Performance, Resources, and Complexity: A Systematic Approach to Microarchitectural Design

By

Shing Ip Kong

B.S. (Washington University, St. Louis) 1982
M.S. (University of California) 1985

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ENGINEERING
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: D. A. Patterson 5/8/89
Chair . . . . Date

David G. Hodges 5/1/89

[illegible] 5/11/89

****

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **MAY 1989**	2. REPORT TYPE	3. DATES COVERED **00-00-1989 to 00-00-1989**
4. TITLE AND SUBTITLE **Performance, Resources, and Complexity: A Systematic Approach to Microarchitectural Design**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
VLSI design in general -- microprocessor design in particular -- has been treated more like an art than a science in the past. The goal of this thesis is to explain the science of VLSI design to someone who wants to build a microprocessor. This can be accomplished by providing a quantitative way to evaluate, and a systematic approach to design, a microprocessor. Resources and complexity are two separate ways a microprocessor designer can pay for performance. The microprocessor designer must evaluate the performance, resources, and complexity tradeoffs quantitatively. In this thesis, the SPUR (SPUR stands for Symbolic Processing Using RISC Machines) CPU microarchitecture is used as example to show how performance, resources, and complexity tradeoffs can be evaluated quantitatively. A systematic approach to microarchitectural design is then developed based on the SPUR CPU design experience. The SPUR CPU is implemented in 1.6um, double layer metal, CMOS technology. It consists of 115,000 transistors, runs at 100ns, and consumes 0.8W of power. Important features of the SPUR CPU are: an internal instruction cache; a four-stage pipeline; support for LISP; a cache controller interface for multiprocessing and virtual memory support; and a parallel coprocessor interface for floating point arithmetic support. All these features make the SPUR CPU significantly different and more complex than previous generations of Berkeley RISC machines.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**236**	

Performance, Resources, and Complexity: A Systematic Approach to Microarchitectural Design

Shing Ip Kong

Abstract

VLSI design in general–microprocessor design in particular–has been treated more like an art than a science in the past. The goal of this thesis is to explain the science of VLSI design to someone who wants to build a microprocessor. This can be accomplished by providing a quantitative way to evaluate, and a systematic approach to design, a microprocessor. Resources and complexity are two separate ways a microprocessor designer can pay for performance. The microprocessor designer must evaluate the performance, resources, and complexity tradeoffs quantitatively. In this thesis, the SPUR (SPUR stands for Symbolic Processing Using RISC Machines) CPU microarchitecure is used as example to show how performance, resources, complexity tradeoffs can be evaluated quantitatively. A systematic approach to microarchitectural design is then developed based on the SPUR CPU design experience. The SPUR CPU is implemented in 1.6μm, double layer metal, CMOS technology. It consists of 115,000 transistors, runs at 100ns, and consumes 0.8W of power. Important features of the SPUR CPU are: an internal instruction cache; a four-stage pipeline; support for LISP; a cache controller interface for multiprocessing and virtual memory support; and a parallel coprocessor interface for floating point arithmetic support. All these features make the SPUR CPU significantly different and more complex than previous generations of Berkeley RISC machines.

D. A. Patterson

Dedicated to my parents,

who gave so much for my education but

unfortunately did not live long enough to share this moment.

Acknowledgements

First I must thank my advisors Professor Patterson, Professor Hodges, and Professor Leachman. This thesis would have been impossible without their advice and signatures. I would also like to thank all the SPUR men and women who are mentioned in Section 6.1.2. In particular, I would like to thank Professor Patterson for his great leadership, Dave Lee, Wook Koh, and Rich Duncombe who worked closely with me on the SPUR CPU, and David Wood and Paul Hansen who worked closely with me on the Cache Controller and Floating Point Unit interfaces.

I must also thank several people in the Berkeley EECS department. Professor Randy Katz, Professor John Ousterhout, and Mark Hill gave me much advice. My office mates and good friends Mike Nelson, Andrew Cherenson, Brent Welch, John Hartman, and Bob Bruce who I must say are great guys despite being "software engineers." I also benefited greatly from the experiences of Manolis Katevenis, Robert Sherburne, David Ungar, Joan Pendleton and all those who made contributions to the first and second generations of Berkeley RISC machines.

Despite all rumors, EECS students do have family and friends outside the department. I want to thank my two sisters Wendy and Janny who give me constant support. My wild and crazy friends at Mary Morse Hall and Mills College gave me great memories and headaches over the years. My fellow bird men and women at the U.C. flying club gave me another view of the world other than 1s and 0s. Last but not least, I must also thank two of my former professors at Washington University. Professor Kline taught my first electrical engineering class. Professor Rosenberger offered me my first electrical engineering job in Computer System Laboratory and introduced me to the fascinating world of VLSI.

Principal funding for the SPUR project was provided by the DARPA under contract N00039-85-C-0269. Additional support was provided by the California MICRO program, a DEC CAD/CAM grant, by the NSF under grant DCR-8202591, by equipment donations from TI, Inc,. and by computer resources provided under DARPA contract N00039-84-C-0089.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

What is hard to get across is the tremendous speed at which things are changing in this business.
Dave Patterson, New York Times, 1988

1.1. The Berkeley Tradition

The history of VLSI chip projects at Berkeley is shown in Table1-1-1. This table also illustrates the evolution of VLSI projects in the research environment because what happened in Berkeley was very typical. In the early 1980s, the Mead and Conway design style enabled us to build 40,000+ transistor (large for the time) NMOS VLSI microprocessors such as RISC I [Pat82], RISC II [Kat83], and SOAR [Ung84]. Fueled by these successes and further advances in technology, we started the SPUR project in 1985 after we completed the SOAR project. The SPUR project's ambitious goal is to build a multiprocessor workstation system [Hil86].

SPUR stands for Symbolic Processing Using RISC machines. Figure 1-1-1(a) is a block diagram of the SPUR multiprocessor. SPUR is a shared-bus multiprocessor consists of 6 to 12 identical high-performance processors. These processors are connected to each other, to standard shared memory, and to input/output devices with a modified TI Nu-Bus which we called the SPUR Bus [Gib87]. Figure 1-1-1(b) is an expanded view of the SPUR processor board. Each SPUR processor contains a 128K-byte cache to reduce the bandwidth required from the bus and the shared memory. Each SPUR processor is implemented on a single board with about 200 standard chips and three custom CMOS chips: the Cache Controller (CC), the Floating Point Unit

Project	Cycle Time (ns)	Transistor Count	Process (μm)	Pin Count	Design Effort (man year)	Generation
1981 RISC I	800	44K	4-NMOS	62	2.3	1st
1983 RISC II Chip A Chip B	500 330	41K	4-NMOS 3-NMOS	62	2.8	
1985 SOAR	400	36K	4-NMOS	84	3.3	2nd
1988 SPUR CPU CC FPU	100	120K 68K 100K	1.6-CMOS	208	4.5	3rd

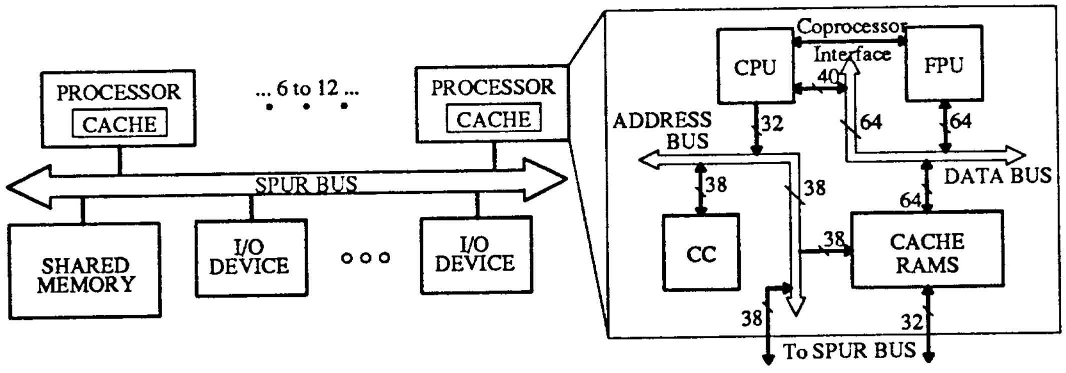
Table 1-1-1 The Berkeley Tradition

(FPU), and the Central Processing Unit (CPU).

The Cache Controller handles cache accesses, performs address translation [Woo86], accesses shared memory over the shared bus, and maintains cache consistency [KEW85]. The Floating Point Unit [BPT87] supports the IEEE standard for binary floating-point arithmetic. Finally, the CPU is based on the Berkeley RISC architecture. The SPUR CPU, however, is different from RISC II because it has a 512-byte internal instruction cache, a longer pipeline, a coprocessor interface, and support for LISP. These three custom VLSI chips are implemented in a 1.6μm double layer metal CMOS technology and each consists of approximately 100,000 transistors.

1.2. Research Motivation

The research reported in this thesis is motivated by the design of the SPUR CPU. Microprocessor design is influenced by many different issues, and their effects were studied by Katevenis in 1983 [Kat83]. Since then, however, many old design issues have changed and many new design issues have emerged due to influences from four areas:



(a) SPUR Workstation Basic Architecture (b) SPUR Processor Board

Figure 1-1-1 The SPUR Workstation

The SPUR multiprocessor workstation is a shared-bus multiprocessor which consists of 6 to 12 identical high-performance custom processors. Each processor contains three custom VLSI CMOS chips: the CPU, the CC, and the FPU. These three chips are connected by a 38-bit address bus, a 64-bit data bus, and the CPU and FPU are connected by a parallel coprocessor interface. The CPU only uses the lower 32 bits of the address bus and the lower 40 bits of the data bus.

System

There is demand for more support for coprocessors, memory management, multiple processors, and operating systems.

Software

There is demand for more support for specialized languages. Better compiler technology is also available for better hardware-software trade-offs.

Simulation

The higher demands in the system and software areas increase the popularity of multiple chip projects such as Berkeley's SPUR, XEROX's Dragon [MoS85], DEC's Firefly [TSJ88], and HP's Spectrum project [BiW86]. A project spanning multiple chip designs requires significantly more detailed behavioral simulation to resolve communication and interaction problems among the chips. The need for detailed simulation is especially true with respect to exceptional conditions such as interrupts and traps.

Technology

CMOS with higher speed and lower power consumption is replacing NMOS. As devices scale to smaller geometries and as the chip area increases, many electrical problems, such as inductance, can no longer be ignored. Furthermore as more functions can be placed on-chip, on-chip interactions become more complex while off-chip communication remains a major bottleneck.

We believe the problems we faced in designing the SPUR CPU were a preview of what the rest of the research community will need to confront in the near future. There are two terms that I will use often in this thesis. Before going any further, let me clarify my definitions to avoid confusion latter.

