wait for 20 ns;
868     w_A <= B"0100";
        wait for 20 ns;
870     w_A <= B"0101";
        wait for 20 ns;
872     w_A <= B"0110";
        wait for 20 ns;
874     w_A <= B"0111";
        wait for 20 ns;
876     w_A <= B"1000";
        wait for 20 ns;
878     w_A <= B"1001";
        wait for 20 ns;
880     w_A <= B"1010";
        wait for 20 ns;
882     w_A <= B"1011";
        wait for 20 ns;
884     w_A <= B"1100";
        wait for 20 ns;
886     w_A <= B"1101";
        wait for 20 ns;
888     w_A <= B"1110";
        wait for 20 ns;
890     w_A <= B"1111";
        wait for 20 ns;
892     w_B <= B"1010";
        w_A <= B"0000";
894     wait for 20 ns;
        w_A <= B"0001";
896     wait for 20 ns;
        w_A <= B"0010";
898     wait for 20 ns;
        w_A <= B"0011";
```

```
900         wait for 20 ns;
            w_A <= B"0100";
902         wait for 20 ns;
            w_A <= B"0101";
904         wait for 20 ns;
            w_A <= B"0110";
906         wait for 20 ns;
            w_A <= B"0111";
908         wait for 20 ns;
            w_A <= B"1000";
910         wait for 20 ns;
            w_A <= B"1001";
912         wait for 20 ns;
            w_A <= B"1010";
914         wait for 20 ns;
            w_A <= B"1011";
916         wait for 20 ns;
            w_A <= B"1100";
918         wait for 20 ns;
            w_A <= B"1101";
920         wait for 20 ns;
            w_A <= B"1110";
922         wait for 20 ns;
            w_A <= B"1111";
924         wait for 20 ns;
            w_B <= B"1011";
926         w_A <= B"0000";
            wait for 20 ns;
928         w_A <= B"0001";
            wait for 20 ns;
930         w_A <= B"0010";
            wait for 20 ns;
932         w_A <= B"0011";
            wait for 20 ns;
934         w_A <= B"0100";
            wait for 20 ns;
936         w_A <= B"0101";
            wait for 20 ns;
938         w_A <= B"0110";
            wait for 20 ns;
940         w_A <= B"0111";
            wait for 20 ns;
942         w_A <= B"1000";
```

```
        wait for 20 ns;
944     w_A <= B"1001";
        wait for 20 ns;
946     w_A <= B"1010";
        wait for 20 ns;
948     w_A <= B"1011";
        wait for 20 ns;
950     w_A <= B"1100";
        wait for 20 ns;
952     w_A <= B"1101";
        wait for 20 ns;
954     w_A <= B"1110";
        wait for 20 ns;
956     w_A <= B"1111";
        wait for 20 ns;
958     w_B <= B"1100";
        w_A <= B"0000";
960     wait for 20 ns;
        w_A <= B"0001";
962     wait for 20 ns;
        w_A <= B"0010";
964     wait for 20 ns;
        w_A <= B"0011";
966     wait for 20 ns;
        w_A <= B"0100";
968     wait for 20 ns;
        w_A <= B"0101";
970     wait for 20 ns;
        w_A <= B"0110";
972     wait for 20 ns;
        w_A <= B"0111";
974     wait for 20 ns;
        w_A <= B"1000";
976     wait for 20 ns;
        w_A <= B"1001";
978     wait for 20 ns;
        w_A <= B"1010";
980     wait for 20 ns;
        w_A <= B"1011";
982     wait for 20 ns;
        w_A <= B"1100";
984     wait for 20 ns;
        w_A <= B"1101";
```

```
986         wait for 20 ns;
            w_A <= B"1110";
988         wait for 20 ns;
            w_A <= B"1111";
990         wait for 20 ns;
            w_B <= B"1101";
992         w_A <= B"0000";
            wait for 20 ns;
994         w_A <= B"0001";
            wait for 20 ns;
996         w_A <= B"0010";
            wait for 20 ns;
998         w_A <= B"0011";
            wait for 20 ns;
1000        w_A <= B"0100";
            wait for 20 ns;
1002        w_A <= B"0101";
            wait for 20 ns;
1004        w_A <= B"0110";
            wait for 20 ns;
1006        w_A <= B"0111";
            wait for 20 ns;
1008        w_A <= B"1000";
            wait for 20 ns;
1010        w_A <= B"1001";
            wait for 20 ns;
1012        w_A <= B"1010";
            wait for 20 ns;
1014        w_A <= B"1011";
            wait for 20 ns;
1016        w_A <= B"1100";
            wait for 20 ns;
1018        w_A <= B"1101";
            wait for 20 ns;
1020        w_A <= B"1110";
            wait for 20 ns;
1022        w_A <= B"1111";
            wait for 20 ns;
1024        w_B <= B"1110";
            w_A <= B"0000";
1026        wait for 20 ns;
            w_A <= B"0001";
1028        wait for 20 ns;
```

```
        w_A <= B"0010";
1030    wait for 20 ns;
        w_A <= B"0011";
1032    wait for 20 ns;
        w_A <= B"0100";
1034    wait for 20 ns;
        w_A <= B"0101";
1036    wait for 20 ns;
        w_A <= B"0110";
1038    wait for 20 ns;
        w_A <= B"0111";
1040    wait for 20 ns;
        w_A <= B"1000";
1042    wait for 20 ns;
        w_A <= B"1001";
1044    wait for 20 ns;
        w_A <= B"1010";
1046    wait for 20 ns;
        w_A <= B"1011";
1048    wait for 20 ns;
        w_A <= B"1100";
1050    wait for 20 ns;
        w_A <= B"1101";
1052    wait for 20 ns;
        w_A <= B"1110";
1054    wait for 20 ns;
        w_A <= B"1111";
1056    wait for 20 ns;
        w_B <= B"1111";
1058    w_A <= B"0000";
        wait for 20 ns;
1060    w_A <= B"0001";
        wait for 20 ns;
1062    w_A <= B"0010";
        wait for 20 ns;
1064    w_A <= B"0011";
        wait for 20 ns;
1066    w_A <= B"0100";
        wait for 20 ns;
1068    w_A <= B"0101";
        wait for 20 ns;
1070    w_A <= B"0110";
        wait for 20 ns;
```

```
1072        w_A <= B"0111";
            wait for 20 ns;
1074        w_A <= B"1000";
            wait for 20 ns;
1076        w_A <= B"1001";
            wait for 20 ns;
1078        w_A <= B"1010";
            wait for 20 ns;
1080        w_A <= B"1011";
            wait for 20 ns;
1082        w_A <= B"1100";
            wait for 20 ns;
1084        w_A <= B"1101";
            wait for 20 ns;
1086        w_A <= B"1110";
            wait for 20 ns;
1088        w_A <= B"1111";
            wait for 20 ns;
1090        wait;
        end process stimulus;
1092    end testbench;
```

Listing B.27. CARRY_SELECT_ADDER_4_SC_TB.vhd Code

2.8.10 Self-Contained 30-bit Carry Select Adder

```
--| CARRY_SELECT_ADDER_30_SC_TB
--| Tests CARRY_SELECT_ADDER_30_SC
library IEEE;
use IEEE.std_logic_1164.all;

entity CARRY_SELECT_ADDER_30_SC_TB is
end CARRY_SELECT_ADDER_30_SC_TB;

architecture testbench of CARRY_SELECT_ADDER_30_SC_TB is
  component CARRY_SELECT_ADDER_30_SC is
    port (i_A      : in  std_logic_vector(29 downto 0);
          i_B      : in  std_logic_vector(29 downto 0);
          i_C      : in  std_logic;
          o_S      : out std_logic_vector(29 downto 0);
          o_C      : out std_logic);
  end component;
  -- Declare signals
  signal w_A : std_logic_vector(29 downto 0) := (others
   =>'0');
  signal w_B : std_logic_vector(29 downto 0) := (others
   =>'0');
  signal w_C : std_logic := '0';
  signal w_S : std_logic_vector(29 downto 0);
  signal w_Cout  : std_logic;

begin
  -- Connect CARRY_SELECT_ADDER_30_SC
  u_CARRY_SELECT_ADDER_30_SC: CARRY_SELECT_ADDER_30_SC
  port map (i_A => w_A,
            i_B => w_B,
            i_C => w_C,
            o_S => w_S,
            o_C => w_Cout);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"00_0000_0000_0000_0000_0000_0000_1111";
    w_B <= B"00_0000_0000_0000_0000_0000_0000_0001";
    wait for 20 ns;
```

```
40      w_C <= '1';
        wait for 20 ns;
42      w_A <= B"11_1111_1111_1111_1111_1111_1111_1111";
        wait for 20 ns;
44      w_C <= '0';
        wait for 20 ns;
46      w_B <= B"00_0000_0000_0000_0000_0000_0000_1111";
        wait for 20 ns;
48      w_C <= '1';
        wait for 20 ns;
50      w_A <= B"00_0000_0000_0000_0000_0000_0000_1111";
        w_B <= B"00_0000_0000_0000_0000_0000_1111_0000";
52      wait;
    end process stimulus;
54  end testbench;
```

Listing B.28. CARRY_SELECT_ADDER_30_SC_TB.vhd Code

2.9 Load Upper Immediate

```
--| LUI_TB
--| Tests LUI
library IEEE;
use IEEE.std_logic_1164.all;

entity LUI_TB is
end LUI_TB;

architecture testbench of LUI_TB is
  component LUI is
    port (i_A : in  std_logic_vector(15 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_A : std_logic_vector(15 downto 0) := B"1111
   _1111_1111_1111";
  signal w_Z : std_logic_vector(31 downto 0);

begin
  -- Connect LUI
  u_LUI: LUI
  port map (i_A => w_A,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"0000_0000_1111_1111";
    wait for 20 ns;
    w_A <= B"1111_1111_0000_0000";
    wait for 20 ns;
    w_A <= B"0101_0101_0101_0101";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.29. LUI_TB.vhd Code