Macroarchitecture

The term macroarchitecture can be defined as the machine language programmer's view of the processor, generally found in the machine language programmer's manual. For a microprocessor, however, a machine language programmer's manual really does not tell the whole story. The macroarchitecture of a microprocessor should also include a interface specification for the board designer.

Microarchitecture

The term microarchitecture will be defined formally in Chapter 5. In the meantime, it is defined informally as the specification of how the macroarchitecture is implemented in a given technology. The microarchitecture may have some impact on the macroarchitecture. This feedback path is one of the main tenets of the original RISC argument.

1.3. Contemporary RISC Processors

This section looks at several RISC processors that were introduced at approximately the same time when the SPUR CPU was being built. I have selected two research projects and three commercial projects. In my opinion, each of the selected processors has its own significant feature or features that make the processor deserve a place in the short but brilliant history of

RISC processors. The two research processors I selected are MIPS-X and CRISP:

MIPS-X

MIPS-X [ChH87] [Hor87b] [Hor87a] was designed at Stanford University. It was implemented in 2μm, double-layer metal, CMOS technology. It contains 150K transistors in an 8mm x 8.5mm die and has 84 signal pins and 24 power pins. The peak operating frequency is 20MHz and the chip dissipates less than 1W. MIPS-X has a 32-word register file, a 512-word direct-mapped (32 blocks of 16 words) on-chip instruction cache, and uses a five-stage pipeline. The five stages are: (1) Instruction Fetch, (2) Register Read, (3) Execute, (4) Memory Access, and (5) Write Back of registers. The execution unit contains a 64-bit to 32-bit funnel shifter, a 32-bit ALU, and a special register MD that is used by the multiplication-step and division-step instructions. Branches are delayed for two cycles. In order to help the compiler to fill these two delay slots, MIPS-X has the option to change these delay instructions into NOOP on the fly ("squash" the instructions). The MIPS-X coprocessor interface treats coprocessor instructions as a form of memory operation and uses the address lines to transmit the coprocessor instructions to the coprocessor(s). The most significant feature of MIPS-X is its fast cycle time. Unlike the SPUR designers who took the conservative approach to increase the chance of getting a reliable CPU, the MIPS-X circuit designer used very aggressive circuit designs to lower the cycle time.

CRISP

CRISP [BDM87] [DiM87] [Ber87] was designed at AT&T Bell Laboratories. It was implemented in 1.75μm, single-layer metal, double-layer polysilicon, CMOS technology. It contains 170K transistors in an 10.35mm x 12.23mm die and has 95 signal pins, 20 power pins, 19 ground pins, and 6 test pins. The peak operating frequency is 16MHz and the chip dissipates 500mW. CRISP can be divided into six functional blocks: (1) Input/Output, (2) Prefetch Buffer, (3) Prefetch and Decode Unit, (4) Decoded Instruction Cache, (5) Execution Unit, and (6) Stack Cache. The prefetch buffer is a 512-byte direct-mapped cache organized into 32 blocks. The decoded instruction cache is a direct-mapped cache with 32 192-bit entries. Each entry is fully decoded instruction. The Execution Unit uses a three-stage pipeline: (1) Operand fetch, (2) ALU operation, and (3) register writeback. The stack cache is implemented with two 32-word byte-addressable register files. Branches can be "folded" into other instruction and is executed implicitly as part of other instructions. The most significant features of CRISP are its architectural innovations: stack cache and branch folding. According to the CRISP designers, stack cache access time is as fast as register but has the advantage of software transparency. Branch folding enable CRISP to execute branch in parallel with other useful instructions. This makes CRISP the first microprocessor that can execute multiple instructions per cycle.

The three commercial processors I selected are R3000, SPARC SF9010IU, and MC88000. Information concerning the detailed microarchitecture of the commercial processors is not as readily available as it is for the research processors. However, there are still enough information for me to judge why these commercial processors deserve special attention.

R3000

R3000 [Kan88] was designed at MIPS Computer. It is implemented in 1.2μm CMOS technology and resides in a 172 pins PGA. The peak operating frequency is 25MHz. R3000 has a 32-word register file. There is no on-chip instruction nor data cache. Integer multiplication is supported by hardware but integer division is supported by software only. Branches are delayed for one cycle. The most significant feature of R3000 is its speed–the 25MHz clock rate probably makes the R3000 the fastest RISC CPU when it was introduced.

SPARC SF9010IU

SPARC SF9010IU [NaA88] is the first implementation of Sun Microsystem's Scalable Processor Architecture (SPARC) [Gar88]. It was implemented in Fujitsu's high speed 20K gate, 1.5μm, 256-pin (156 of them are signal pins) gate-array. The peak operating frequency is 16.67MHz. SF9010IU has a 120-word register file organized into eight global registers and seven overlapped windows, an dual-instruction buffer, and uses a four-stage pipeline. The four stages are: (1) Instruction Fetch, (2) Decode, (3) Execute, and (4) Write. Integer multiplication step is supported by hardware but integer division is supported by software only. Branches are delayed for one cycle and the delay instruction can be "squashed" depending on the branch outcome. The most significant feature of SPARC SF9010IU is its simplicity. It is so simple that it was implemented in single gate array in a relatively short period of time.

MC88000

MC88000 [DRN88] was designed at Motorola. It was implemented in 1.5μm CMOS technology and resides in a 181 pins PGA. The peak operating frequency is 20MHz. MC88000 has a 32-word register file but there is no on-chip instruction nor data cache. Integer multiplication, integer division, as well as floating point arithmetics are all supported by hardware. The most significant feature of MC88000 is that it follows a supercomputer model that employs a scoreboard similar to the CDC 7600. The centerpiece of the architecture is a set of multiple pipelined functional units that can execute independently and concurrently. The usage of these functional units are controlled via scoreboarding. MC88000 is one of the few RISC processors that uses on-chip resources for floating point hardware instead of for instruction cache.

1.4. Research Goal and Thesis Organization

The goal of this thesis is not to formulate the theory of VLSI design but to explain the science of VLSI design to someone who wants to build a microprocessor. This can be accomplished by providing a quantitative way to evaluate and a systematic way to design microarchitecture. Since this research is based on the SPRU CPU design experience, the SPUR CPU microarchitecture and the lessons I learned must be introduced first. This thesis is organized as follows:

Chapter 2 Describes the SPUR CPU microarchitecture.

Chapter 3 Discusses the lessons I learned in designing the SPUR CPU.

Chapter 4 Develops a quantitative way to evaluate a microprocessor's microarchitecture. Different features of the SPUR CPU microarchitecture are then evaluated as examples.

Chapter 5 Develops a systematic approach to design a microprocessor's microarchitecture. I illustrate this approach by using it to recreate the SPUR CPU microarchitecture.

Chapter 6 Summarizes the thesis. I also say a few words about what I think the future will be like based on my experience in SPUR,