2.10 Overflow Detection

```
--| Over_Detect_TB.vhd
--| Test the functionality of the Overflow Detection
library IEEE;
use IEEE.std_logic_1164.all;

entity Over_Detect_TB is
end Over_Detect_TB;

architecture testbench of Over_Detect_TB is
--| Declare Components
  component Over_Detect is
    port (i_AB_NAND : in  std_logic;
          i_AB_OR   : in  std_logic;
          i_AB_XNOR : in  std_logic;
          i_S       : in  std_logic_vector(31 downto 0);
          i_RD      : in  std_logic_vector(31 downto 0);
          i_RT      : in  std_logic_vector(31 downto 0);
          i_OVER_CTRL : in  std_logic_vector(1 downto 0);
          o_Z       : out std_logic_vector(31 downto 0));
  end component;

  --| Declare Signals
  signal w_A : std_logic := '0';
  signal w_B : std_logic := '0';
  signal w_AB_NAND : std_logic;
  signal w_AB_OR : std_logic;
  signal w_AB_XNOR : std_logic;
  signal w_S : std_logic_vector(31 downto 0) := B"0000
   _0000_0000_0000_1010_0101_0110_1001";
  signal w_RD : std_logic_vector(31 downto 0) := B"1111
   _0000_1111_0000_1111_0000_1111_0000";
  signal w_RT : std_logic_vector(31 downto 0) := B"0101
   _0101_0101_0101_0101_0101_0101_0101";
  signal w_OVER_CTRL : std_logic_vector(1 downto 0) := B"00";
  signal w_Z : std_logic_vector(31 downto 0);
begin
  w_AB_NAND <= w_A NAND w_B;
  w_AB_OR <= w_A OR w_B;
  w_AB_XNOR <= w_A XNOR w_B;

  u_Over_Detect: Over_Detect
```

```
    port map(i_AB_NAND => w_AB_NAND,
40          i_AB_OR => w_AB_OR,
            i_AB_XNOR => w_AB_XNOR,
42          i_S => w_S,
            i_RD => w_RD,
44          i_RT => w_RT,
            i_OVER_CTRL => w_OVER_CTRL,
46          o_Z => w_Z);
    stimulus: process is
48  begin
      -- A, B, and S Positive  - Should not cause an overflow
50    wait for 20 ns;
      w_OVER_CTRL <= B"01";
52    wait for 20 ns;
      w_OVER_CTRL <= B"10";
54    wait for 20 ns;
      w_OVER_CTRL <= B"11";
56    wait for 20 ns;
      -- A and B positive, S negative.  Should cause an
     overflow
58    w_OVER_CTRL <= B"00";
      w_S <= B"1100_0000_0000_0000_1010_0101_0110_1001";
60    wait for 20 ns;
      w_OVER_CTRL <= B"01";
62    wait for 20 ns;
      w_OVER_CTRL <= B"10";
64    wait for 20 ns;
      w_OVER_CTRL <= B"11";
66    wait for 20 ns;
      -- A and B negative, S negative.  Should not cause an
     overflow
68    w_OVER_CTRL <= B"00";
      w_A <= '1';
70    w_B <= '1';
      wait for 20 ns;
72    w_OVER_CTRL <= B"01";
      wait for 20 ns;
74    w_OVER_CTRL <= B"10";
      wait for 20 ns;
76    w_OVER_CTRL <= B"11";
      wait for 20 ns;
78    -- A and B negative, S positive.  Should cause an
     overflow
```

```
    w_OVER_CTRL <= B"00";
    w_S <= B"0000_0000_0000_0000_1010_0101_0110_1001";
    wait for 20 ns;
    w_OVER_CTRL <= B"01";
    wait for 20 ns;
    w_OVER_CTRL <= B"10";
    wait for 20 ns;
    w_OVER_CTRL <= B"11";
    wait for 20 ns;
    -- A positive, B negative, and S positive.  Should not cause an overflow.
    w_OVER_CTRL <= B"00";
    w_A <= '0';
    wait for 20 ns;
    w_OVER_CTRL <= B"01";
    wait for 20 ns;
    w_OVER_CTRL <= B"10";
    wait for 20 ns;
    w_OVER_CTRL <= B"11";
    wait for 20 ns;
    -- A positive, B negative, and S negative.  Should not cause an overflow.
    w_OVER_CTRL <= B"00";
    w_S <= B"1100_0000_0000_0000_1010_0101_0110_1001";
    wait for 20 ns;
    w_OVER_CTRL <= B"01";
    wait for 20 ns;
    w_OVER_CTRL <= B"10";
    wait for 20 ns;
    w_OVER_CTRL <= B"11";
    wait for 20 ns;
    -- A negative, B positive, and S negative.  Should not cause an overflow.
    w_OVER_CTRL <= B"00";
    w_A <= '1';
    w_B <= '0';
    wait for 20 ns;
    w_OVER_CTRL <= B"01";
    wait for 20 ns;
    w_OVER_CTRL <= B"10";
    wait for 20 ns;
    w_OVER_CTRL <= B"11";
    wait for 20 ns;
```

```
    -- A negative, B positivie, and S positive.  Should not
    cause an overflow.
    w_OVER_CTRL <= B"00";
    w_S <= B"0000_0000_0000_0000_1010_0101_0110_1001";
    wait for 20 ns;
    w_OVER_CTRL <= B"01";
    wait for 20 ns;
    w_OVER_CTRL <= B"10";
    wait for 20 ns;
    w_OVER_CTRL <= B"11";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.30. Over_Detect_TB.vhd Code

2.11 Comparator

```
--| Comparator_TB.vhd
--| Test the functionality of the comparator
library IEEE;
use IEEE.std_logic_1164.all;

entity Comparator_TB is
end Comparator_TB;

architecture testbench of Comparator_TB is
--| Declare Components
  component Comparator is
    port (i_S      : in  std_logic_vector(31 downto 0);
          i_NAND_AB   : in  std_logic;
          i_OR_AB     : in  std_logic;
          i_XNOR_AB   : in  std_logic;
          i_unsigned  : in  std_logic;
          i_COMP_SEL  : in  std_logic_vector(2 downto 0);
          o_Z      : out std_logic);
  end component;

  --| Declare Signals
  signal w_S : std_logic_vector(31 downto 0) := (others =>
   '0');
  signal w_A : std_logic := '0';
  signal w_B : std_logic := '0';
  signal w_NAND_AB : std_logic;
  signal w_OR_AB : std_logic;
  signal w_XNOR_AB : std_logic;
  signal w_unsigned : std_logic := '0';
  signal w_COMP_SEL : std_logic_vector(2 downto 0) := (others
    => '0');
  signal w_Z : std_logic;
begin
  w_NAND_AB <= w_A NAND w_B;
  w_OR_AB <= w_A OR w_B;
  w_XNOR_AB <= w_A XNOR w_B;
  u_Comparator: Comparator
  port map (i_S => w_S,
            i_NAND_AB => w_NAND_AB,
            i_OR_AB => w_OR_AB,
            i_XNOR_AB => w_XNOR_AB,
```

```
            i_unsigned => w_UNSIGNED,
            i_COMP_SEL => w_COMP_SEL,
            o_Z => w_Z);

    stimulus: process is
    begin
      -- Check all outputs for A=B=0, unsigned = 0
      wait for 20 ns;
      w_COMP_SEL <= B"001";
      wait for 20 ns;
      w_COMP_SEL <= B"010";
      wait for 20 ns;
      w_COMP_SEL <= B"011";
      wait for 20 ns;
      w_COMP_SEL <= B"100";
      wait for 20 ns;
      w_COMP_SEL <= B"101";
      wait for 20 ns;
      w_COMP_SEL <= B"110";
      wait for 20 ns;
      w_COMP_SEL <= B"111";
      wait for 20 ns;
      w_COMP_SEL <= B"000";
      w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
      wait for 20 ns;
      w_COMP_SEL <= B"001";
      wait for 20 ns;
      w_COMP_SEL <= B"010";
      wait for 20 ns;
      w_COMP_SEL <= B"011";
      wait for 20 ns;
      w_COMP_SEL <= B"100";
      wait for 20 ns;
      w_COMP_SEL <= B"101";
      wait for 20 ns;
      w_COMP_SEL <= B"110";
      wait for 20 ns;
      w_COMP_SEL <= B"111";
      wait for 20 ns;
      w_COMP_SEL <= B"000";
      w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
      wait for 20 ns;
      w_COMP_SEL <= B"001";
```

```
83      wait for 20 ns;
        w_COMP_SEL <= B"010";
85      wait for 20 ns;
        w_COMP_SEL <= B"011";
87      wait for 20 ns;
        w_COMP_SEL <= B"100";
89      wait for 20 ns;
        w_COMP_SEL <= B"101";
91      wait for 20 ns;
        w_COMP_SEL <= B"110";
93      wait for 20 ns;
        w_COMP_SEL <= B"111";
95
        -- Check all outputs for A=1, B=0, unsigned = 0
97      wait for 20 ns;
        w_COMP_SEL <= B"000";
99      w_S <= (others => '0');
        w_A <= '1';
101     wait for 20 ns;
        w_COMP_SEL <= B"001";
103     wait for 20 ns;
        w_COMP_SEL <= B"010";
105     wait for 20 ns;
        w_COMP_SEL <= B"011";
107     wait for 20 ns;
        w_COMP_SEL <= B"100";
109     wait for 20 ns;
        w_COMP_SEL <= B"101";
111     wait for 20 ns;
        w_COMP_SEL <= B"110";
113     wait for 20 ns;
        w_COMP_SEL <= B"111";
115     wait for 20 ns;
        w_COMP_SEL <= B"000";
117     w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
        wait for 20 ns;
119     w_COMP_SEL <= B"001";
        wait for 20 ns;
121     w_COMP_SEL <= B"010";
        wait for 20 ns;
123     w_COMP_SEL <= B"011";
        wait for 20 ns;
125     w_COMP_SEL <= B"100";
```

```
        wait for 20 ns;
127     w_COMP_SEL <= B"101";
        wait for 20 ns;
129     w_COMP_SEL <= B"110";
        wait for 20 ns;
131     w_COMP_SEL <= B"111";
        wait for 20 ns;
133     w_COMP_SEL <= B"000";
        w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
135     wait for 20 ns;
        w_COMP_SEL <= B"001";
137     wait for 20 ns;
        w_COMP_SEL <= B"010";
139     wait for 20 ns;
        w_COMP_SEL <= B"011";
141     wait for 20 ns;
        w_COMP_SEL <= B"100";
143     wait for 20 ns;
        w_COMP_SEL <= B"101";
145     wait for 20 ns;
        w_COMP_SEL <= B"110";
147     wait for 20 ns;
        w_COMP_SEL <= B"111";
149
        -- Check all outputs for A=0, B=1, unsigned = 0
151     wait for 20 ns;
        w_COMP_SEL <= B"000";
153     w_S <= (others => '0');
        w_A <= '0';
155     w_B <= '1';
        wait for 20 ns;
157     w_COMP_SEL <= B"001";
        wait for 20 ns;
159     w_COMP_SEL <= B"010";
        wait for 20 ns;
161     w_COMP_SEL <= B"011";
        wait for 20 ns;
163     w_COMP_SEL <= B"100";
        wait for 20 ns;
165     w_COMP_SEL <= B"101";
        wait for 20 ns;
167     w_COMP_SEL <= B"110";
        wait for 20 ns;
```

```
169     w_COMP_SEL <= B"111";
        wait for 20 ns;
171     w_COMP_SEL <= B"000";
        w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
173     wait for 20 ns;
        w_COMP_SEL <= B"001";
175     wait for 20 ns;
        w_COMP_SEL <= B"010";
177     wait for 20 ns;
        w_COMP_SEL <= B"011";
179     wait for 20 ns;
        w_COMP_SEL <= B"100";
181     wait for 20 ns;
        w_COMP_SEL <= B"101";
183     wait for 20 ns;
        w_COMP_SEL <= B"110";
185     wait for 20 ns;
        w_COMP_SEL <= B"111";
187     wait for 20 ns;
        w_COMP_SEL <= B"000";
189     w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
        wait for 20 ns;
191     w_COMP_SEL <= B"001";
        wait for 20 ns;
193     w_COMP_SEL <= B"010";
        wait for 20 ns;
195     w_COMP_SEL <= B"011";
        wait for 20 ns;
197     w_COMP_SEL <= B"100";
        wait for 20 ns;
199     w_COMP_SEL <= B"101";
        wait for 20 ns;
201     w_COMP_SEL <= B"110";
        wait for 20 ns;
203     w_COMP_SEL <= B"111";

205     -- Check all outputs for A=1, B=1, unsigned = 0
        wait for 20 ns;
207     w_COMP_SEL <= B"000";
        w_S <= (others => '0');
209     w_A <= '1';
        wait for 20 ns;
211     w_COMP_SEL <= B"001";
```

```
        wait for 20 ns;
213     w_COMP_SEL <= B"010";
        wait for 20 ns;
215     w_COMP_SEL <= B"011";
        wait for 20 ns;
217     w_COMP_SEL <= B"100";
        wait for 20 ns;
219     w_COMP_SEL <= B"101";
        wait for 20 ns;
221     w_COMP_SEL <= B"110";
        wait for 20 ns;
223     w_COMP_SEL <= B"111";
        wait for 20 ns;
225     w_COMP_SEL <= B"000";
        w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
227     wait for 20 ns;
        w_COMP_SEL <= B"001";
229     wait for 20 ns;
        w_COMP_SEL <= B"010";
231     wait for 20 ns;
        w_COMP_SEL <= B"011";
233     wait for 20 ns;
        w_COMP_SEL <= B"100";
235     wait for 20 ns;
        w_COMP_SEL <= B"101";
237     wait for 20 ns;
        w_COMP_SEL <= B"110";
239     wait for 20 ns;
        w_COMP_SEL <= B"111";
241     wait for 20 ns;
        w_COMP_SEL <= B"000";
243     w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
        wait for 20 ns;
245     w_COMP_SEL <= B"001";
        wait for 20 ns;
247     w_COMP_SEL <= B"010";
        wait for 20 ns;
249     w_COMP_SEL <= B"011";
        wait for 20 ns;
251     w_COMP_SEL <= B"100";
        wait for 20 ns;
253     w_COMP_SEL <= B"101";
        wait for 20 ns;
```

```
255     w_COMP_SEL <= B"110";
        wait for 20 ns;
257     w_COMP_SEL <= B"111";

259     -- Check all outputs for A=B=0, unsigned = 1
        wait for 20 ns;
261     w_COMP_SEL <= B"000";
        w_S <= (others => '0');
263     w_A <= '0';
        w_B <= '0';
265     w_unsigned <= '1';
        wait for 20 ns;
267     w_COMP_SEL <= B"001";
        wait for 20 ns;
269     w_COMP_SEL <= B"010";
        wait for 20 ns;
271     w_COMP_SEL <= B"011";
        wait for 20 ns;
273     w_COMP_SEL <= B"100";
        wait for 20 ns;
275     w_COMP_SEL <= B"101";
        wait for 20 ns;
277     w_COMP_SEL <= B"110";
        wait for 20 ns;
279     w_COMP_SEL <= B"111";
        wait for 20 ns;
281     w_COMP_SEL <= B"000";
        w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
283     wait for 20 ns;
        w_COMP_SEL <= B"001";
285     wait for 20 ns;
        w_COMP_SEL <= B"010";
287     wait for 20 ns;
        w_COMP_SEL <= B"011";
289     wait for 20 ns;
        w_COMP_SEL <= B"100";
291     wait for 20 ns;
        w_COMP_SEL <= B"101";
293     wait for 20 ns;
        w_COMP_SEL <= B"110";
295     wait for 20 ns;
        w_COMP_SEL <= B"111";
297     wait for 20 ns;
```

```
    w_COMP_SEL <= B"000";
    w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
    wait for 20 ns;
    w_COMP_SEL <= B"001";
    wait for 20 ns;
    w_COMP_SEL <= B"010";
    wait for 20 ns;
    w_COMP_SEL <= B"011";
    wait for 20 ns;
    w_COMP_SEL <= B"100";
    wait for 20 ns;
    w_COMP_SEL <= B"101";
    wait for 20 ns;
    w_COMP_SEL <= B"110";
    wait for 20 ns;
    w_COMP_SEL <= B"111";

    -- Check all outputs for A=1, B=0, unsigned = 1
    wait for 20 ns;
    w_COMP_SEL <= B"000";
    w_S <= (others => '0');
    w_A <= '1';
    wait for 20 ns;
    w_COMP_SEL <= B"001";
    wait for 20 ns;
    w_COMP_SEL <= B"010";
    wait for 20 ns;
    w_COMP_SEL <= B"011";
    wait for 20 ns;
    w_COMP_SEL <= B"100";
    wait for 20 ns;
    w_COMP_SEL <= B"101";
    wait for 20 ns;
    w_COMP_SEL <= B"110";
    wait for 20 ns;
    w_COMP_SEL <= B"111";
    wait for 20 ns;
    w_COMP_SEL <= B"000";
    w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
    wait for 20 ns;
    w_COMP_SEL <= B"001";
    wait for 20 ns;
    w_COMP_SEL <= B"010";
```

```
341         wait for 20 ns;
            w_COMP_SEL <= B"011";
343         wait for 20 ns;
            w_COMP_SEL <= B"100";
345         wait for 20 ns;
            w_COMP_SEL <= B"101";
347         wait for 20 ns;
            w_COMP_SEL <= B"110";
349         wait for 20 ns;
            w_COMP_SEL <= B"111";
351         wait for 20 ns;
            w_COMP_SEL <= B"000";
353         w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
            wait for 20 ns;
355         w_COMP_SEL <= B"001";
            wait for 20 ns;
357         w_COMP_SEL <= B"010";
            wait for 20 ns;
359         w_COMP_SEL <= B"011";
            wait for 20 ns;
361         w_COMP_SEL <= B"100";
            wait for 20 ns;
363         w_COMP_SEL <= B"101";
            wait for 20 ns;
365         w_COMP_SEL <= B"110";
            wait for 20 ns;
367         w_COMP_SEL <= B"111";

369         -- Check all outputs for A=0, B=1, unsigned = 1
            wait for 20 ns;
371         w_COMP_SEL <= B"000";
            w_S <= (others => '0');
373         w_A <= '0';
            w_B <= '1';
375         wait for 20 ns;
            w_COMP_SEL <= B"001";
377         wait for 20 ns;
            w_COMP_SEL <= B"010";
379         wait for 20 ns;
            w_COMP_SEL <= B"011";
381         wait for 20 ns;
            w_COMP_SEL <= B"100";
383         wait for 20 ns;
```

```
        w_COMP_SEL <= B"101";
385     wait for 20 ns;
        w_COMP_SEL <= B"110";
387     wait for 20 ns;
        w_COMP_SEL <= B"111";
389     wait for 20 ns;
        w_COMP_SEL <= B"000";
391     w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
        wait for 20 ns;
393     w_COMP_SEL <= B"001";
        wait for 20 ns;
395     w_COMP_SEL <= B"010";
        wait for 20 ns;
397     w_COMP_SEL <= B"011";
        wait for 20 ns;
399     w_COMP_SEL <= B"100";
        wait for 20 ns;
401     w_COMP_SEL <= B"101";
        wait for 20 ns;
403     w_COMP_SEL <= B"110";
        wait for 20 ns;
405     w_COMP_SEL <= B"111";
        wait for 20 ns;
407     w_COMP_SEL <= B"000";
        w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
409     wait for 20 ns;
        w_COMP_SEL <= B"001";
411     wait for 20 ns;
        w_COMP_SEL <= B"010";
413     wait for 20 ns;
        w_COMP_SEL <= B"011";
415     wait for 20 ns;
        w_COMP_SEL <= B"100";
417     wait for 20 ns;
        w_COMP_SEL <= B"101";
419     wait for 20 ns;
        w_COMP_SEL <= B"110";
421     wait for 20 ns;
        w_COMP_SEL <= B"111";
423
        -- Check all outputs for A=1, B=1, unsigned = 1
425     wait for 20 ns;
        w_COMP_SEL <= B"000";
```

```
427     w_S <= (others => '0');
        w_A <= '1';
429     wait for 20 ns;
        w_COMP_SEL <= B"001";
431     wait for 20 ns;
        w_COMP_SEL <= B"010";
433     wait for 20 ns;
        w_COMP_SEL <= B"011";
435     wait for 20 ns;
        w_COMP_SEL <= B"100";
437     wait for 20 ns;
        w_COMP_SEL <= B"101";
439     wait for 20 ns;
        w_COMP_SEL <= B"110";
441     wait for 20 ns;
        w_COMP_SEL <= B"111";
443     wait for 20 ns;
        w_COMP_SEL <= B"000";
445     w_S <= B"0000_0110_1000_0100_0001_0010_0100_0000";
        wait for 20 ns;
447     w_COMP_SEL <= B"001";
        wait for 20 ns;
449     w_COMP_SEL <= B"010";
        wait for 20 ns;
451     w_COMP_SEL <= B"011";
        wait for 20 ns;
453     w_COMP_SEL <= B"100";
        wait for 20 ns;
455     w_COMP_SEL <= B"101";
        wait for 20 ns;
457     w_COMP_SEL <= B"110";
        wait for 20 ns;
459     w_COMP_SEL <= B"111";
        wait for 20 ns;
461     w_COMP_SEL <= B"000";
        w_S <= B"1000_0110_1000_0100_0001_0010_0100_0000";
463     wait for 20 ns;
        w_COMP_SEL <= B"001";
465     wait for 20 ns;
        w_COMP_SEL <= B"010";
467     wait for 20 ns;
        w_COMP_SEL <= B"011";
469     wait for 20 ns;
```

```
    w_COMP_SEL <= B"100";
    wait for 20 ns;
    w_COMP_SEL <= B"101";
    wait for 20 ns;
    w_COMP_SEL <= B"110";
    wait for 20 ns;
    w_COMP_SEL <= B"111";
    wait;
  end process stimulus;
end testbench;
```

Listing B.31. Comparator_TB.vhd Code

2.12 Signed/Unsigned Extensions

2.12.1 Signed 16-Bit to 32-Bit Extension

```
--| Sign_Extend16_32_TB.vhd
--| Test the Sign_Extend16_32.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity Sign_Extend16_32_TB is
end Sign_Extend16_32_TB;

architecture testbench of Sign_Extend16_32_TB is
--| Declare components
  component Sign_Extend16_32 is
    port (i_A : in  std_logic_vector(15 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

--| Declare Signals
  signal w_A : std_logic_vector(15 downto 0) := B"0000
   _1111_0101_1001";
  signal w_Z : std_logic_vector(31 downto 0);
begin
  u_Sign_Extend16_32: Sign_Extend16_32
  port map(i_A => w_A,
           o_Z => w_Z);

  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"1100_1111_0101_1001";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.32. Sign_Extend16_32_TB.vhd Code

2.12.2 Unsigned 5-Bit to 32-Bit Extension

```
--| Extend5_32_TB.vhd
--| Test the Extend5_32.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity Extend5_32_TB is
end Extend5_32_TB;

architecture testbench of Extend5_32_TB is
--| Declare components
  component Extend5_32 is
    port (i_A : in  std_logic_vector(4 downto 0);
          o_Z : out std_logic_vector(31 downto 0));
  end component;

--| Declare Signals
  signal w_A : std_logic_vector(15 downto 0) := B"0_1001";
  signal w_Z : std_logic_vector(31 downto 0);
begin
  u_Extend5_32: Extend5_32
  port map(i_A => w_A,
           o_Z => w_Z);

  stimulus: process is
  begin
    wait for 20 ns;
    w_A <= B"1_1001";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.33. Extend5_32_TB.vhd Code

2.13 1-Bit Register

```
--| myReg1.vhd
--| Test 1-bit register
library IEEE;
use IEEE.std_logic_1164.all;

entity myReg1_TB is
end myReg1_TB;

architecture testbench of myReg1_TB is
--| Declare components
  component myReg1 is
    port (i_clk   : in  std_logic;
          i_reset : in  std_logic;
          i_D     : in  std_logic;
          o_Q     : out std_logic);
  end component;

--| Declare signals
  signal c_clk : std_logic := '0';
  signal w_reset : std_logic := '1';
  signal w_D : std_logic := '1';
  signal w_Q : std_logic;
begin
  u_myReg1: myReg1
  port map (i_clk => c_clk,
            i_reset => w_reset,
            i_D => w_D,
            o_Q => w_Q);

  --| Generate the stimulus
  stimulus : process
   begin
    wait for 100 ns;
    w_reset <= '0';
    wait for 10 ns;
    w_D <= '0';
    wait for 20 ns;
    w_D <= '1';
    wait for 20 ns;
    w_D <= '0';
    wait for 20 ns;
```

```
    w_D <= '1';
43    wait;
   end process;
45
  --| Generate the clock signal
47  clock_gen: process is
  begin
49    c_clk <= '1' after 10 ns, '0' after 20 ns;
    wait for 20 ns;
51  end process clock_gen;
end testbench;
```

Listing B.34. myReg1_TB.vhd Code

2.14 5-Bit to 32-Bit Decoder

```
--| Decoder5_32_TB
--| Tests Decoder5_32
library IEEE;
use IEEE.std_logic_1164.all;

entity Decoder5_32_TB is
end Decoder5_32_TB;

architecture testbench of Decoder5_32_TB is
  component Decoder5_32 is
    port (i_S : in  std_logic_vector(4 downto 0);
          o_Z : out std_logic_vector(31 downto 0)
          );
  end component;

  -- Declare signals
  signal w_S  : std_logic_vector(4 downto 0) := (others =>
   '0');
  signal w_Z  : std_logic_vector(31 downto 0);

begin
  -- Connect Decoder5_32
  u_Decoder5_32: Decoder5_32
  port map (i_S => w_S,
            o_Z => w_Z);

  -- Process for stimulating inputs
  stimulus: process is
  begin
    wait for 20 ns;
    w_S <= B"00001";
    wait for 20 ns;
    w_S <= B"00010";
    wait for 20 ns;
    w_S <= B"00011";
    wait for 20 ns;
    w_S <= B"00100";
    wait for 20 ns;
    w_S <= B"00101";
    wait for 20 ns;
    w_S <= B"00110";
```

```
    wait for 20 ns;
    w_S <= B"00111";
    wait for 20 ns;
    w_S <= B"01000";
    wait for 20 ns;
    w_S <= B"01001";
    wait for 20 ns;
    w_S <= B"01010";
    wait for 20 ns;
    w_S <= B"01011";
    wait for 20 ns;
    w_S <= B"01100";
    wait for 20 ns;
    w_S <= B"01101";
    wait for 20 ns;
    w_S <= B"01110";
    wait for 20 ns;
    w_S <= B"01111";
    wait for 20 ns;
    w_S <= B"10000";
    wait for 20 ns;
    w_S <= B"10001";
    wait for 20 ns;
    w_S <= B"10010";
    wait for 20 ns;
    w_S <= B"10011";
    wait for 20 ns;
    w_S <= B"10100";
    wait for 20 ns;
    w_S <= B"10101";
    wait for 20 ns;
    w_S <= B"10110";
    wait for 20 ns;
    w_S <= B"10111";
    wait for 20 ns;
    w_S <= B"11000";
    wait for 20 ns;
    w_S <= B"11001";
    wait for 20 ns;
    w_S <= B"11010";
    wait for 20 ns;
    w_S <= B"11011";
    wait for 20 ns;
```