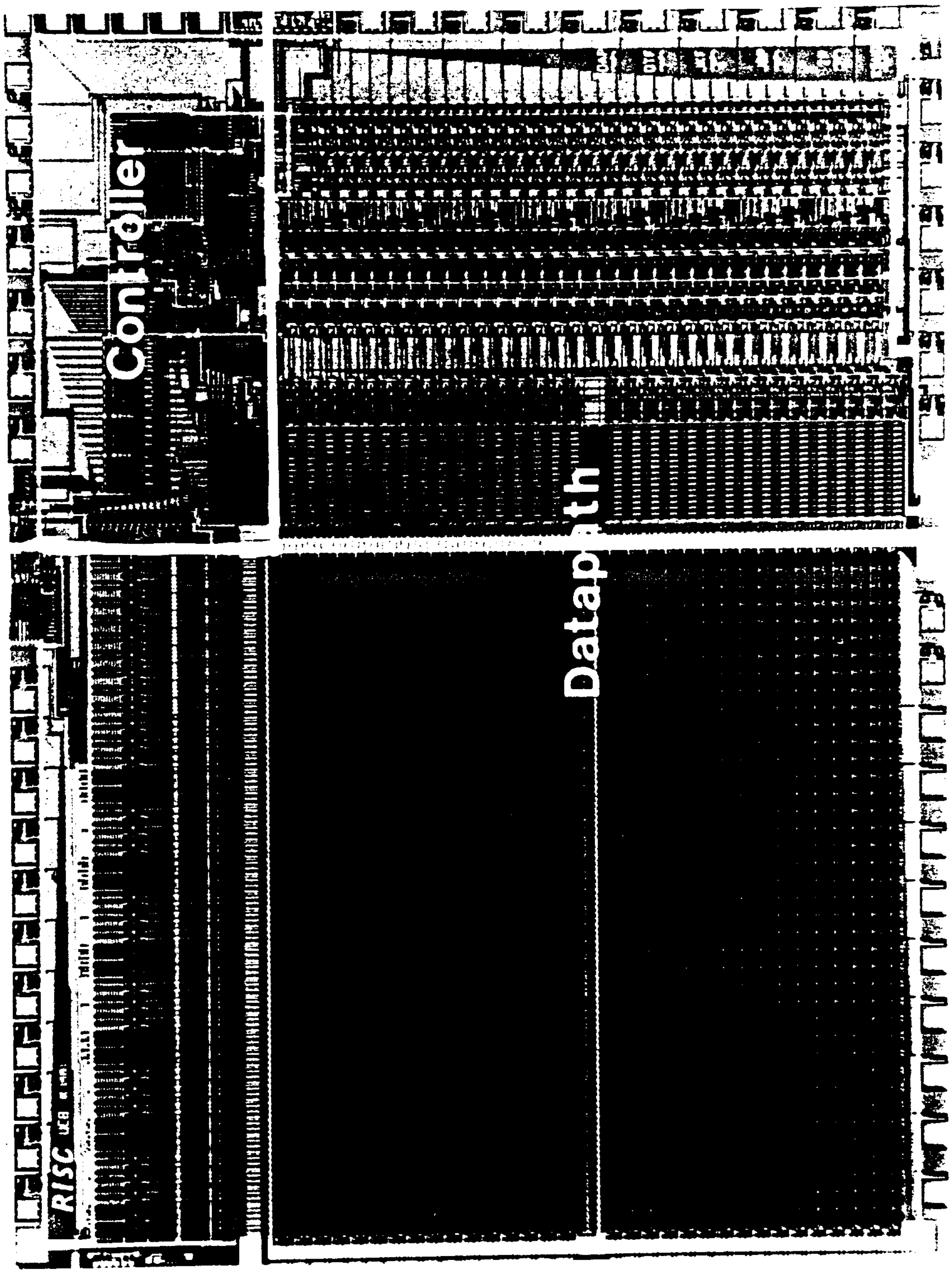


Figure 1-4-1 The First Generation Berkeley RISC Machine: RISC I

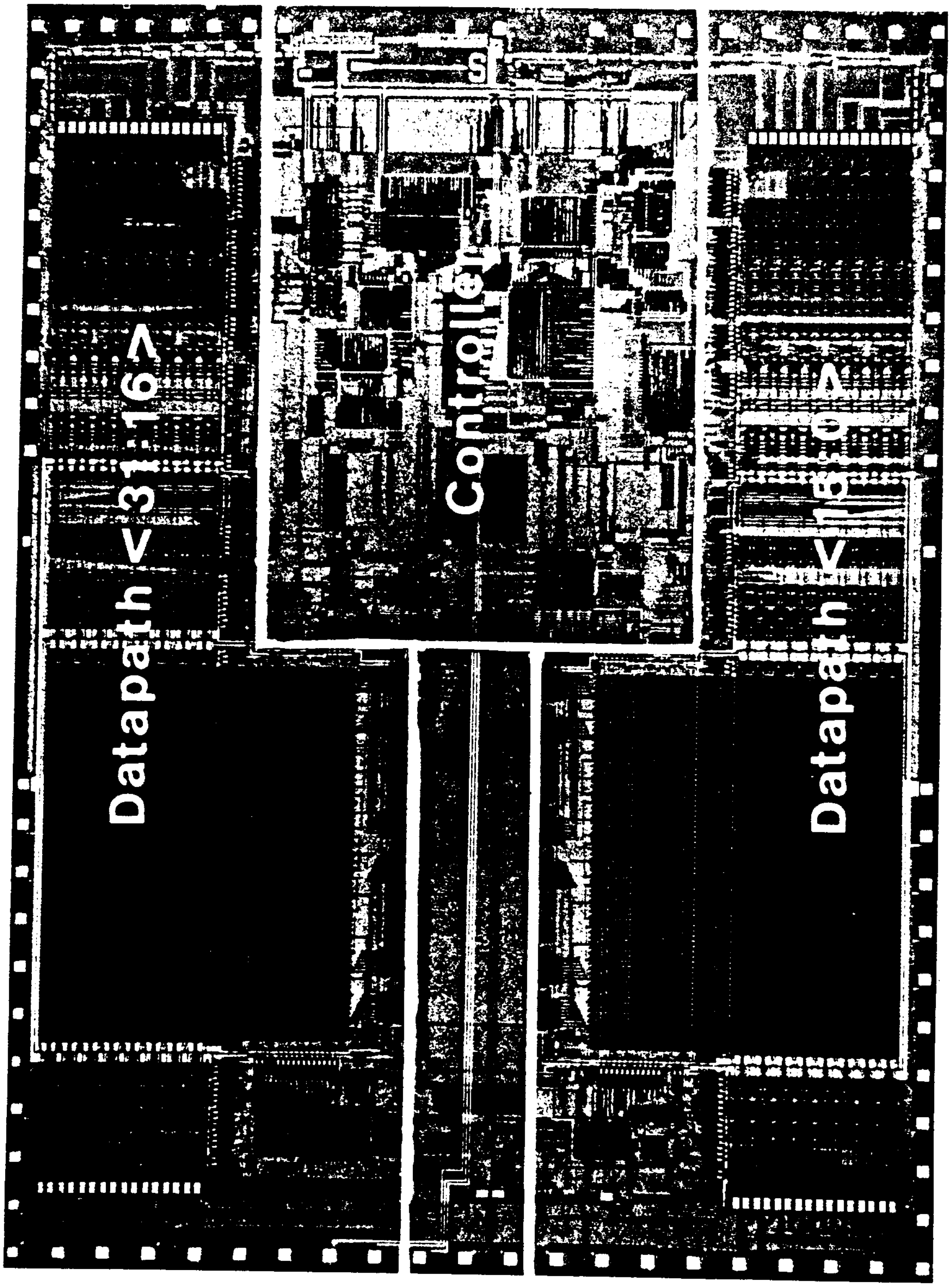


Figure 1-4-2 The Second Generation Berkeley RISC Machine: SOAR

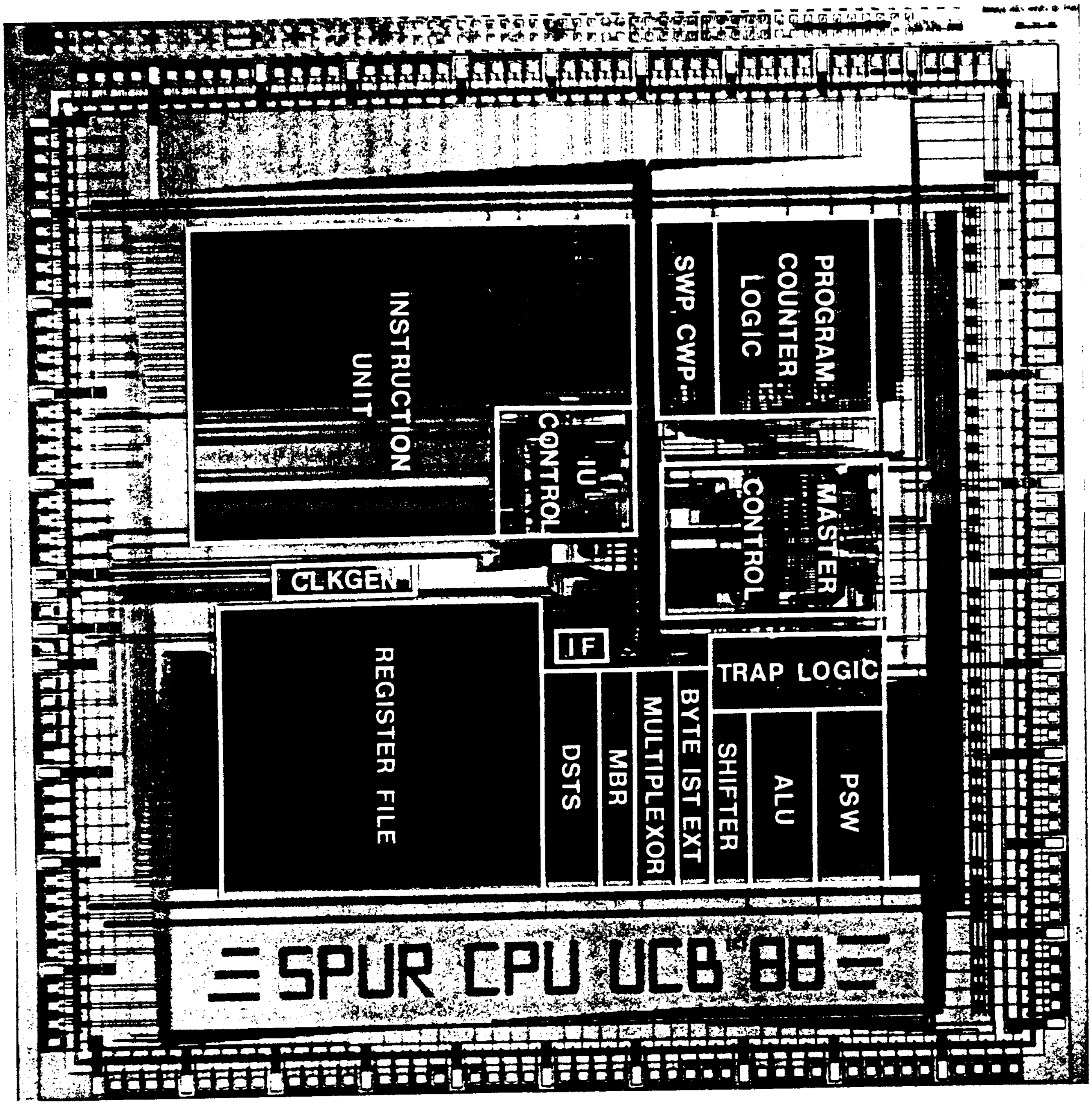


Figure 1-4-3 The Third Generation Berkeley RISC Machine: SPUR CPU

1.5. REFERENCES

[BDM87] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan, "Architectural Innovations in the CRISP Microprocessor ", *COMPCON 87*, San Francisco, California, February 23-27, 1987.

[Ber87] A. Berenbaum et al., "CRISP: A Pipelined 32-bit Microprocessor with 13-kbit of Cache Memory", *IEEE Journal of Solid-State Circuits SC-22*, 5 (October, 1987).

[BiW86] J. S. Birnbaum and W. S. Worley, Jr., "Beyond RISC: High-Precision Architecture", *COMPCON 86*, San Francisco, March 3-6, 1986.

[BPT87] B. K. Bose, L. Pei, G. S. Taylor and D. A. Patterson, "Fast Multiply and Divide for a VLSI Floating-Point Unit", *Proc. Eighth IEEE Int'l. Symposium on Computer Arithmetic*, May 1987, 87-94.

[ChH87] P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X", *The 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2-5, 1987.

[DiM87] D. R. Ditzel and H. R. McLellan, "The Hardware Architecture of the CRISP Microprocessor", *The 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2-5, 1987.

[DRN88] C. Dobbs, P. Reed and T. Ng, "Supercomputing on Chip", *VLSI Systems Design IX*, 5 (May 1988).

[Gar88] R. Garner, "SPARC: Scalable Processor Architecture", *Sun Technology*, Summer, 1988.

[Gib87] G. Gibson, "Estimating Performance of Single Bus, Shared Memory Multiprocessors", Report No. UCB/Computer Science Dpt. 87/355, Computer Science Division, EECS Department, University of California, Berkeley, May 1987.

[Hil86] M. Hill et al., "Design Decisions in SPUR", *Computer 19*, 11 (November 1986).

[Hor87a] A. Horowitz et al., "MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache", *IEEE Journal of Solid-State Circuits SC-22*, 5 (October, 1987).

[Hor87b] M. Horowitz et al., "A 32b Microprocessor with On-Chip 2Kbyte Instruction Cache", *ISSCC 87*, New York, February 25-27, 1987.

[Kan88] G. Kane, *MIPS RISC Architecture*, Prentice Hall, 1988.

[Kat83] G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, Doctoral Dissertation, Computer Science Division, EECS Department, University of California, Berkeley, October 1983.

[KEW85] R. Katz, S. Eggers, D. Wood, C. Perkins and R. Sheldon, "Implementing A Cache Consistency Protocol", *The 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, June 17-19, 1985.

[MoS85] L. Monier and P. Sindhu, "The Architecture of the Dragon", *COMPCON 85*, San Francisco, February 25-28, 1985.

[NaA88] M. Namjoo and A. Agrawal, "Implementing SPARC: A High-Performance 32-bit RISC Microprocessor", *Sun Technology*, Winter, 1988.

[Pat82] D. A. Patterson, "A RISCy Approach to Computer Design", *COMPCON 1982*, February, 1982.

[TSJ88] C. P. Thacker, L. C. Stewart and E. H. S. Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers 37*, 8 (August 1988).

[Ung84] D. Ungar et al., "Architecture of SOAR: Smalltalk on a RISC", *The 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, June 5-7, 1984.

[Woo86] D. A. Wood et al., "An In-Cache Address Translation Mechanism", *The 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 2-5, 1986.

Chapter 2

THE SPUR CPU MICROARCHITECTURE

If things are too complex, I can't understand them.

Seymour Cray, 1976

This chapter describes the microarchitecture of the SPUR CPU. Section 2.1 gives an overview and covers all the important features. The SPUR CPU can be divided into two units: the Instruction Unit and the Execution Unit. Section 2.2 describes the Instruction Unit and Section 2.3 describes the Execution Unit. Finally, Section 2.4 describes the controller that controls the SPUR CPU. The following naming conventions are used in describing the microarchitecture:

- Register names start with an upper case letter and the rest are lower case except to improve readability. Examples are Dst1 and IfetPC.
- Functional block names are in upper case letters only. Examples are ALU and EXT_INS.
- Signal names start with a lower case letter and the rest are lower case except to improve readability. Examples are busA and trapType.

The goal of this chapter is to give an overall picture of the SPUR CPU microarchitecture, so I can use the SPUR CPU as an example in latter chapters. Please refer to Appendix A for a detailed discussion of the microarchitecture.

2.1. The SPUR CPU Microprocessor

The SPRU CPU is a third generation Berkeley RISC microprocessor, and this section describes the important features of the SPUR CPU microarchitecture.

2.1.1. Overview of the SPUR CPU

The SPUR CPU is similar to RISC II that it has a reduced instruction set and a 138-register register file organized into 10 global registers and eight overlapped register windows (see Appendix A). However, unlike RISC II, the SPUR CPU also has a 512-byte on-chip instruction cache, a four-stage pipeline, a cache controller interface, and a parallel coprocessor interface. Internally,

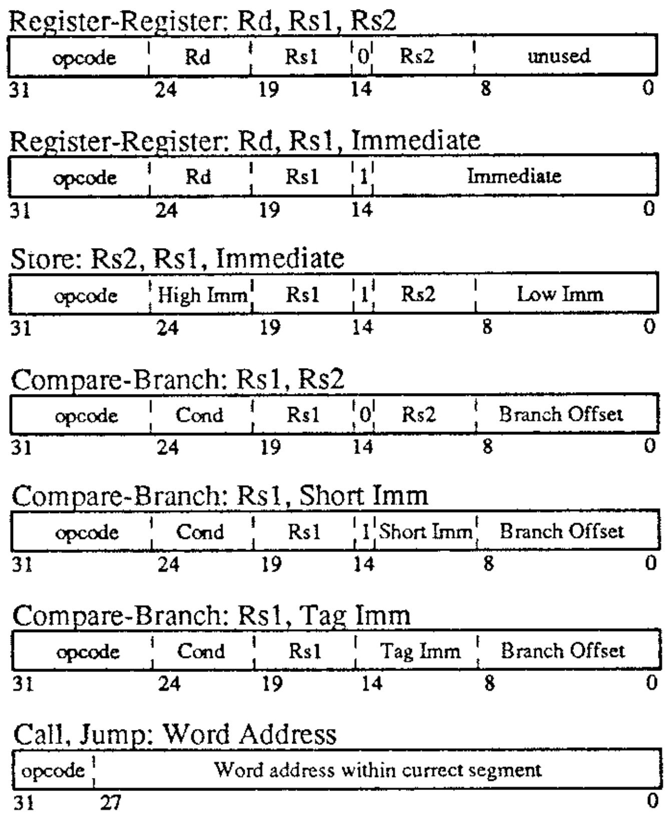


Figure 2-1-1 SPUR Instruction Formats

Register-Register instructions use the Rs1-Rs2 or Rs1-Immediate pair to specify the source operands and the result is stored into the register specified by Rd. For Store instructions, Rs2 contains the value to be stored and the effective address is formed by adding Rs1 to the concatenation of the High Imm and Low Imm fields. Compare-Branch instructions' formats are selected by the Cond field. The three formats are: (1) Rs1-Rs2 format–compare the two registers' contents, the two registers' type-tags, or both contents and tags; (2) Short Imm format–compare the zero extension of the Short Imm field with Rs1's contents; (3) Tag Imm format–compare the 6-bit Tag Imm with Rs1's type-tag.

the SPUR CPU uses a combination of a byte extractor/insertor and a shifter instead of the more complicated barrel shifter. It also uses an extra adder to calculate the branch address to support the 1-cycle Compare-Branch instructions. Finally, eight extra tag bits are attached to each 32-bit register to support LISP. This makes SPUR CPU register 40 bits wide (see Figure 2-1-2). The SPUR CPU was fabricated in 1.6μm, double layer metal, CMOS technology. The die size is 1.15cm x 1.15cm and is packaged in a 208 pin pin grid array.

The SPUR CPU is a register-to-register machine in which load and store are the only type of instructions that access memory. The effective address of load and store instructions can either be the sum of two registers or the sum of one register and an immediate constant. The SPUR memory system does *not* support byte addressing and the two least significant bits of the 32-bit address are always ignored by the memory system.

The SPUR CPU modes of operation can be divided into two orthogonal sets: (1) User vs. Kernel and (2) Virtual vs. Physical. Only when the SPUR CPU is in kernel mode can privileged instructions be executed. In virtual mode, data and instruction addresses generated by the SPUR CPU are interpreted by the SPUR memory system as virtual addresses. In physical mode, addresses are interpretated as physical address, which is useful in debugging and bootstrapping the system. The mode of operation is controlled by writing different bit patterns into the Kernel Processor Status Word–Kpsw (see Appendix A).

2.1.2. Instruction Formats

SPUR CPU instruction set [Tay85] (see Appendix A-3) can be grouped into four genetic instruction types: Register-Register, Store, Compare-Branch, and Call-Jump. The formats of these genetic types are shown in Figure 2-1-1.

Load and Return type instructions are special cases of Register-Register in which (Rs1 + Rs2) or (Rs1 + Immediate) are used as the the effective address. The Rd field specifies the register to be loaded for the Load type instruction and is not used for Return type instruction. In order to modify any special register, its contents must first be read into a general purpose register and

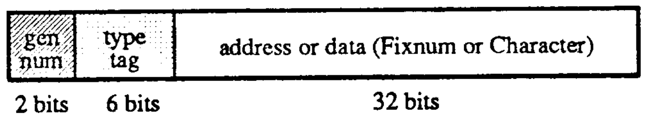


Figure 2-1-2 SPUR Pointer

A SPUR pointer is a 40-bit word composed of a 32-bit address, a 6-bit type tag, and a 2-bit generation number. These three parts are logically independent. The 6-bit type tag allows up to 64 possible types. The SPUR CPU hardware only recognizes two data types: *Fixnum*, which is an integer that fits in a 32-bit word, and *Character*. In SPUR, *Fixnum* and *Character* are not referenced indirectly through pointers but are represented as immediate data in the 32-bit "address"field. Besides these two immediate data types, the SPUR CPU hardware also recognizes two pointer types: *Cons* and *Nil.*

then written back to the special register after the modification.

2.1.3. LISP Support

The SPUR CPU supports LISP by three types of tag checking [Tay86] [ZHH88]: data type checking for general operations, pointer type checking for list operations, and generation checking for garbage collection.

Data Type Checking. In a runtime typing system such as LISP, the type of a variable is not known at compile time and can change during the course of execution. Therefore, every object's type must be stored within the object itself or in all the pointers that point to it. SPUR stores the type in the pointer because the type information is then available before the memory reference. Figure 2-1-2 shows the SPUR pointer which can be stored in a 40-bit CPU general purpose register. The SPUR CPU instructions can be divided into two groups with respect to data-type checking:

(1) Data-type checking is not performed and the tag field is ignored. LOAD is an example.

(2) Data-type checking is done in parallel with the data operation and traps conditionally. For example, ADD will trap if either operand is not a *Fixnum.*

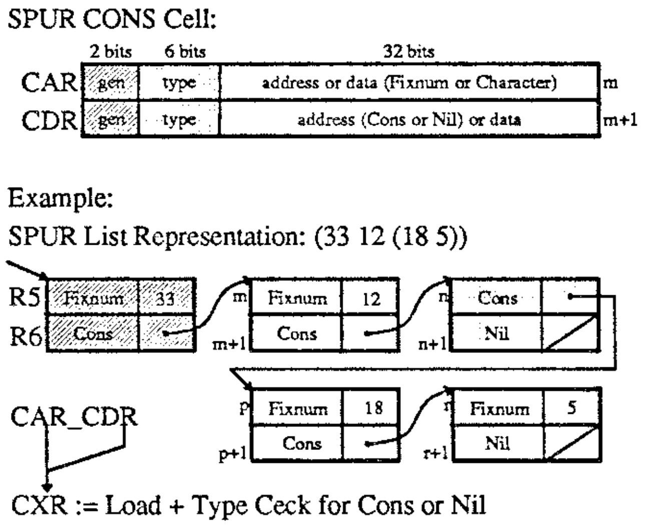


Figure 2-1-3 Pointer-Type Checking

Fixnums 33, 12, 18, and 5 are stored as immediate data. Assume the shaded CONS cell is already in the CPU registers R5 and R6 and all other CONS cells are still in memory. More specificly, assume the dotted CONS cell is in memory location n and n + 1, then the operation:

CAR R6 <=> CXR (R6) => Load at location m (data *Fixnum* 12);
CDR R6 <=> CXR (R6+1) => Load at location m+1 (pointer *Cons* n);
CAR R5 <=> CXR (R5) => Trap because R5 is a *Fixnum.*

In order to simplify the figure, this example does not show the generation portion of the tag.

Pointer Type Checking. Figure 2-1-3 shows how SPUR represents a LISP list element by a pair of consecutive storage elements called a CONS cell–one represents the CAR pointer and the other represents the CDR pointer. Since the SPUR CPU is a load-store machine, CAR and CDR operations are similar to the load operation. However, CAR and CDR operations are defined in Common LISP to work only for a *Cons* pointer (points to another CONS cell) or *Nil* pointer (points to nothing). The SPUR CPU supports this feature by a special load instruction: the CXR instruction performs the load in parallel with the pointer-type checking for *Cons* or *Nil.*

Generation Checking. SPUR uses the generation scavenging garbage collection algorithm [Ung84] [LiH83]. This is based on the observation that the longer an object has been in use, the more likely it is to continue to be in use. In SPUR, objects are separated dynamically into three generations. Each object's generation is recorded in its two generation bits. For simplicity, Figure

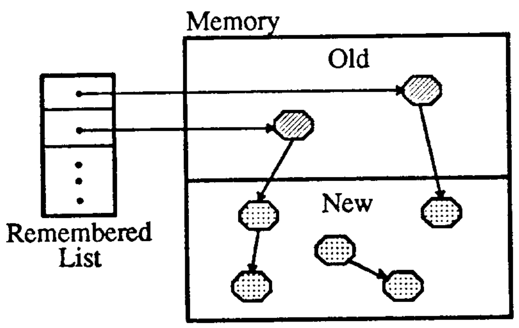


Figure 2-1-4 Generation Checking

In this simplified drawing, objects are separated into two generations. New objects are allocated in the younger generation and move to the older generation if they survive a garbage collection. The garbage collector restricts its attention to the younger generations as much as possible. In order to identify new objects that are currently in use, a "Remembered List" is used to keep track of all the older objects that contain pointers to the younger generation.