```
84      w_S <= B"11100";
        wait for 20 ns;
86      w_S <= B"11101";
        wait for 20 ns;
88      w_S <= B"11110";
        wait for 20 ns;
90      w_S <= B"11111";
        wait for 20 ns;
92      wait;
    end process stimulus;
94  end testbench;
```

Listing B.35. Decoder5_32_TB.vhd Code

2.15 N-Bit Register

```
--| myRegN.vhd
--| Test N-bit register
library IEEE;
use IEEE.std_logic_1164.all;

entity myRegN_TB is
end myRegN_TB;

architecture testbench of myRegN_TB is
--| Declare components
  component myRegN is
    generic (m_width : integer := 32);
    port (i_clk    : in  std_logic;
          i_reset  : in  std_logic;
          i_D      : in  std_logic_vector (m_width-1 downto 0);
          o_Q      : out std_logic_vector (m_width-1 downto 0));
  end component;

--| Declare signals
  signal c_clk : std_logic := '0';
  signal w_reset : std_logic := '1';
  signal w_D : std_logic_vector (31 downto 0) := B"1111
   _1111_1111_1111_1111_1111_1111_1111";
  signal w_Q : std_logic_vector (31 downto 0);
begin
  u_myRegN: myRegN
  generic map (m_width => 32)
  port map (i_clk => c_clk,
            i_reset => w_reset,
            i_D => w_D,
            o_Q => w_Q);

  --| Generate the stimulus
  stimulus : process
   begin
    wait for 100 ns;
    w_reset <= '0';
    wait for 20 ns;
    w_D <= (others => '0');
    wait for 20 ns;
    w_D <= B"0101_0101_0101_0101_0101_0101_0101_0101";
```

```
    wait for 20 ns;
    w_D <= B"1010_1010_1010_1010_1010_1010_1010_1010";
    wait for 20 ns;
    w_D <= B"0000_1111_0000_1111_0000_1111_0000_1111";
    wait;
  end process;

  --| Generate the clock signal
  clock_gen: process is
  begin
    c_clk <= '1' after 10 ns, '0' after 20 ns;
    wait for 20 ns;
  end process clock_gen;
end testbench;
```

Listing B.36. myRegN_TB.vhd Code

2.16 Enabled Register

```
--| Enabled_Register_TB.vhd
--| Bank of 32, 32-bit registers. On the rising clock edge, only the contents of
--| the selected register are modified to be equal to the value of i_data. All
--| other registers retain their previous value.
--|
--| INPUTS:
--| i_clk - Clock
--| i_reset - Reset
--| i_data - Data
--| i_sel - Encoded select signal to determine which register should be enabled
--|
--| OUTPUTS:
--| o_Q00 - Output of register 0
--| o_Q01 - Output of register 1
--| ...
--| o_Q30 - Output of register 30
--| o_Q31 - Output of register 31
library IEEE;
use IEEE.std_logic_1164.all;

entity Enabled_Register_TB is
end Enabled_Register_TB;

architecture test_bench of Enabled_Register_TB is
--| Declare Components
  component Enabled_Register is
    port (i_clk   : in  std_logic;
          i_reset : in  std_logic;
          i_data  : in  std_logic_vector(31 downto 0);
          i_en    : in  std_logic;
          o_Q     : out std_logic_vector(31 downto 0));
  end component;

--| Declare Signals
  signal c_clk : std_logic := '0';
  signal w_reset : std_logic := '1';
  signal w_data : std_logic_vector(31 downto 0) := B"0000_0000_0000_0000_0000_0000_0000_0001";
```

```
38    signal w_en : std_logic := '0';
      signal w_Q : std_logic_vector(31 downto 0);
40  begin
    -- Connect the Enabled_Register
42    u_Enabled_Register: Enabled_Register
      port map (i_clk => c_clk,
44              i_reset => w_reset,
                i_data => w_data,
46              i_en => w_en,
                o_Q => w_Q);
48
      stimulus : process
50     begin
        wait for 100 ns;
52      w_reset <= '0';
        wait for 20 ns;
54      w_data <= B"0000_0000_0000_0000_0000_0000_0000_0010";
        w_en <= '1';
56      wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0000_0000_0100";
58      w_en <= '0';
        wait for 20 ns;
60      w_data <= B"0000_0000_0000_0000_0000_0000_0000_1000";
        wait for 20 ns;
62      w_data <= B"0000_0000_0000_0000_0000_0000_0001_0000";
        wait for 20 ns;
64      w_data <= B"0000_0000_0000_0000_0000_0000_0010_0000";
        wait for 20 ns;
66      w_data <= B"0000_0000_0000_0000_0000_0000_0100_0000";
        w_en <= '1';
68      wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0000_1000_0000";
70      wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0001_0000_0000";
72      wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0010_0000_0000";
74      w_en <= '0';
        wait for 20 ns;
76      w_data <= B"0000_0000_0000_0000_0000_0100_0000_0000";
        wait for 20 ns;
78      wait;
       end process;
80
```

```
  --| Generate the clock signal
82  clock_gen: process is
    begin
84    c_clk <= '1' after 10 ns, '0' after 20 ns;
      wait for 20 ns;
86  end process clock_gen;

88 end test_bench;
```

Listing B.37. Enabled_Register_TB.vhd Code

2.17 GPR Bank

```
--| GPR_Bank_TB.vhd
--| Bank of 32, 32-bit registers. On the rising clock edge, only the contents of
--| the selected register are modified to be equal to the value of i_data. All
--| other registers retain their previous value.
--|
--| INPUTS:
--| i_clk - Clock
--| i_reset - Reset
--| i_data - Data
--| i_sel - Encoded select signal to determine which register should be enabled
--|
--| OUTPUTS:
--| o_Q00 - Output of register 0
--| o_Q01 - Output of register 1
--| ...
--| o_Q30 - Output of register 30
--| o_Q31 - Output of register 31
library IEEE;
use IEEE.std_logic_1164.all;

entity GPR_Bank_TB is
end GPR_Bank_TB;

architecture test_bench of GPR_Bank_TB is
--| Declare Components
  component GPR_Bank is
    port (i_clk   : in  std_logic;
          i_reset : in  std_logic;
          i_data  : in  std_logic_vector(31 downto 0);
          i_sel   : in  std_logic_vector(4 downto 0);
          o_Q00   : out std_logic_vector(31 downto 0);
          o_Q01   : out std_logic_vector(31 downto 0);
          o_Q02   : out std_logic_vector(31 downto 0);
          o_Q03   : out std_logic_vector(31 downto 0);
          o_Q04   : out std_logic_vector(31 downto 0);
          o_Q05   : out std_logic_vector(31 downto 0);
          o_Q06   : out std_logic_vector(31 downto 0);
          o_Q07   : out std_logic_vector(31 downto 0);
```

```
        o_Q08    : out std_logic_vector(31 downto 0);
        o_Q09    : out std_logic_vector(31 downto 0);
        o_Q10    : out std_logic_vector(31 downto 0);
        o_Q11    : out std_logic_vector(31 downto 0);
        o_Q12    : out std_logic_vector(31 downto 0);
        o_Q13    : out std_logic_vector(31 downto 0);
        o_Q14    : out std_logic_vector(31 downto 0);
        o_Q15    : out std_logic_vector(31 downto 0);
        o_Q16    : out std_logic_vector(31 downto 0);
        o_Q17    : out std_logic_vector(31 downto 0);
        o_Q18    : out std_logic_vector(31 downto 0);
        o_Q19    : out std_logic_vector(31 downto 0);
        o_Q20    : out std_logic_vector(31 downto 0);
        o_Q21    : out std_logic_vector(31 downto 0);
        o_Q22    : out std_logic_vector(31 downto 0);
        o_Q23    : out std_logic_vector(31 downto 0);
        o_Q24    : out std_logic_vector(31 downto 0);
        o_Q25    : out std_logic_vector(31 downto 0);
        o_Q26    : out std_logic_vector(31 downto 0);
        o_Q27    : out std_logic_vector(31 downto 0);
        o_Q28    : out std_logic_vector(31 downto 0);
        o_Q29    : out std_logic_vector(31 downto 0);
        o_Q30    : out std_logic_vector(31 downto 0);
        o_Q31    : out std_logic_vector(31 downto 0));
    end component;

--| Declare Signals

    signal c_clk : std_logic := '0';
    signal w_reset : std_logic := '1';
    signal w_data : std_logic_vector(31 downto 0) := B"0000
     _0000_0000_0000_0000_0000_0000_0001";
    signal w_sel : std_logic_vector(4 downto 0) := (others =>
     '0');
    signal w_Q00 : std_logic_vector(31 downto 0);
    signal w_Q01 : std_logic_vector(31 downto 0);
    signal w_Q02 : std_logic_vector(31 downto 0);
    signal w_Q03 : std_logic_vector(31 downto 0);
    signal w_Q04 : std_logic_vector(31 downto 0);
    signal w_Q05 : std_logic_vector(31 downto 0);
    signal w_Q06 : std_logic_vector(31 downto 0);
    signal w_Q07 : std_logic_vector(31 downto 0);
    signal w_Q08 : std_logic_vector(31 downto 0);
```

```
80    signal w_Q09 : std_logic_vector(31 downto 0);
      signal w_Q10 : std_logic_vector(31 downto 0);
82    signal w_Q11 : std_logic_vector(31 downto 0);
      signal w_Q12 : std_logic_vector(31 downto 0);
84    signal w_Q13 : std_logic_vector(31 downto 0);
      signal w_Q14 : std_logic_vector(31 downto 0);
86    signal w_Q15 : std_logic_vector(31 downto 0);
      signal w_Q16 : std_logic_vector(31 downto 0);
88    signal w_Q17 : std_logic_vector(31 downto 0);
      signal w_Q18 : std_logic_vector(31 downto 0);
90    signal w_Q19 : std_logic_vector(31 downto 0);
      signal w_Q20 : std_logic_vector(31 downto 0);
92    signal w_Q21 : std_logic_vector(31 downto 0);
      signal w_Q22 : std_logic_vector(31 downto 0);
94    signal w_Q23 : std_logic_vector(31 downto 0);
      signal w_Q24 : std_logic_vector(31 downto 0);
96    signal w_Q25 : std_logic_vector(31 downto 0);
      signal w_Q26 : std_logic_vector(31 downto 0);
98    signal w_Q27 : std_logic_vector(31 downto 0);
      signal w_Q28 : std_logic_vector(31 downto 0);
100   signal w_Q29 : std_logic_vector(31 downto 0);
      signal w_Q30 : std_logic_vector(31 downto 0);
102   signal w_Q31 : std_logic_vector(31 downto 0);
  begin
104   -- Connect the GPR_BANK
      u_GPR_BANK: GPR_BANK
106   port map (i_clk => c_clk,
              i_reset => w_reset,
108           i_data => w_data,
              i_sel => w_sel,
110           o_Q00 => w_Q00,
              o_Q01 => w_Q01,
112           o_Q02 => w_Q02,
              o_Q03 => w_Q03,
114           o_Q04 => w_Q04,
              o_Q05 => w_Q05,
116           o_Q06 => w_Q06,
              o_Q07 => w_Q07,
118           o_Q08 => w_Q08,
              o_Q09 => w_Q09,
120           o_Q10 => w_Q10,
              o_Q11 => w_Q11,
122           o_Q12 => w_Q12,
```

```
            o_Q13 => w_Q13,
            o_Q14 => w_Q14,
            o_Q15 => w_Q15,
            o_Q16 => w_Q16,
            o_Q17 => w_Q17,
            o_Q18 => w_Q18,
            o_Q19 => w_Q19,
            o_Q20 => w_Q20,
            o_Q21 => w_Q21,
            o_Q22 => w_Q22,
            o_Q23 => w_Q23,
            o_Q24 => w_Q24,
            o_Q25 => w_Q25,
            o_Q26 => w_Q26,
            o_Q27 => w_Q27,
            o_Q28 => w_Q28,
            o_Q29 => w_Q29,
            o_Q30 => w_Q30,
            o_Q31 => w_Q31);

    stimulus : process
     begin
      wait for 100 ns;
      w_reset <= '0';
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0000_0010";
      w_sel <= B"00001";
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0000_0100";
      w_sel <= B"00010";
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0000_1000";
      w_sel <= B"00011";
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0001_0000";
      w_sel <= B"00100";
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0010_0000";
      w_sel <= B"00101";
      wait for 20 ns;
      w_data <= B"0000_0000_0000_0000_0000_0000_0100_0000";
      w_sel <= B"00110";
      wait for 20 ns;
```