2-1-4 only shows two generations. The SPUR CPU has a special store instruction which compares the generation number of the operands in parallel with the store and traps to a routine that updates the "Remembered List" whenever necessary.

2.1.4. The Basic Ingredients–Blocks, Clocking, and Pipeline

Block Diagram. Figure 2-1-5 can be considered as an abstract floor plan which shows the relative position of each block within the SPUR CPU. The dimension of each block, however, is not drawn in scale.

Clocking. The SPUR CPU uses a four-phase non-overlap clock, that is each cycle consists of four phases. Each phase has an nominal duration of 18 ns and there are 7 ns nominal non-overlap time between each phase. This makes the SPUR CPU cycle time 100 ns.

Pipeline. The SPUR CPU uses a uniform four-stage pipeline (Figure 2-1-6). Each pipe stage corresponds to one clock cycle and all instructions take four cycles to finish. The four stages are:

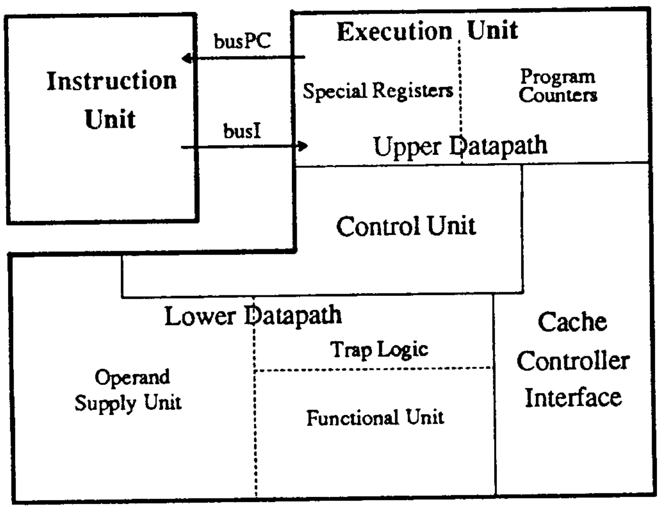


Figure 2-1-5 The SPUR CPU Abstract Block Diagram

The SPUR CPU can be divided into two units: *Instruction Unit* and the *Execution Unit*. The *Execution Unit* can be further divided into four parts: the *Cache Controller Interface*, which is out of the scope of this chapter, the *Lower Datapath* that handles all the data manipulating operations, the *Upper Datapath* that handles all the the program control operations, and the *Control Unit* that controls the *Upper Datapath* and *Lower Datapath*.

Ifet	Instruction fetch. The instruction is delivered to the *Execution Unit*.
Exec	Register read and instruction execution for register–to–register instructions or register read and effective–address calculation for memory access instructions.
Mem	Memory access for all memory access instructions.
Wr	Register write. Write results back to the register file.

Unlike the RISC II 3-stage pipeline, the extra Mem stage in the SPUR CPU pipeline eliminates the need to stall the pipeline whenever a load instruction is executed. This was considered to be an important at the beginning of the project because the frequency of Loads was expected to be higher in LISP than C. Similar to RISC II, a branch conflict in the SPUR CPU pipeline is resolved by a delay branch with one instruction in the delay slot. Delay branch with the option to cancel the instruction in the delay slot was considered but not implemented due to the

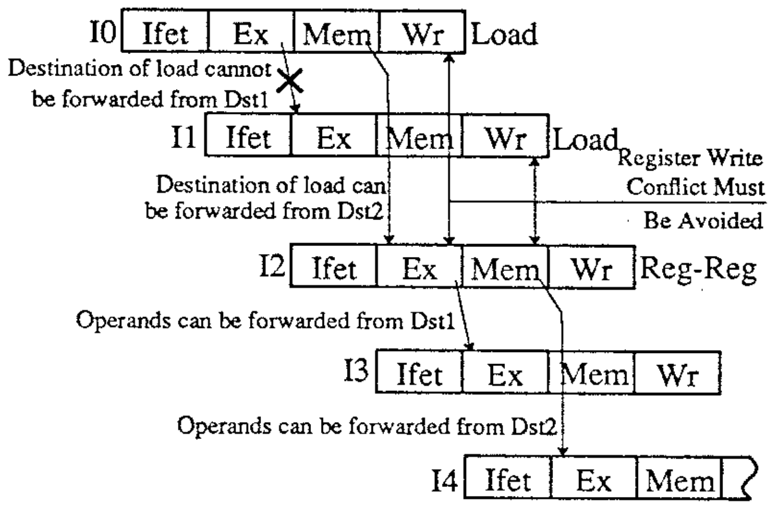

I0
Ifet
Ex
Mem
Wr
Load
Destination of load cannot be forwarded from Dst1
I1
Ifet
Ex
Mem
Wr
Load
Register Write Conflict Must Be Avoided
Destination of load can be forwarded from Dst2
I2
Ifet
Ex
Mem
Wr
Reg-Reg
Operands can be forwarded from Dst1
I3
Ifet
Ex
Mem
Wr
Operands can be forwarded from Dst2
I4
Ifet
Ex
Mem

Figure 2-1-6 The SPUR CPU Pipeline

Register-to-Register instruction (*I2*) always finishes its execution at the end of its *Exec* stage. However, to avoid register write conflicts with possible previous Load instructions (*I0, I1*), *I2's* result is not written into the register file until its *Wr* stage. Because of this delay, two temporary registers, Dst1 and Dst2, are needed to store the result at the end of its Exec and Mem stage respectively. Instructions *I3* and *I4*, will not be able to read *I2's* result from the register file but the result can be read directly from *Dst1* and *Dst2*. This is referred to as internal forwarding.

complexities involving interactions of internal forwarding, pipeline suspension, and the coprocessor interface.

As illustrated in Figure 2-1-6, data conflicts in the SPUR CPU pipeline are resolved by internal forwarding in which operands are supplied by temporary registers Dst1 or Dst2. For the Load instruction (*I0* in Figure 2-1-6) the value to be loaded comes from the external data bus at the end of the *Mem* stage and goes directly into temporary register Dst2. Internal forwarding via Dst1 is therefore impossible for the instruction immediately following the Load. In order to simplify the internal forwarding logic, the destination register of the Load instruction is defined to have an unknown value for the instruction immediately following the Load.

2.2. Instruction Unit

The Instruction Unit provides the SPUR CPU a 512-byte direct-mapped on-chip instruction cache. The Instruction Unit's organization, control, and operation are discussed in Section 2.2.1, Section 2.2.2, and Section 2.2.3 respectively. The implementation of the Instruction Unit is described in Rich Duncombe's Master of Science report [Dun86].

2.2.1. Instruction Unit Organization

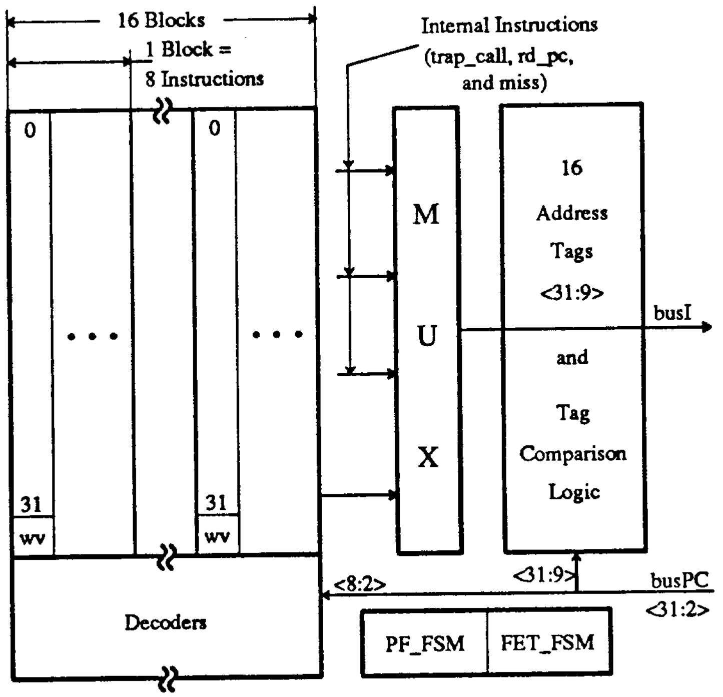

16 Blocks
1 Block = 8 Instructions
Internal Instructions (trap_call, rd_pc, and miss)
0
0
31
wv
31
wv
M U X
16 Address Tags <31:9> and Tag Comparison Logic
busI
<8:2>
<31:9>
busPC <31:2>
Decoders
PF_FSM
FET_FSM

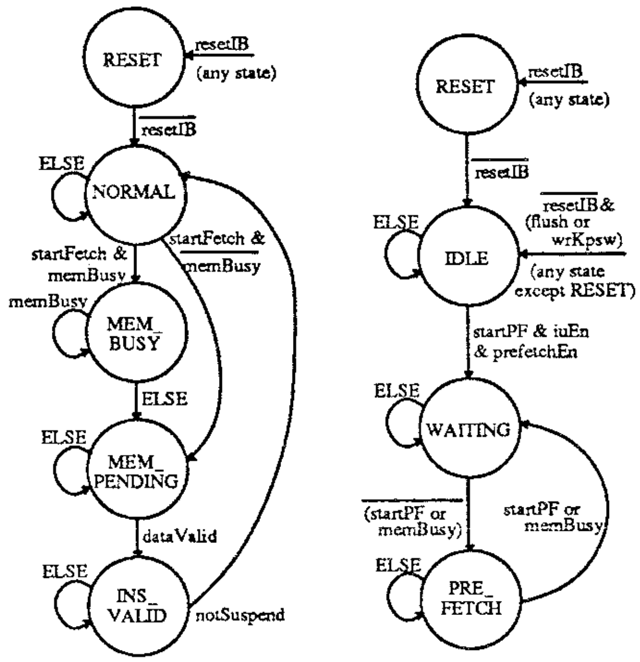
Figure 2-2-1 Instruction Unit Block Diagram

The 512-byte direct-mapped instruction cache is organized into sixteen blocks with eight instructions per block. The Execution Unit requests an instruction by placing an address onto busPC and the Instruction Unit delivers the instruction via busI. Since all SPUR instructions are 4 bytes long, only the upper 30 bits of busPC are used to access this cache. The Instruction Unit is controlled by two finite state machines: the Fetch Finite State Machine (FET_FSM) and the Prefetch Finite State Machine (PF_FSM). Besides acting as an internal instruction cache, the Instruction Unit also provides internal instructions to simply the control of the Execution Unit.

The Instruction Unit, as shown in Figure 2-2-1, contains a 512-byte direct-mapped on-chip instruction cache [Hil87]. This cache is different from a regular cache that during a cache miss, only the missing instruction rather than the full eight-instruction cache block is brought immediately into the cache from the next higher level of memory. After the Execution Unit has received this missing instruction and has resumed its normal operation, the Instruction Unit will try to prefetch the rest of the block into the cache one instruction per cycle starting at the instruction immediately after the missing instruction. In other words, each eight-instruction block in Figure 2-2-1 is divided into eight one-instruction sub-blocks. On a cache miss, the requested sub-block is brought into the cache immediately and a prefetching process is triggered to bring the rest of the sub-blocks into the cache. Under ideal conditions, prefetching is fast enough that after the first miss in a block, there will not be any more cache misses within that block as long as the Execution Unit is executing sequential code.

Prefetching has the lowest priority among all cache access because the instruction being prefetched may not be needed by the Execution Unit at all. Therefore, prefetching is interrupted whenever the Execution Unit performs a data access (load or store) or whenever there is a new miss in the Instruction Unit. Furthermore, if the instruction being prefetched is not in the external cache, it will *not* cause an external cache miss. The prefetcher simply move onto the next instruction. Since prefetching is not always successful, each instruction must have a "Word Valid Bit" (WV) to indicate its validity.

The Instruction Unit also plays a role in simplifying the control of the Execution Unit by providing internal instructions to "fool" the Execution Unit pipeline. For example, the internal instructions *trap_call* and *rd_pc* are used to simplify trap handing. This will be explained in Section 2.3.3. The *miss* internal instruction is used whenever the Execution Unit requests an instruction that is not currently in the Instruction Unit. This case is discussed in Section 2.2.3.



Fetch Finite State Machine Prefetch Finite State Machine

Figure 2-2-2 Simplified State Diagrams

The diagram on the left is the state diagram for the Fetch Finite State Machine and the one on the right is for the Prefetch Finite State Machine. In the Fetch Finite State Machine, the input signal *startFetch* is a composite signal:

startFetch = notSuspend & notWrKpsw & (ibMiss or flush)

Outputs of these machines are not shown here for simplicity. All but one outputs are used to control the datapath of the Instruction Unit. The exception is *startPF*. It is an output of the Fetch Finite State Machine and it triggers the Prefetch Finite State Machine:

startPF = MEM_BUSY or (NORMAL and notSuspend and (ibMiss or flush))

2.2.2. Instruction Unit Control

Initially, we envisioned a single finite state machine controlling the entire Instruction Unit. This turned out to be a difficult design task and the result was so hard to understand that we had little confidence in its correctness. Further investigation revealed that prefetching should occur in parallel with other Instruction Unit operations and is quite autonomous. This gave us the idea of delegating the control to two independent finite state machines: one for prefetching, the Prefetch Finite State Machine, and one for the rest of the operations. For the lack of a better name, it is

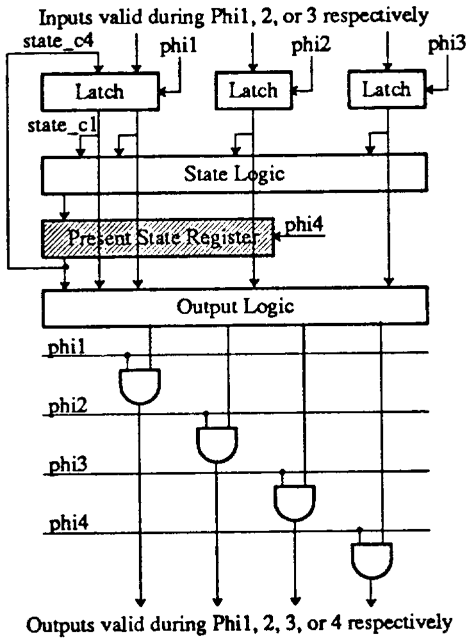


Figure 2-2-3 Generic Structure of the I-Unit Finite State Machines

The state of the finite state machine is determined by the contents of the Present State Register (shaded in this figure). Since the output of the Present State Register is updated every $\phi 4$ and is stable by $\phi 1$, a state begins in $\phi 1$. The state informations *state_c1* must be used to generate outputs that are valid during $\phi 4$ to prevent race condition. Similarly, *state_c4* must be used to generate outputs that are valid during $\phi 1$. For outputs that are valid during $\phi 2$ and $\phi 3$, either *state_c1* or *state_c4* can be used. Output of this finite state machine is a function of the Present State and any inputs that are valid during ϕN (N=1, 2, or 3) can affect outputs that are valid during $\phi N{+}1$ of the same cycle. For example, outputs that are valid during $\phi 4$ can be a function of inputs that are valid during $\phi 1$, $\phi 2$, and $\phi 3$.

called the Fetch Finite State Machine. The simplified state diagrams of these two finite state machines are shown in Figure 2-2-2.

Both finite state machines only have a small number of states and are implemented by the generic structure shown in Figure 2-2-3. The State Logic and Output Logic blocks are implemented by PLAs. There are two reasons why they not combined into one single block (Figure 2-2-4) as suggested by most "classical" VLSI text books. First, separating them makes the designer's job easier. More importantly, in this arrangement the outputs depend on the Present State and any inputs that are valid during phase N (N=1, 2, or 3) can affect outputs that are valid

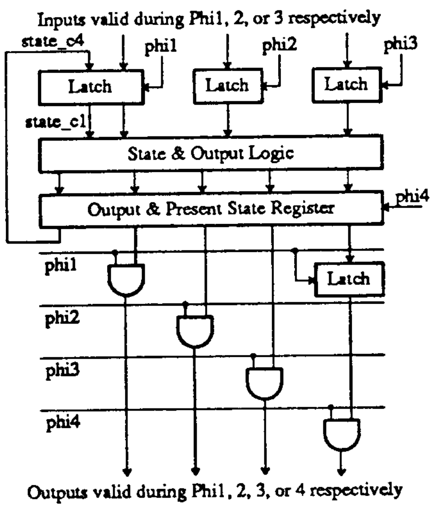

Inputs valid during Phi1, 2, or 3 respectively
state_c4
phi1
phi2
phi3
Latch
Latch
Latch
state_c1
State & Output Logic
Output & Present State Register
phi4
phi1
Latch
phi2
phi3
phi4
Outputs valid during Phi1, 2, 3, or 4 respectively

Figure 2-2-4 Classical Implementation of Finite State Machine

In this classical implementation, the State & Output Logic evaluates the next state and output during $\phi 4$. Since the output and next state are evaluated together and the results are latched into the Output & Present State Register at the end of $\phi 4$, the outputs can only be affect by the previous state and inputs from the previous cycle. In contrast, the outputs of the organization in Figure 2-2-3 are functions of the present state and inputs from any previous phase of the current cycle. Notice that a latch must be placed between the Output register and the output signals that are valid during $\phi 4$ to prevent race condition.

during phase N+1 of the same cycle. On the other hand, if we combine the Output Logic and State Logic as tradition dictates (Figure 2-2-4), the output signals can only depend on the previous state and the inputs from the previous cycle rather than the current state and the inputs from the previous clock phase. This will increase the latency of the Instruction Unit.

Since we did not implement the single finite state machine version of the control, it is hard to judge the advantages of separating the control into two finite state machines in terms of implementation metrics such as area, power consumption, and number of logic gates. However, this separation greatly simplifies the design and verification effort by allowing us to focus our attention on one thing at a time. This illustrates an important point in VLSI design: logic optimization is important as long as you are still trying to meet implementation constraints. Once these constraints are met, continuing optimization can be counterproductive not only because the design

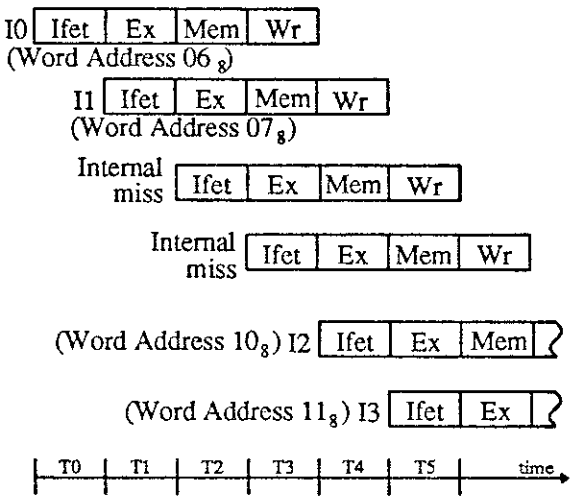

I0 Ifet Ex Mem Wr
(Word Address 06 8)
I1 Ifet Ex Mem Wr
(Word Address 07 8)
Internal miss Ifet Ex Mem Wr
Internal miss Ifet Ex Mem Wr
(Word Address 10 8) I2 Ifet Ex Mem
(Word Address 11 8) I3 Ifet Ex
T0 T1 T2 T3 T4 T5 time

Figure 2-2-5 Fetch and Prefetch–Execution Unit's Perspective

Assume instructions *I0*, *I1*, *I2*, and *I3* are in consecutive word addresses (octal) 06, 07, 10, and 11 and only *I0* is currently (Cycle T0) in the Instruction Unit cache array. Under ideal conditions (most of the time), the Instruction Unit will be able to prefetch *I1* in time. However, when the Execution Unit requests *I2*, the block boundary is crossed and a miss occurs. Instead of suspending everything in the Execution Unit's Pipeline, the Instruction Unit inserts internal instruction *miss* into the pipeline such that both *I0* and *I1* can proceed while *I2* is being fetched. For the Instruction Unit's Perspective, please refer to Table 2-2-1.

time could be spent on something else, but also because it may make the design harder to understand and thus harder to verify and modify.