```
166     w_data <= B"0000_0000_0000_0000_0000_0000_1000_0000";
        w_sel <= B"00111";
168     wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0001_0000_0000";
170     w_sel <= B"01000";
        wait for 20 ns;
172     w_data <= B"0000_0000_0000_0000_0000_0010_0000_0000";
        w_sel <= B"01001";
174     wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0000_0100_0000_0000";
176     w_sel <= B"01010";
        wait for 20 ns;
178     w_data <= B"0000_0000_0000_0000_0000_1000_0000_0000";
        w_sel <= B"01011";
180     wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0001_0000_0000_0000";
182     w_sel <= B"01100";
        wait for 20 ns;
184     w_data <= B"0000_0000_0000_0000_0010_0000_0000_0000";
        w_sel <= B"01101";
186     wait for 20 ns;
        w_data <= B"0000_0000_0000_0000_0100_0000_0000_0000";
188     w_sel <= B"01110";
        wait for 20 ns;
190     w_data <= B"0000_0000_0000_0000_1000_0000_0000_0000";
        w_sel <= B"01111";
192     wait for 20 ns;
        w_data <= B"0000_0000_0000_0001_0000_0000_0000_0000";
194     w_sel <= B"10000";
        wait for 20 ns;
196     w_data <= B"0000_0000_0000_0010_0000_0000_0000_0000";
        w_sel <= B"10001";
198     wait for 20 ns;
        w_data <= B"0000_0000_0000_0100_0000_0000_0000_0000";
200     w_sel <= B"10010";
        wait for 20 ns;
202     w_data <= B"0000_0000_0000_1000_0000_0000_0000_0000";
        w_sel <= B"10011";
204     wait for 20 ns;
        w_data <= B"0000_0000_0001_0000_0000_0000_0000_0000";
206     w_sel <= B"10100";
        wait for 20 ns;
208     w_data <= B"0000_0000_0010_0000_0000_0000_0000_0000";
```

```
    w_sel <= B"10101";
    wait for 20 ns;
    w_data <= B"0000_0000_0100_0000_0000_0000_0000_0000";
    w_sel <= B"10110";
    wait for 20 ns;
    w_data <= B"0000_0000_1000_0000_0000_0000_0000_0000";
    w_sel <= B"10111";
    wait for 20 ns;
    w_data <= B"0000_0001_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11000";
    wait for 20 ns;
    w_data <= B"0000_0010_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11001";
    wait for 20 ns;
    w_data <= B"0000_0100_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11010";
    wait for 20 ns;
    w_data <= B"0000_1000_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11011";
    wait for 20 ns;
    w_data <= B"0001_0000_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11100";
    wait for 20 ns;
    w_data <= B"0010_0000_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11101";
    wait for 20 ns;
    w_data <= B"0100_0000_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11110";
    wait for 20 ns;
    w_data <= B"1000_0000_0000_0000_0000_0000_0000_0000";
    w_sel <= B"11111";
    wait for 20 ns;
    wait;
  end process;

  --| Generate the clock signal
  clock_gen: process is
  begin
    c_clk <= '1' after 10 ns, '0' after 20 ns;
    wait for 20 ns;
  end process clock_gen;

end test_bench;
```

Listing B.38. Enabled_Register_TB.vhd Code

2.18 Instruction Decoder

```
--| Instruction_Decoder_TB
--| Tests Instruction_Decoder
library IEEE;
use IEEE.std_logic_1164.all;

entity Instruction_Decoder_TB is
end Instruction_Decoder_TB;

architecture testbench of Instruction_Decoder_TB is
  component Instruction_Decoder is
    port (i_instr31   : in  std_logic;
          i_instr30   : in  std_logic;
          i_instr29   : in  std_logic;
          i_instr28   : in  std_logic;
          i_instr27   : in  std_logic;
          i_instr26   : in  std_logic;
          i_instr20   : in  std_logic;
          i_instr19   : in  std_logic;
          i_instr18   : in  std_logic;
          i_instr17   : in  std_logic;
          i_instr16   : in  std_logic;
          i_instr05   : in  std_logic;
          i_instr04   : in  std_logic;
          i_instr03   : in  std_logic;
          i_instr02   : in  std_logic;
          i_instr01   : in  std_logic;
          i_instr00   : in  std_logic;
          o_ALU_SRC_A0  : out std_logic;
          o_ALU_SRC_B   : out std_logic_vector(1 downto 0);
          o_INV_B       : out std_logic;
          o_LTZ         : out std_logic;
          o_EQ          : out std_logic;
          o_NE          : out std_logic;
          o_LEZ         : out std_logic;
          o_GTZ         : out std_logic;
          o_RIGHT_SHIFT : out std_logic;
          o_ADDER       : out std_logic;
          o_AND         : out std_logic;
          o_OR          : out std_logic;
          o_XOR         : out std_logic;
          o_NOR         : out std_logic;
```

```
        o_COMPARE    : out std_logic;
        o_LUI        : out std_logic;
        o_REG_SEL    : out std_logic_vector(1 downto 0);
        o_OVER_CTRL  : out std_logic_vector(1 downto 0);
        o_UNSIGNED   : out std_logic;
        o_imm_extend : out std_logic;
        o_FSM_CTRL   : out std_logic_vector(1 downto 0);
        o_INVALID_INSTR: out std_logic);
  end component;

  -- Declare signals
  signal w_instruction  : std_logic_vector(31 downto 0);
  signal w_ALU_SRC_A0   : std_logic;
  signal w_ALU_SRC_B    : std_logic_vector(1 downto 0);
  signal w_INV_B        : std_logic;
  signal w_LTZ          : std_logic;
  signal w_EQ           : std_logic;
  signal w_NE           : std_logic;
  signal w_LEZ          : std_logic;
  signal w_GTZ          : std_logic;
  signal w_RIGHT_SHIFT  : std_logic;
  signal w_ADDER        : std_logic;
  signal w_AND          : std_logic;
  signal w_OR           : std_logic;
  signal w_XOR          : std_logic;
  signal w_NOR          : std_logic;
  signal w_COMPARE      : std_logic;
  signal w_LUI          : std_logic;
  signal w_REG_SEL      : std_logic_vector(1 downto 0);
  signal w_OVER_CTRL    : std_logic_vector(1 downto 0);
  signal w_UNSIGNED     : std_logic;
  signal w_imm_extend   : std_logic;
  signal w_FSM_CTRL     : std_logic_vector(1 downto 0);
  signal w_INVALID_INSTR : std_logic;

begin
  -- Connect myNAND2
  u_Instruction_Decoder: Instruction_Decoder
  port map (i_instr31 => w_instruction(31),
        i_instr30 => w_instruction(30),
        i_instr29 => w_instruction(29),
        i_instr28 => w_instruction(28),
        i_instr27 => w_instruction(27),
```

```
85            i_instr26 => w_instruction(26),
              i_instr20 => w_instruction(20),
87            i_instr19 => w_instruction(19),
              i_instr18 => w_instruction(18),
89            i_instr17 => w_instruction(17),
              i_instr16 => w_instruction(16),
91            i_instr05 => w_instruction(5),
              i_instr04 => w_instruction(4),
93            i_instr03 => w_instruction(3),
              i_instr02 => w_instruction(2),
95            i_instr01 => w_instruction(1),
              i_instr00 => w_instruction(0),
97           o_ALU_SRC_A0 => w_ALU_SRC_A0,
             o_ALU_SRC_B => w_ALU_SRC_B,
99           o_INV_B => w_INV_B,
             o_LTZ => w_LTZ,
101          o_EQ => w_EQ,
             o_NE => w_NE,
103          o_LEZ => w_LEZ,
             o_GTZ => w_GTZ,
105          o_RIGHT_SHIFT => w_RIGHT_SHIFT,
             o_ADDER => w_ADDER,
107          o_AND => w_AND,
             o_OR => w_OR,
109          o_XOR => w_XOR,
             o_NOR => w_NOR,
111          o_COMPARE => w_COMPARE,
             o_LUI => w_LUI,
113          o_REG_SEL => w_REG_SEL,
             o_OVER_CTRL => w_OVER_CTRL,
115          o_UNSIGNED => w_UNSIGNED,
             o_imm_extend => w_imm_extend,
117          o_FSM_CTRL => w_FSM_CTRL,
             o_INVALID_INSTR => w_INVALID_INSTR);
119
   -- Process for stimulating inputs
121 stimulus: process is
   begin
123   -- SLL R2, R1, 5
      w_instruction <= B"00000000000000010001000101000000";
125   wait for 20 ns;
      -- NOP
127   w_instruction <= B"00000000000000000000000000000000";
```

```
        wait for 20 ns;
129     -- SRL R3, R2, 7
        w_instruction <= B"00000000000000100001100111000010";
131     wait for 20 ns;
        -- SRA R5 R4 15
133     w_instruction <= B"00000000000001000010101111000011";
        wait for 20 ns;
135     -- SLLV R3 R2 R1
        w_instruction <= B"00000000010001000011000000000100";
137     wait for 20 ns;
        -- SRLV R7 R6 R8
139     w_instruction <= B"00000001000001100011100000000110";
        wait for 20 ns;
141     -- SRAV R9 R10 R11
        w_instruction <= B"00000001011010100100100000000111";
143     wait for 20 ns;
        -- ADD R12 R13 R14
145     w_instruction <= B"00000001101011100110000000100000";
        wait for 20 ns;
147     -- ADDU R15 R16 R17
        w_instruction <= B"00000010000100010111100000100001";
149     wait for 20 ns;
        -- SUB R18 R19 R20
151     w_instruction <= B"00000010011101001001000000100010";
        wait for 20 ns;
153     -- SUBU R21 R22 R23
        w_instruction <= B"00000010110101111010100000100011";
155     wait for 20 ns;
        -- AND R24 R25 R26
157     w_instruction <= B"00000011001110101100000000100100";
        wait for 20 ns;
159     -- OR R27 R28 R29
        w_instruction <= B"00000011101111001101100000100101";
161     wait for 20 ns;
        -- XOR R30 R31 R0
163     w_instruction <= B"00000011111000000111000000100110";
        wait for 20 ns;
165     -- NOR R1 R2 R3
        w_instruction <= B"00000000010000110000100000100111";
167     wait for 20 ns;
        -- SLT R4 R5 R6
169     w_instruction <= B"00000000101001100010000000101010";
        wait for 20 ns;
```

```
171     -- SLTU R7 R8 R9
        w_instruction <= B"00000001000010010011100000101011";
173     wait for 20 ns;
        -- BGEZ R10 20
175     w_instruction <= B"00000101010000010000000000010100";
        wait for 20 ns;
177     -- BLTZ R12 64
        w_instruction <= B"00000101100000000000000001000000";
179     wait for 20 ns;
        -- BEQ R14 R15 128
181     w_instruction <= B"00010001110011100000000010000000";
        wait for 20 ns;
183     -- BNE R16 R17 860
        w_instruction <= B"00010110000100000000001101011100";
185     wait for 20 ns;
        -- BLEZ R18 -240
187     w_instruction <= B"00011010010000001111111100010000";
        wait for 20 ns;
189     -- BGTZ R20 -32
        w_instruction <= B"00011110100000001111111111100000";
191     wait for 20 ns;
        -- ADDI R22 R23 10053
193     w_instruction <= B"00100010110101100010011101000101";
        wait for 20 ns;
195     -- ADDIU R24 R0 32767
        w_instruction <= B"00100100010110000111111111111111";
197     wait for 20 ns;
        -- SLTI R25 R26 25
199     w_instruction <= B"00101011001110010000000000011001";
        wait for 20 ns;
201     -- SLTIU R27 R28 47
        w_instruction <= B"00101111011110110000000000101111";
203     wait for 20 ns;
        -- ANDI R29 R30 30
205     w_instruction <= B"00110011101111010000000000011110";
        wait for 20 ns;
207     -- ORI R31 R0 5
        w_instruction <= B"00110100011111110000000000000101";
209     wait for 20 ns;
        -- XORI R1 R0 127
211     w_instruction <= B"00111000001000010000000001111111";
        wait for 20 ns;
213     -- LUI R2 200
```

```
    w_instruction <= B"00111100000000100000000011001000";
    wait for 20 ns;
    -- LW R4 R5 256
    w_instruction <= B"10001100100001000000000100000000";
    wait for 20 ns;
    -- SW R6 R7 128
    w_instruction <= B"10101100110001100000000010000000";
    wait for 20 ns;
    wait;
  end process stimulus;
end testbench;
```

Listing B.39. Instruction_Decoder_TB.vhd Code

2.19 Encoders

2.19.1 Encoder6_3

```
--| Encoder6_3_TB.vhd
--| Test the Encoder6_3
library IEEE;
use IEEE.std_logic_1164.all;

entity Encoder6_3_TB is
end Encoder6_3_TB;

architecture testbench of Encoder6_3_TB is
--| Declare Components
  component Encoder6_3 is
    port (i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          o_Z : out std_logic_vector(2 downto 0));
  end component;

--| Declare Signals
  signal w_1 : std_logic := '0'; -- Input 1
  signal w_2 : std_logic := '0'; -- Input 2
  signal w_3 : std_logic := '0'; -- Input 3
  signal w_4 : std_logic := '0'; -- Input 4
  signal w_5 : std_logic := '0'; -- Input 5
  signal w_Z : std_logic_vector(2 downto 0); -- Output

begin
  -- Connect the Encoder6_3
  u_Encoder6_3 : Encoder6_3
  port map (
    i_1 => w_1,
    i_2 => w_2,
    i_3 => w_3,
    i_4 => w_4,
    i_5 => w_5,
    o_Z => w_Z);

  stimulus: process is
  begin
```

```
    wait for 20 ns;
42      w_1 <= '1';
    wait for 20 ns;
44      w_1 <= '0';
        w_2 <= '1';
46      wait for 20 ns;
        w_2 <= '0';
48      w_3 <= '1';
        wait for 20 ns;
50      w_3 <= '0';
        w_4 <= '1';
52      wait for 20 ns;
        w_4 <= '0';
54      w_5 <= '1';
        wait for 20 ns;
56      wait;
    end process;
58  end testbench;
```

Listing B.40. Encoder6_3_TB.vhd Code

2.19.2 Encoder9_4

```
--| Encoder9_4_TB.vhd
--| Test the Encoder9_4
library IEEE;
use IEEE.std_logic_1164.all;

entity Encoder9_4_TB is
end Encoder9_4_TB;

architecture testbench of Encoder9_4_TB is
--| Declare Components
  component Encoder9_4 is
    port (i_1 : in  std_logic;
          i_2 : in  std_logic;
          i_3 : in  std_logic;
          i_4 : in  std_logic;
          i_5 : in  std_logic;
          i_6 : in  std_logic;
          i_7 : in  std_logic;
          i_8 : in  std_logic;
          o_Z : out std_logic_vector(3 downto 0));
  end component;

--| Declare Signals
  signal w_1 : std_logic := '0'; -- Input 1
  signal w_2 : std_logic := '0'; -- Input 2
  signal w_3 : std_logic := '0'; -- Input 3
  signal w_4 : std_logic := '0'; -- Input 4
  signal w_5 : std_logic := '0'; -- Input 5
  signal w_6 : std_logic := '0'; -- Input 6
  signal w_7 : std_logic := '0'; -- Input 7
  signal w_8 : std_logic := '0'; -- Input 8
  signal w_Z : std_logic_vector(3 downto 0); -- Output

begin
  -- Connect the Encoder9_4
  u_Encoder9_4 : Encoder9_4
  port map (
    i_1 => w_1,
    i_2 => w_2,
    i_3 => w_3,
    i_4 => w_4,
```

```
42      i_5 => w_5,
        i_6 => w_6,
44      i_7 => w_7,
        i_8 => w_8,
46      o_Z => w_Z);

48    stimulus: process is
      begin
50      wait for 20 ns;
        w_1 <= '1';
52      wait for 20 ns;
        w_1 <= '0';
54      w_2 <= '1';
        wait for 20 ns;
56      w_2 <= '0';
        w_3 <= '1';
58      wait for 20 ns;
        w_3 <= '0';
60      w_4 <= '1';
        wait for 20 ns;
62      w_4 <= '0';
        w_5 <= '1';
64      wait for 20 ns;
        w_5 <= '0';
66      w_6 <= '1';
        wait for 20 ns;
68      w_6 <= '0';
        w_7 <= '1';
70      wait for 20 ns;
        w_7 <= '0';
72      w_8 <= '1';
        wait for 20 ns;
74      wait;
    end process;
76  end testbench;
```

Listing B.41. Encoder9_4_TB.vhd Code

2.20 Instruction Decoder/Encoder

```
--| Instruction_Dec_Enc_TB
--| Tests Instruction_Dec_Enc
library IEEE;
use IEEE.std_logic_1164.all;

entity Instruction_Dec_Enc_TB is
end Instruction_Dec_Enc_TB;

architecture testbench of Instruction_Dec_Enc_TB is
  component Instruction_Dec_Enc is
    port (i_instr31      : in  std_logic;
          i_instr30      : in  std_logic;
          i_instr29      : in  std_logic;
          i_instr28      : in  std_logic;
          i_instr27      : in  std_logic;
          i_instr26      : in  std_logic;
          i_instr20      : in  std_logic;
          i_instr19      : in  std_logic;
          i_instr18      : in  std_logic;
          i_instr17      : in  std_logic;
          i_instr16      : in  std_logic;
          i_instr05      : in  std_logic;
          i_instr04      : in  std_logic;
          i_instr03      : in  std_logic;
          i_instr02      : in  std_logic;
          i_instr01      : in  std_logic;
          i_instr00      : in  std_logic;
          i_BRANCH_OVERRIDE : in  std_logic;
          i_DO_NOT_STORE    : in  std_logic;
          i_PC_STORE        : in  std_logic;
          o_ALU_SRC_A       : out std_logic_vector(1 downto 0);
          o_ALU_SRC_B       : out std_logic_vector(1 downto 0);
          o_ALU_INV_B       : out std_logic;
          o_COMP_SEL        : out std_logic_vector(2 downto 0);
          o_OVER_CTRL       : out std_logic_vector(1 downto 0);
          o_ALU_OUTPUT      : out std_logic_vector(3 downto 0);
          o_REG_SEL      : out std_logic_vector(1 downto 0);
          o_UNSIGNED        : out std_logic;
          o_imm_extend      : out std_logic;
          o_FSM_CTRL        : out std_logic_vector(1 downto 0);
          o_INVALID_INSTR : out std_logic);
```

```
42    end component;

44    -- Declare signals
      signal w_instruction   : std_logic_vector(31 downto 0);
46    signal w_BRANCH_OVERRIDE: std_logic :='0';
      signal w_DO_NOT_STORE : std_logic :='0';
48    signal w_PC_STORE      : std_logic :='0';
      signal w_ALU_SRC_A     : std_logic_vector(1 downto 0);
50    signal w_ALU_SRC_B     : std_logic_vector(1 downto 0);
      signal w_ALU_INV_B     : std_logic;
52    signal w_COMP_SEL      : std_logic_vector(2 downto 0);
      signal w_OVER_CTRL     : std_logic_vector(1 downto 0);
54    signal w_ALU_OUTPUT    : std_logic_vector(3 downto 0);
      signal w_REG_SEL       : std_logic_vector(1 downto 0);
56    signal w_UNSIGNED      : std_logic;
      signal w_imm_extend    : std_logic;
58    signal w_FSM_CTRL      : std_logic_vector(1 downto 0);
      signal w_INVALID_INSTR : std_logic;
60
  begin
62    -- Connect Instruction_Dec_Enc
      u_Instruction_Dec_Enc: Instruction_Dec_Enc
64    port map (i_instr31 => w_instruction(31),
              i_instr30 => w_instruction(30),
66            i_instr29 => w_instruction(29),
              i_instr28 => w_instruction(28),
68            i_instr27 => w_instruction(27),
              i_instr26 => w_instruction(26),
70            i_instr20 => w_instruction(20),
              i_instr19 => w_instruction(19),
72            i_instr18 => w_instruction(18),
              i_instr17 => w_instruction(17),
74            i_instr16 => w_instruction(16),
              i_instr05 => w_instruction(5),
76            i_instr04 => w_instruction(4),
              i_instr03 => w_instruction(3),
78            i_instr02 => w_instruction(2),
              i_instr01 => w_instruction(1),
80            i_instr00 => w_instruction(0),
             i_BRANCH_OVERRIDE => w_BRANCH_OVERRIDE,
82           i_DO_NOT_STORE => w_DO_NOT_STORE,
             i_PC_STORE => w_PC_STORE,
84           o_ALU_SRC_A => w_ALU_SRC_A,
```

```
            o_ALU_SRC_B => w_ALU_SRC_B,
            o_ALU_INV_B => w_ALU_INV_B,
            o_COMP_SEL => w_COMP_SEL,
            o_OVER_CTRL => w_OVER_CTRL,
            o_ALU_OUTPUT => w_ALU_OUTPUT,
            o_REG_SEL => w_REG_SEL,
            o_UNSIGNED => w_UNSIGNED,
            o_imm_extend => w_imm_extend,
            o_FSM_CTRL => w_FSM_CTRL,
            o_INVALID_INSTR => w_INVALID_INSTR);

    -- Process for stimulating inputs
    stimulus: process is
    begin
      -- SLL R2, R1, 5
      w_instruction <= B"00000000000000010001000101000000";
      wait for 20 ns;
      -- NOP
      w_instruction <= B"00000000000000000000000000000000";
      wait for 20 ns;
      -- SRL R3, R2, 7
      w_instruction <= B"00000000000000100001100111000010";
      wait for 20 ns;
      -- SRA R5 R4 15
      w_instruction <= B"00000000000001000010101111000011";
      wait for 20 ns;
      -- SLLV R3 R2 R1
      w_instruction <= B"00000000010001000011000000000100";
      wait for 20 ns;
      -- SRLV R7 R6 R8
      w_instruction <= B"00000010000011000111000000000110";
      wait for 20 ns;
      -- SRAV R9 R10 R11
      w_instruction <= B"00000010110101001001000000000111";
      wait for 20 ns;
      -- ADD R12 R13 R14
      w_instruction <= B"00000011010111001100000000100000";
      wait for 20 ns;
      -- ADDU R15 R16 R17
      w_instruction <= B"00000100001000101111000001000001";
      wait for 20 ns;
      -- SUB R18 R19 R20
      w_instruction <= B"00000100111010010010000001000010";
```

```
128     wait for 20 ns;
        -- SUBU R21 R22 R23
130     w_instruction <= B"00000010110101111010100000100011";
        wait for 20 ns;
132     -- AND R24 R25 R26
        w_instruction <= B"00000011001110101100000000100100";
134     wait for 20 ns;
        -- OR R27 R28 R29
136     w_instruction <= B"00000011101111001101100000100101";
        wait for 20 ns;
138     -- XOR R30 R31 R0
        w_instruction <= B"00000011111000001111000000100110";
140     wait for 20 ns;
        -- NOR R1 R2 R3
142     w_instruction <= B"00000000010000110000100000100111";
        wait for 20 ns;
144     -- SLT R4 R5 R6
        w_instruction <= B"00000000101001100010000000101010";
146     wait for 20 ns;
        -- SLTU R7 R8 R9
148     w_instruction <= B"00000001000010010011100000101011";
        wait for 20 ns;
150     -- BGEZ R10 20
        w_instruction <= B"00000101010000010000000000010100";
152     wait for 20 ns;
        w_BRANCH_OVERRIDE <= '1';
154     wait for 20 ns;
        -- BLTZ R12 64
156     w_instruction <= B"00000101100000000000000001000000";
        w_BRANCH_OVERRIDE <= '0';
158     wait for 20 ns;
        w_BRANCH_OVERRIDE <= '1';
160     wait for 20 ns;
        -- BEQ R14 R15 128
162     w_instruction <= B"00010001110011100000000010000000";
        w_BRANCH_OVERRIDE <= '0';
164     wait for 20 ns;
        w_BRANCH_OVERRIDE <= '1';
166     wait for 20 ns;
        -- BNE R16 R17 860
168     w_instruction <= B"00010110000100000000001101011100";
        w_BRANCH_OVERRIDE <= '0';
170     wait for 20 ns;
```