2.2.3. Instruction Unit Operation

Figure 2-2-5 shows Execution Unit's view of how a miss in the Instruction Unit cache array is handled under ideal conditions. After the Execution Unit is fooled by the internal *miss* instructions during cycles T2 and T3, the Instruction Unit tries to fetch *I2* and prefetch instructions in the same block as I2 from the external cache during cycles T4 and T5. This is further illustrated in Table 2-2-1. Notice that in Figure 2-2-5 during Cycle T2 and T3, instruction *I0* and *I1* are allowed to proceed. This is necessary to prevent deadlock since instructions *I0* or *I1* could be a Load or Store type instruction which would start accessing the external cache at the end of their

Cycle Figure 2-2-5	Fetch Finite State Machine State & Actions	Prefetch Finite State Machine State & Actions
T0	**NORMAL:** Deliver I0 (06)	**PREFETCH:** Prefetch (00) Receive and write I1 (07) into cache
T1	**NORMAL:** Deliver I1 (07)	**PREFETCH:** Prefetch (01) Instruction (00) is not written (Note 1)
T2	**NORMAL:** Deliver "miss" opcode Fetch I2 (10) from external cache	**PREFETCH:** Prefetch is blocked by Fetch
T3	**MEMPEND:** (Note 2) Deliver "miss" opcode Receive and write I2 (10) (Note 3)	**WAITING:** As soon as I2 is received, prefetch I2 (11)
T4	**INSVALID:** Deliver I2 (10) (Note 4)	**PREFETCH:** Prefetch (12) Receive and writes I3 (11)
T5	**NORMAL:** Deliver I2 (10)	**PREFETCH:** Prefetch (13) Receive and Write (12)

Table 2-2-1 Fetch and Prefetch–Instruction Unit's Perspective

Notice that prefetching does *not* cross block boundary. This is illustrated by Cycle T0 in which the prefetcher has already "wrap" around and start prefetching instruction at word address 00 instead of instruction at word address 010 (octal).

Notes:

1. Assume instruction (00) is already in the cache - no need to write.
2. Assume the external cache is not busy.
   Otherwise, it will go to MEM_BUSY and wait.
3. Assume the external cache can deliver the instruction in one cycle.
   Otherwise, it will stay in MEMPEND until it receives *I2*.
4. I2 is not received until phi3 of Cycle T3. Therefore it cannot
   be delivered to the Execution Unit until Cycle T4.

Exec stage. If their already started cache access are not allowed to finish, the I-Unit cannot start the fetch for *I2* because the external cache in SPUR is *not* separated into data and instruction caches. A deadlock would have occured because the Execution Unit would wait for *I2* but the Instruction Unit could not fetch *I2* until the cache is free.

Figure 2-2-5 and Table 2-2-1 show the ideal case in which the cache miss is handled in two cycles and the prefetch of *I3* is successful. In practice, either *I0* or *I1* can block off the cache access path and the Fetch machine must first go into MEMBUSY state during T3 and wait for cache to be free before starting the fetch of *I2* and entering the MEMPEND state. Once in the MEMPEND state, the external cache may not be able to deliver the *I2* within one cycle and the Fetch machine must stay there until *I2* is received. In other words, the Instruction Unit may take more than two cycles to recover from the miss and more than two *miss* instructions must be inserted.

The prefetching can be turned off by setting a bit in the Kernel Processor Status Word. In that case, the Prefetch Finite State machine will remain in the IDLE state and will not prefetch any instructions. Furthermore, the whole Instruction Unit can be disabled and none of the instructions will be cached. If the Instruction Unit is disabled, the Prefetch Finite State machine will remain in the IDLE state and the Fetch Finite State machine will continuously cycle between NORMAL, MEM_PENDING (or MEM_BUSY), and INS_VALID states. In other words, Instruction Unit disabled is just a special case in which every access to the Instruction Unit results in a miss.

2.3. Execution Unit

The Execution Unit executes the instructions delivered by the Instruction Unit. The Execution Unit's datapath organization is discussed in Section 2.3.1. The Execution Unit's operation under normal and adverse conditions are discussed in Section 2.3.2 and Section 2.3.3, respectively. The implementation of the Execution Unit's datapath is described by Dave Lee [Lee86].

2.3.1. Execution Unit Datapath

The Execution Unit datapath can be divided into two parts: the Lower Datapath and the Upper Datapath. The Lower Datapath performs all the register-to-register operations and can be further divided into an Operand Supplier and a Functional Unit. Here is a brief description of each block in the Lower Datapath, from left to right as shown in Figure 2-3-1 (please refer to the naming conventions described in the beginning of Chapter 2, P.14):

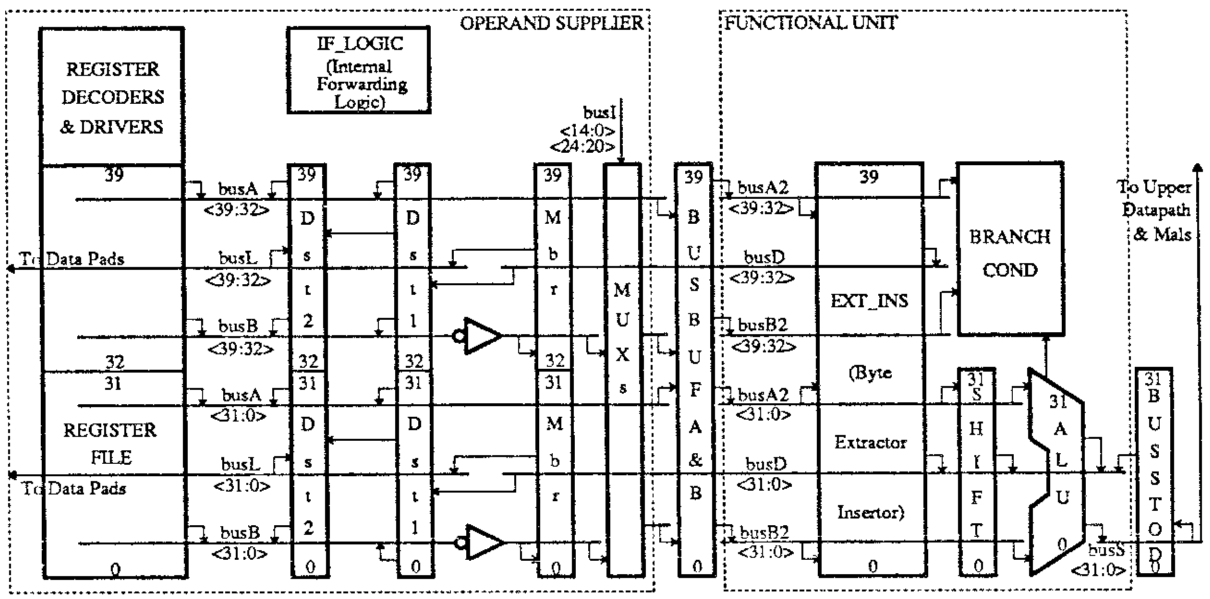


Figure 2-3-1 The SPUR CPU Lower Datapath

The SPUR CPU Lower Datapath is 40 bits wide–the upper eight bits (39-32) are for tags. BUSBUFA & B serves as operands buffers. Everything to its left can be considered as the Operand Supplier and everything to its right as the Functional Unit. BusA, busB, busA2, and busB2 route operands from the Operand Supplier to the Functional Unit. The result of the computation is routed back to the Operand Supplier via busD. BusL connects the Lower Datapath to the data pads and busS connects the Lower Datapath to the Upper Datapath and the Memory Address Latches–Mals.

REGISTER FILE is a 138-word, 40-bit, dual-port read, but single-port write register file. It is organized into ten global registers and eight overlapping register windows (see Appendix A). Register R0 is hardwired to zero.

Dst2<39:0> is the second temporary register for the 4-stage pipeline. The result of every instruction that requires writing to the register file is saved here at the end of the Mem stage.

Dst1<39:0> is the first temporary register for the 4-stage pipeline. The result of every instruction that requires writing to the register file (except Load) is saved here at the end of the Exec stage.

IF_LOGIC detects data hazards and instructs Dst1 or Dst2 or both to override the REGISTER FILE and supply the operand or operands.

Mbr<39:0> is the memory buffer register. It stores the data to be written to external memory.

MUXs route immediate constants into the datapath as operands.

BUSBUFA & B<39:0> latch in busA and busB, respectively during $\phi 1$ and drive busA2 and busB2 during $\phi 2$. They serve as buffers between the Operand Supplier (items listed above) and the Functional Unit (items listed below).

EXT_INS<39:0> is the byte extractor and inserter. The SPUR CPU uses this together with the SHIFTER to replace the more traditional 32-bit barrel shifter.

SHIFTER<31:0> is a maximum of 3-bit left shift and 1-bit arithmetic and logic right shift.

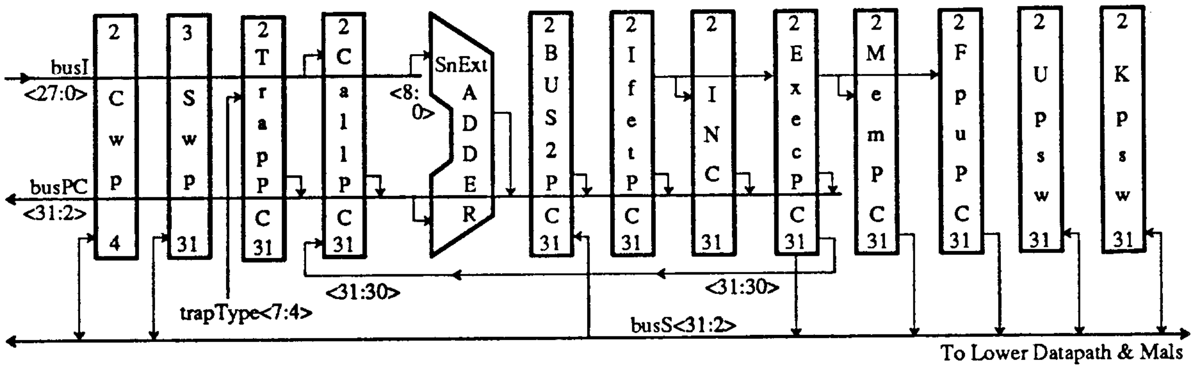


Figure 2-3-2 The SPUR CPU Upper Datapath

The SPUR CPU Upper Datapath is 30 bits wide because its main function is to provide instruction address whose two LSBs are always ignored by the word addressing SPUR memory system. BusI, which contains the instruction, provides the immediate offsets for all the Call, Jump, and Compare-Branch type instructions. BusPC contains the instruction address to be sent to the Instruction Unit. BusS, as explained before, connects the Lower Datapath, the Upper Datapath, and the Memory Address Latches–Mals.

ALU<31:0> performs A + B, A – B, A XOR B, A AND B, and A OR B functions.

BRANCH COND evaluates the branch conditions for all the Compare-Branch type instructions.

BUSSTOD<31:0> is a buffer between the Upper Datapath and the Lower Datapath. The Upper Datapath deposits values into it via busS during $\phi 2$ and it is one of the potential busD drivers during $\phi 4$.