```
    w_BRANCH_OVERRIDE <= '1';
    wait for 20 ns;
    -- BLEZ R18 -240
    w_instruction <= B"00011010010000001111111100010000";
    w_BRANCH_OVERRIDE <= '0';
    wait for 20 ns;
    w_BRANCH_OVERRIDE <= '1';
    wait for 20 ns;
    -- BGTZ R20 -32
    w_instruction <= B"00011110100000001111111111100000";
    w_BRANCH_OVERRIDE <= '0';
    wait for 20 ns;
    w_BRANCH_OVERRIDE <= '1';
    wait for 20 ns;
    -- ADDI R22 R23 10053
    w_instruction <= B"00100010110101100010011101000101";
    w_BRANCH_OVERRIDE <= '0';
    wait for 20 ns;
    -- ADDIU R24 R0 32767
    w_instruction <= B"00100100010110000111111111111111";
    wait for 20 ns;
    -- SLTI R25 R26 25
    w_instruction <= B"00101011001110010000000000011001";
    wait for 20 ns;
    -- SLTIU R27 R28 47
    w_instruction <= B"00101111011110110000000000101111";
    wait for 20 ns;
    -- ANDI R29 R30 30
    w_instruction <= B"00110011101111010000000000011110";
    wait for 20 ns;
    -- ORI R31 R0 5
    w_instruction <= B"00110100011111110000000000000101";
    wait for 20 ns;
    -- XORI R1 R0 127
    w_instruction <= B"00111000001000010000000001111111";
    wait for 20 ns;
    -- LUI R2 200
    w_instruction <= B"00111100000000100000000011001000";
    wait for 20 ns;
    -- LW R4 R5 256
    w_instruction <= B"10001100100001000000000100000000";
    w_DO_NOT_STORE <= '1';
    wait for 20 ns;
```

```
214     w_DO_NOT_STORE <= '0';
        wait for 20 ns;
216     -- SW R6 R7 128
        w_instruction <= B"10101100110001100000000010000000";
218     wait for 20 ns;
        wait;
220   end process stimulus;
end testbench;
```

Listing B.42. Instruction_Dec_Enc_TB.vhd Code

2.21 Finite State Machine Control Signal Generator

```
--| FSM_Control_Signal_Generator_TB
--| Tests FSM_Control_Signal_Generator
library IEEE;
use IEEE.std_logic_1164.all;

entity FSM_Control_Signal_Generator_TB is
end FSM_Control_Signal_Generator_TB;

architecture testbench of FSM_Control_Signal_Generator_TB is
  component FSM_Control_Signal_Generator is
    port (i_state          : in  std_logic_vector(3 downto 0);
          o_MEM_READ       : out std_logic;
          o_MEM_WRITE      : out std_logic;
          o_MEM_ADDRESS_SEL   : out std_logic;
          o_STORE_FROM_MEM    : out std_logic;
          o_PC_STORE       : out std_logic;
          o_PC_EN          : out std_logic;
          o_DO_NOT_STORE      : out std_logic;
          o_BRANCH_OVERRIDE   : out std_logic;
          o_STORE_INSTRUCTION : out std_logic);
  end component;

  -- Declare signals
  signal w_state         : std_logic_vector(3 downto 0) := (
   others => '0');
  signal w_MEM_READ      : std_logic;
  signal w_MEM_WRITE     : std_logic;
  signal w_MEM_ADDRESS_SEL  : std_logic;
  signal w_STORE_FROM_MEM   : std_logic;
  signal w_PC_STORE      : std_logic;
  signal w_PC_EN         : std_logic;
  signal w_DO_NOT_STORE  : std_logic;
  signal w_BRANCH_OVERRIDE  : std_logic;
  signal w_STORE_INSTRUCTION  : std_logic;


begin
  -- Connect myNAND2
  u_FSM_Control_Signal_Generator:
   FSM_Control_Signal_Generator
  port map (i_state => w_state,
```

```
            o_MEM_READ => w_MEM_READ,
            o_MEM_WRITE => w_MEM_WRITE,
            o_MEM_ADDRESS_SEL => w_MEM_ADDRESS_SEL,
            o_STORE_FROM_MEM => w_STORE_FROM_MEM,
            o_PC_STORE => w_PC_STORE,
            o_PC_EN => w_PC_EN,
            o_DO_NOT_STORE => w_DO_NOT_STORE,
            o_BRANCH_OVERRIDE => w_BRANCH_OVERRIDE,
            o_STORE_INSTRUCTION => w_STORE_INSTRUCTION);

-- Process for stimulating inputs
stimulus: process is
begin
  -- State 0
  wait for 20 ns;
  -- State 1
  w_state <= B"0001";
  wait for 20 ns;
  -- State 2
  w_state <= B"0010";
  wait for 20 ns;
  -- State 3
  w_state <= B"0011";
  wait for 20 ns;
  -- State 4
  w_state <= B"0100";
  wait for 20 ns;
  -- State 5
  w_state <= B"0101";
  wait for 20 ns;
  -- State 6
  w_state <= B"0110";
  wait for 20 ns;
  -- State 7
  w_state <= B"0111";
  wait for 20 ns;
  -- State 8
  w_state <= B"1000";
  wait for 20 ns;
  -- State 9
  w_state <= B"1001";
  wait for 20 ns;
  -- State 10
```

```
83      w_state <= B"1010";
        wait for 20 ns;
85      -- State 11
        w_state <= B"1011";
87      wait for 20 ns;
        -- State 12
89      w_state <= B"1100";
        wait for 20 ns;
91      -- State 13
        w_state <= B"1101";
93      wait for 20 ns;
        -- State 14
95      w_state <= B"1110";
        wait for 20 ns;
97      -- State 15
        w_state <= B"1111";
99      wait for 20 ns;
        wait;
101   end process stimulus;
end testbench;
```

Listing B.43. FSM_Control_Signal_Generator_TB.vhd Code

2.22 Memory Emulator

```
--| MEM_EMULATOR.vhd
--| Emulate the behaviour of a 32-bit addressable memory. Does not
--| really use incoming address to access a memory location. Instead,
--| it simply counts the number of times i_read_enable is '1'. Each
--| time i_read_enable is '1', the next instruction to be simulated
--| on the FSM/MIPS is provided on o_data. For write operations,
--| a single internal register will retain the value of i_data
--| until it is overwritten by the next write operation. Write operations
--| occur when i_write_enable is '1'.
--|
--| INPUTS:
--| i_clk - clock input
--| i_address - address
--| i_read_enable - enable read
--| i_write_enable - enable write
--| i_data - input data
--|
--| OUTPUTS:
--| o_data - output data
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MEM_EMULATOR is
  port (i_clk        : in  std_logic;
        i_reset      : in  std_logic;
        i_address    : in  std_logic_vector(31 downto 0);
        i_read_enable : in  std_logic;
        i_write_enable  : in  std_logic;
        i_data       : in  std_logic_vector(31 downto 0);
        o_data       : out std_logic_vector(31 downto 0);
        o_written_data  : out std_logic_vector(31 downto 0);
        o_MEM_READY    : out std_logic);
end MEM_EMULATOR;
```

```
36 architecture a_MEM_EMULATOR of MEM_EMULATOR is

38   component Enabled_Register is
       port (i_clk    : in  std_logic;
40           i_reset  : in  std_logic;
             i_data   : in  std_logic_vector(31 downto 0);
42           i_en     : in  std_logic;
             o_Q      : out std_logic_vector(31 downto 0));
44   end component;

46   signal f_read_count : unsigned(9 downto 0) := (others =>
      '0');
     signal f_MEM_READY : std_logic := '0';
48   signal f_read : std_logic := '0';
     signal f_write : std_logic := '0';
50 begin

52   o_MEM_READY <= f_MEM_READY;

54   -- Connect write register
     u_Write_Register: Enabled_Register
56   port map (i_clk => i_clk,
             i_reset => i_reset,
58           i_data => i_data,
             i_en => i_write_enable,
60           o_Q => o_written_data);

62   mem_read_process : process (i_clk, i_reset, i_read_enable,
      f_read_count)
     begin
64     if i_reset = '1' then
         f_read_count <= (others => '0');
66     elsif rising_edge(i_clk) then
         f_read <= i_read_enable;
68       f_write <= i_write_enable;
         if i_read_enable = '1' then
70         if f_read = '0' then
             f_MEM_READY <= '1';
72           case to_integer(f_read_count) is
               when 0 =>
74               -- LUI R1 16
                 o_data <= B"00111100000000010000000000010000";
```

```
76              when 1 =>
                  -- SRL R1 R1 16
78                o_data <= B"00000000000000010000110000000010";
                when 2 =>
80                -- NOP
                  o_data <= B"00000000000000000000000000000000";
82              when 3 =>
                  -- LUI R2 12
84                o_data <= B"00111100000000100000000000001100";
                when 4 =>
86                -- SRLV R2 R2 R1
                  o_data <= B"00000000001000100001000000000110";
88              when 5 =>
                  -- LUI R3 -23
90                o_data <= B"00111100000000111111111111101001";
                when 6 =>
92                -- SRAV R4 R3 R1
                  o_data <= B"00000000001000110010000000000111";
94              when 7 =>
                  -- SRA R3 R3 16
96                o_data <= B"00000000000000110001110000000011";
                when 8 =>
98                -- SLL R5 R2 1
                  o_data <= B"00000000000000100010100001000000";
100             when 9 =>
                  -- SLLV R6 R5 R1
102               o_data <= B"00000000001001010011000000000100";
                when 10 =>
104               -- ADD R4 R1 R3
                  o_data <= B"00000000001000110010000000100000";
106             when 11 =>
                  -- ADDU R7 R1 R2
108               o_data <= B"00000000001000100011100000100001";
                when 12 =>
110               -- SUB R8 R3 R5
                  o_data <= B"00000000011001010100000000100010";
112             when 13 =>
                  -- SUBU R9  R7 R8
114               o_data <= B"00000000111010000100100000100011";
                when 14 =>
116               -- AND R10 R9 R3
                  o_data <= B"00000001001000110101000000100100";
118             when 15 =>
```

```
        -- OR R11 R1 R2
120     o_data <= B"00000000001000100101100000100101";
      when 16 =>
122     -- XOR R12 R2 R5
        o_data <= B"00000000010001010110000000100110";
124   when 17 =>
        -- NOR R13 R1 R2
126     o_data <= B"00000000001000100110100000100111";
      when 18 =>
128     -- SLT R14 R8 R1
        o_data <= B"00000001000000010111000000101010";
130   when 19 =>
        -- SLT R14 R1 R2
132     o_data <= B"00000000001000100111000000101010";
      when 20 =>
134     -- SLT R14 R2 R1
        o_data <= B"00000000010000010111000000101010";
136   when 21 =>
        -- SLTU R15 R1 R4
138     o_data <= B"00000000001001000111100000101011";
      when 22 =>
140     -- BGEZ R1 8
        o_data <= B"00000100001000010000000000001000";
142   when 23 =>
        -- BLTZ R2 24
144     o_data <= B"00000100010000000000000000011000";
      when 24 =>
146     -- BEQ R0 RO -6
        o_data <= B"00010000000000001111111111111010";
148   when 25 =>
        -- BNE R0 R0 37
150     o_data <= B"00010100000000000000000000100101";
      when 26 =>
152     -- BLEZ R8 12
        o_data <= B"00011001000000000000000000001100";
154   when 27 =>
        -- BGTZ R4 23
156     o_data <= B"00011100100000000000000000010111";
      when 28 =>
158     -- ADDI R16 R12 -5
        o_data <= B"00100001100100001111111111111011";
160   when 29 =>
        -- ADDIU R17 R9 -1
```

```
162                 o_data <= B"00100101001100011111111111111111";
                  when 30 =>
164                 -- SLTI R18 R13 -17
                    o_data <= B"00101001101100101111111111101111";
166               when 31 =>
                    -- SLTIU R19 R13 -2
168                 o_data <= B"00101101101100111111111111111110";
                  when 32 =>
170                 -- ANDI R20 R9 25
                    o_data <= B"00110001001101000000000000011001";
172               when 33 =>
                    -- ORI R21 R9 25
174                 o_data <= B"00110101001101010000000000011001";
                  when 34 =>
176                 -- XORI R22 R9 25
                    o_data <= B"00111001001101100000000000011001";
178               when 35 =>
                    -- LW R23 R1 1024
180                 o_data <= B"10001100001101110000010000000000";
                  when 36 =>
182                 -- Data to send for load word command
                    o_data <= B"00001111000011110000111100001111";
184               when 37 =>
                    -- SW R8 R1 2048
186                 o_data <= B"10101100001010000000100000000000";
                  when others =>
188                 -- NOP
                    o_data <= B"00000000000000000000000000000000";
190             end case;
                f_read_count <= f_read_count + 1;
192           end if;
          elsif i_write_enable = '1' then
194           if f_write = '0' then
                f_MEM_READY <= '1';
196           end if;
          else
198           f_MEM_READY <= '0';
        end if;
200     end if;
    end process;
202
end a_MEM_EMULATOR;
```

Listing B.44. MEM_EMULATOR.vhd Code

2.23 Controller Finite State Machine (FSM)

```
--| Controller_FSM_TB.vhd
--| Test the functionality of the Controller_FSM using the MEM_EMULATOR
--| so that the controller can "read" instructions from memory, transition
--| through the appropriate states, and send the proper control signals out
library IEEE;
use IEEE.std_logic_1164.all;

entity Controller_FSM_TB is
end Controller_FSM_TB;

architecture testbench of Controller_FSM_TB is
--| Define Components
  --| Controller Finite State Machine
  component Controller_FSM is
    port (i_clk          : in  std_logic;
          i_reset        : in  std_logic;
          i_MEM_OUT      : in  std_logic_vector(31 downto 0);
          i_MEM_READY    : in std_logic;
          o_MEM_READ     : out std_logic;
          o_MEM_WRITE    : out std_logic;
          o_MEM_ADDRESS_SEL : out std_logic;
          o_STORE_FROM_MEM  : out std_logic;
          o_PC_EN        : out std_logic;
          o_ALU_SRC_A    : out std_logic_vector(1 downto 0);
          o_ALU_SRC_B    : out std_logic_vector(1 downto 0);
          o_ALU_INV_B    : out std_logic;
          o_COMP_SEL     : out std_logic_vector(2 downto 0);
          o_OVER_CTRL    : out std_logic_vector(1 downto 0);
          o_ALU_OUTPUT   : out std_logic_vector(3 downto 0);
          o_REG_SEL      : out std_logic_vector(1 downto 0);
          o_UNSIGNED     : out std_logic;
          o_imm_extend   : out std_logic;
          o_RS_SEL       : out std_logic_vector(4 downto 0);
          o_RT_SEL       : out std_logic_vector(4 downto 0);
          o_immediate    : out std_logic_vector(15 downto 0));
  end component;

  --| Memory Emulator
```

```
39    component MEM_EMULATOR is
        port (i_clk        : in  std_logic;
41            i_reset      : in  std_logic;
              i_address    : in  std_logic_vector(31 downto 0);
43            i_read_enable : in  std_logic;
              i_write_enable  : in  std_logic;
45            i_data       : in  std_logic_vector(31 downto 0);
              o_data       : out std_logic_vector(31 downto 0);
47            o_written_data  : out std_logic_vector(31 downto 0);
             o_MEM_READY    : out std_logic);
49    end component;

51  --| Define signals
    signal c_clk           : std_logic := '0';
53  signal w_reset           : std_logic := '1';
    signal w_MEM_OUT         : std_logic_vector(31 downto 0);
55  signal w_MEM_READY       : std_logic;
    signal w_MEM_IN        : std_logic_vector(31 downto 0) := B"
     11110000111100001111000011110000";
57  signal w_MEM_READ        : std_logic;
    signal w_MEM_WRITE       : std_logic;
59  signal w_MEM_ADDRESS_SEL   : std_logic;
    signal w_STORE_FROM_MEM    : std_logic;
61  signal w_PC_EN           : std_logic;
    signal w_ALU_SRC_A       : std_logic_vector(1 downto 0);
63  signal w_ALU_SRC_B       : std_logic_vector(1 downto 0);
    signal w_ALU_INV_B       : std_logic;
65  signal w_COMP_SEL        : std_logic_vector(2 downto 0);
    signal w_OVER_CTRL       : std_logic_vector(1 downto 0);
67  signal w_ALU_OUTPUT      : std_logic_vector(3 downto 0);
    signal w_REG_SEL         : std_logic_vector(1 downto 0);
69  signal w_UNSIGNED        : std_logic;
    signal w_imm_extend      : std_logic;
71  signal w_RS_SEL        : std_logic_vector(4 downto 0);
    signal w_RT_SEL        : std_logic_vector(4 downto 0);
73  signal w_immediate       : std_logic_vector(15 downto 0);
    signal w_written_data    : std_logic_vector(31 downto 0);
75  signal w_address         : std_logic_vector(31 downto 0) :=
     (others => '0');

77  begin
    u_Controller_FSM : Controller_FSM
79  port map (i_clk => c_clk,
```

```
          i_reset => w_reset,
81        i_MEM_OUT => w_MEM_OUT,
          i_MEM_READY => w_MEM_READY,
83        o_MEM_READ => w_MEM_READ,
          o_MEM_WRITE => w_MEM_WRITE,
85        o_MEM_ADDRESS_SEL => w_MEM_ADDRESS_SEL,
          o_STORE_FROM_MEM => w_STORE_FROM_MEM,
87        o_PC_EN => w_PC_EN,
          o_ALU_SRC_A => w_ALU_SRC_A,
89        o_ALU_SRC_B => w_ALU_SRC_B,
          o_ALU_INV_B => w_ALU_INV_B,
91        o_COMP_SEL => w_COMP_SEL,
          o_OVER_CTRL => w_OVER_CTRL,
93        o_ALU_OUTPUT => w_ALU_OUTPUT,
          o_REG_SEL => w_REG_SEL,
95        o_UNSIGNED => w_UNSIGNED,
          o_imm_extend => w_imm_extend,
97        o_RS_SEL => w_RS_SEL,
          o_RT_SEL => w_RT_SEL,
99        o_immediate => w_immediate);

101   u_MEM_EMULATOR : MEM_EMULATOR
    port map (i_clk => c_clk,
103         i_reset => w_reset,
          i_address => w_address,
105       i_read_enable => w_MEM_READ,
          i_write_enable => w_MEM_WRITE,
107       i_data => w_MEM_IN,
          o_data => w_MEM_OUT,
109       o_written_data => w_written_data,
          o_MEM_READY => w_MEM_READY);
111
    --| Generate the stimulus
113 stimulus : process is
     begin
115   wait for 100 ns;
      w_reset <= '0';
117  end process;

119 --| Generate the clock signal
    clock_gen: process is
121 begin
      c_clk <= '1' after 10 ns, '0' after 20 ns;
```

```
123     wait for 20 ns;
    end process clock_gen;
125 end testbench;
```

Listing B.45. Controller_FSM_TB.vhd Code

2.24 Basic MIPS Top Level

```
--| Basic_MIPS_TB.vhd
--| Test the functionality of the Basic_MIPS using the
   MEM_EMULATOR
--| so that the controller can "read" instructions from
   memory, transition
--| through the appropriate states, and send the proper
   control signals out
library IEEE;
use IEEE.std_logic_1164.all;

entity Basic_MIPS_TB is
end Basic_MIPS_TB;

architecture testbench of Basic_MIPS_TB is
--| Define Components
  --| Controller Finite State Machine
  component Basic_MIPS is
    port (i_clk          : in  std_logic;
          i_reset        : in  std_logic;
          i_MEM_OUT      : in  std_logic_vector(31 downto 0);
          i_MEM_READY    : in  std_logic;
          o_MEM_READ     : out std_logic;
          o_MEM_WRITE    : out std_logic;
          o_MEM_IN       : out std_logic_vector(31 downto 0);
          o_MEM_ADDRESS  : out std_logic_vector(31 downto 0));
  end component;

  --| Memory Emulator
  component MEM_EMULATOR is
    port (i_clk          : in  std_logic;
          i_reset        : in  std_logic;
          i_address      : in  std_logic_vector(31 downto 0);
          i_read_enable  : in  std_logic;
          i_write_enable : in  std_logic;
          i_data         : in  std_logic_vector(31 downto 0);
          o_data         : out std_logic_vector(31 downto 0);
          o_written_data : out std_logic_vector(31 downto 0);
          o_MEM_READY    : out std_logic);
  end component;

--| Define signals
```

```
39    signal c_clk           : std_logic := '0';
      signal w_reset           : std_logic := '1';
41    signal w_MEM_OUT         : std_logic_vector(31 downto 0);
      signal w_MEM_READY       : std_logic;
43    signal w_MEM_IN        : std_logic_vector(31 downto 0);
      signal w_MEM_READ        : std_logic;
45    signal w_MEM_WRITE       : std_logic;
      signal w_written_data    : std_logic_vector(31 downto 0);
47    signal w_address         : std_logic_vector(31 downto 0);

49  begin
      u_Basic_MIPS : Basic_MIPS
51    port map (i_clk => c_clk,
              i_reset => w_reset,
53            i_MEM_OUT => w_MEM_OUT,
              i_MEM_READY => w_MEM_READY,
55            o_MEM_READ => w_MEM_READ,
              o_MEM_WRITE => w_MEM_WRITE,
57            o_MEM_IN => w_MEM_IN,
              o_MEM_ADDRESS => w_address);
59
     u_MEM_EMULATOR : MEM_EMULATOR
61    port map (i_clk => c_clk,
              i_reset => w_reset,
63            i_address => w_address,
              i_read_enable => w_MEM_READ,
65            i_write_enable => w_MEM_WRITE,
              i_data => w_MEM_IN,
67            o_data => w_MEM_OUT,
              o_written_data => w_written_data,
69            o_MEM_READY => w_MEM_READY);

71    --| Generate the stimulus
      stimulus : process is
73     begin
        wait for 100 ns;
75      w_reset <= '0';
      end process;
77
     --| Generate the clock signal
79    clock_gen: process is
      begin
81      c_clk <= '1' after 10 ns, '0' after 20 ns;
```

```
    wait for 20 ns;
  end process clock_gen;
end testbench;
```

Listing B.46. BASIC_MIPS_TB.vhd Code

Appendix C. Version History

- Version 1.4
  - Converted document to AFIT Report Format
- Version 1.3
  - Deleted Future Work section.
- Version 1.2
  - Updated document to include hyperlinks in the Table of Contents to the various sections of the document.
  - Updated block diagram figures so they are computer graphics rather than hand drawn.
- Version 1.1
  - Updated "Comparator" to account for less than comparisons depending on whether the instruction calling for the comparison was an SLT, SLTU, SLTI, or an SLTIU as opposed to BLTZ. Added a control signal to the "FSM_Control_Signal_Generator" which is used by the "Comparator" to differentiate between these instructions.
  - Updated "Controller_FSM" and "FSM_Control_Signal_Generator" to correct timing problems and handshaking signal issues discovered when implementing TMR MIPS. Allows the interface between Basic MIPS and the "TMR_Voter" to communicate correctly without affecting the ability of Basic MIPS to communicate correctly with memory in TSR MIPS. Controller states 1, 7, and 9 now require the i_MEM_READY signal to return to '0' prior to transitioning to the next state. States 7 and 9 now transition

to a new state, 10, instead of 0. The control signal generator was modified so that the o_PC_EN and o_DO_NOT_STORE signals match up with the new state transition method.

- Version 1.0
  - Original Document.

Bibliography

1. I. MIPS Technologies, *$MIPS32^{TM}$ Architecture for Programmers Volume I: Introduction to the $MIPS32^{TM}$ Architecture*, 1st ed., 1225 Charleston Road, Mountain View, CA 94043-1353, mar 2001.

2. ——, *$MIPS32^{TM}$ Architecture for Programmers Volume II: The $MIPS32^{TM}$ Instruction Set*, 1st ed., 1225 Charleston Road, Mountain View, CA 94043-1353, mar 2001.

3. N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective Fourth Edition.* Addison Wesley, 2011.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
12–09–2019	Technical Report	Sept 2016 — Sept 2019

4. TITLE AND SUBTITLE

Basic MIPS Architecture Version 1.4

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hamilton, Nicolas S, Maj, USAF

5d. PROJECT NUMBER

18G169C

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering an Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/EN/TR-19-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Undisclosed

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes in detail the architecture of a MIPS-like processor used for Adaptive-Hybrid Redundancy research. There may be many other applications for this MIPS-like processor beyond this specific research area.

15. SUBJECT TERMS

MIPS, Processor, Adaptive-Hybrid Redundancy

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Nicolas Hamilton, AFIT/ENG
U	U	U	UU	709	19b. TELEPHONE NUMBER *(include area code)* (937) 255-6565 x4220; nicolas.namilton@afit.edu