The Upper Datapath, where all the special registers reside, performs all the program control related operations. Below is a brief description of each block in the Upper Datapath, from left to

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;
Exec Stage: Phase 1	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ; BUSBUFA <– busA, if (busI<14>=0) BUSBUFB <– (not busB) else BUSBUFB <– Sign Extend (busI<13:0>) ;
Phase 2	busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of (ALU, SHIFTER, or EXT_INS) <– busA2, Port B of (ALU, SHIFTER, or EXT_INS) <– busB2 ;
Phase 4	busD <– Output Port of (ALU, SHIFTER, or EXT_INS) ; Dst1 <– busD ;
Mem Stage: Phase 1	busPC <– INC ; IfetPC <– busPC, I-Unit <– busPC ;
Phase 3	Dst2 <– Dst1 ;
Wr Stage: Phase 3	busA <– Dst1, busB <– (not Dst2) ; REG_FILE[rd] <– (busA & (not busB)) ; (Note 1)

Table 2-3-1 Register–Register Operation

Load, Return, Read Special, and Write Special type instructions are special cases of Register-Register. Their operations are shown in Appendix A.

Note:

1. To write a register, the true and compliment values are put onto busA and busB respectively.

right as shown in Figure 2-3-2:

Cwp<4:2> is the Current window pointer that points to the register window that is currently in use.

Swp<31:3> is the saved window pointer that points to the memory location where the last overflow register window (pointed to by Swp<9:7>) is saved.

TrapPC<31:2> holds the word address of the trap vector. Whenever a trap occurs, the hardware loads the upper 30 bits of the byte address (hex) 000010T0 into TrapPC<31:2> where T is a function of the trap type.

CallPC<31:2> holds the target address for Call and Jump type instructions. This address is formed by concatenating the 2 MSBs of ExecPC and a 28-bit value provided by the instruction.

ADDER<31:2> calculates the target address for all the Compare-Branch type instructions while the ALU is doing the comparison. The 8-bit offset is first sign extended before adding

Stage/Phase	Actions
Ifet Stage:	
Phase 3	busI <– I-Unit[busPC] ;
Exec Stage:	
Phase 1	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ; Mbr <– (not busB), BUSBUFA <– busA, BUSBUFB <– Sign Extend (busI<24:20> cat busI<8:0>) ;
Phase 2	busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of ALU <– busA2, Port B of ALU <– busB2 ;
Phase 4	busS <– ALU ; Address Pads <– busS
Mem Stage:	
Phase 1	busPC <– INC, busL <– Mbr ; IfetPC <– busPC, I-Unit <– busPC, Data Pads <– busL ;

Table 2-3-2 Store Operation

to the ExecPC.

BUSS2PC<31:2> receives values from the Lower Datapath via busS during $\phi 4$ and drives busI during $\phi 1$ for Jump_Register or Return type instructions.

IfetPC<31:2> normally holds the addresses of the instructions currently in the *Ifet* stage. If the instruction currently in the *Ifet* stage is not in the Instruction Unit, IfetPC must hold onto the next instruction's address until the miss is serviced.

INC<31:2> is the incrementor. It evaluates the next instruction's address for sequential operation.

ExecPC<31:2> and MemPC<31:2> hold the addresses of the instructions currently in the *Exec* and *Mem* stages of the pipeline respectively. This chain of PCs is needed for trap

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;

Exec Stage:
Phase 1
	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ;
	BUSBUFA <– busA,
	if ((*Cond* = *eq_tc*) or (*Cond* = *neq_tc*)) (Note 1)
	BUSBUFB<39:32> <– busI<14:9>
	else if (busI<14> = 0) BUSBUFB <– Zero Extend (busI<13:9>)
	else BUSBUFB <– busB ;

Phase 2	busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of (ALU, BRANCH_COND) <– busA2, Port B of (ALU, BRANCH_COND) <– busB2 ;

Mem Stage:
Phase 1
	if (BRANCH_COND = valid)
	busPC <– ExecPC + Sign Extend (busI<8:0>)
	else busPC <– INC ;
	IfetPC <– busPC, I-Unit <– busPC ;

Table 2-3-3 Compare–Branch Operation

Notes:
1. *Cond* is the conditional field of the instruction (Figure 2-1-1). *eq_tc* and *neq_tc* are two possible branch conditions (see Appendix A).

handling (see Section 2.3.3).

FpuPC<31:2> holds the address of the last FPU (coprocessor) instruction that was send to the FPU. This is needed for parallel operation between the FPU and CPU [HaK86].

Upsw<31:2> is the user processor status word (see Appendix A).

Kpsw<31:2> is the kernel processor status word (see Appendix A).

In order to get a better idea on how each block in the datapaths is being used during each of the four phases at different pipeline stages, you must understand the operation of the Execution Unit. The operations of the Execution Unit under normal conditions and adverse conditions are discussed in Section 2.3.2 and Section 2.3.3, respectively.

2.3.2. Execution Unit Operation–Normal Conditions

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ; CallPC <– ExecPC<31:30> cat busI<27:0> ;
Exec Stage: Phase 2	if (opcode = CALL) Cwp <– Cwp + 1, busS <– ExecPC ; BUSSTOD <– busS ;
Phase 4	busD <– BUSSTOD ; Dst1 <– busD ;
Mem Stage: Phase 1	busPC <– CallPC ; IfetPC <– busPC, I-Unit <– busPC ;
Phase 3	Dst2 <– Dst1 ;
Wr Stage: Phase 3	if (opcode = CALL) { Update Backup Copy of Cwp, busA <– Dst1, busB <– (not Dst2) ; REG_FILE[rd] <– (busA & (not busB))} ;

Table 2-3-4 Call–Jump Operation

In Table 2-3-1 through Table 2-3-4, the four generic instruction types–Register-Register, Store, Compare-Branch, and Call-Jump–are used to show the Execution Unit operation under normal conditions. Similar tables for the rest of the instruction types are presented in Appendix A. In these tables, different phases are separated by a single horizontal line and different pipeline stages are separated by the double horizontal lines. Within a phase, all operations are in parallel unless they are separated by semicolon.

The general timing of the SPUR CPU is summarized in Figure 2-3-2. The SPUR CPU uses a four-phase non-overlap clock [JBH87]. The duration of each phase is 18ns and the non-overlap time is 7ns. The critical paths within each phase must be shorter than the phase duration (18ns) because because all latches in the SPUR CPU latch in data during the falling clock edge. Figure 2-3-2 shows that the critical paths for the register file, the functional unit, the instruction unit, and the external cache all have at least 4ns safety margin. This is probably why most of the CPUs we

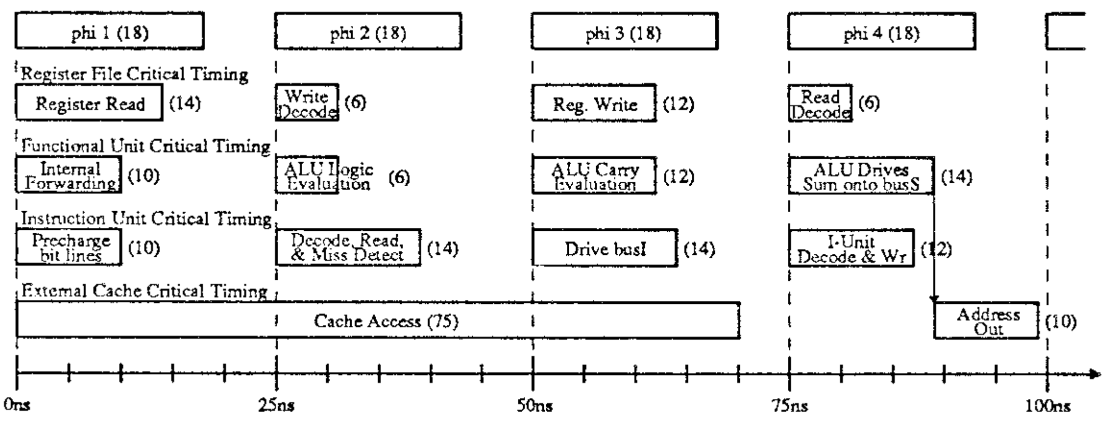


Figure 2-3-3 The SPUR CPU Timing

Each block represents a time interval and the number inside the parentheses is the duration of that time interval in ns. The bit lines for both the register file and the instruction cache array are precharged to high before read and write. For the register file, they are precharged during $\phi 2$ and $\phi 4$. For the instruction cache, they are precharged during $\phi 1$ and $\phi 3$. Notice that almost all actions are triggered by the the clock. There is no self time circuit in the SPUR CPU.

received can run at 80ns cycle time. Figure 2-3-2 also illustrates one big drawback of multi-phase clocking–there is a lot of dead time (horizontal white space between the boxes). Notice that not only do we waste time during the non-overlap time, we also waste the time at the end of each phase due to the requirement for a safety margin.

2.3.3. Execution Unit Operation–Adverse Conditions

Trap Request Handling. A trap request is caused by unusual conditions that arise at run time. Trap request handling refers to the handling of these unusual run time conditions. The detection of these conditions will be discussed in next section. This section explains how the SPUR CPU handles trap requests. A trap request is handled in three steps: (1) branch to a location defined by the trap type, (2) open a new register window, and (3) save the addresses of the instructions that are affected. As illustrated in Figure 2-3-4, all these can be accomplished by the

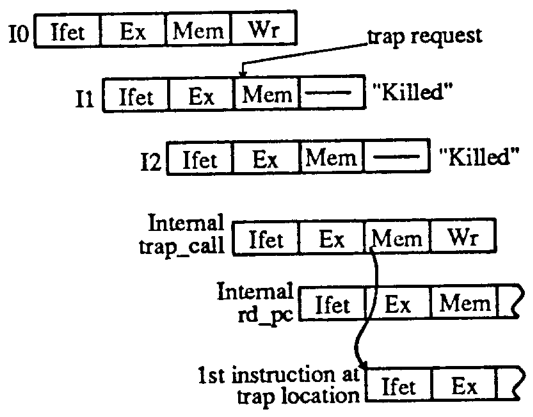


Figure 2-3-4 Pipeline During Trap

In this example, trap request, which is asserted in the *Mem* stage of *I1*, can only due to either *I1* or some external asynchronous unusual condition that happens to kill *I1*. Obviously, *I1's* address must be saved but *I2's* address must also be saved because *I0* may be a delay branch. Due to the assertion of trap request, instructions *I3* and *I4's* positions in the pipeline are replaced by two internally generated instructions: *trap_call*, and *read_pc*. *Trap_call* branches to the trap location, opens a new window, and saves *I1's* address in R10 of the new window. *Read_PC* saves *I2's* address in R16 of the new window.

internal instruction sequence *trap_call* followed by *rd_pc*. Internal instructions are generated by the Instruction Unit (see Figure 2-2-1). As far as the Execution Unit is concerned, *rd_pc* is the same as "rd_special r16, ExecPC" and the only difference between *trap_call* and the regular *call* is that TrapPC is used as the target address instead of CallPC (Figure 2-3-2). Douglas Johnson has evaluated the effectiveness of the SPUR CPU trap architecture [Joh88].

Pipeline Suspension. In theory, the SPUR CPU pipeline can be suspended for an infinite number of cycles as illustrated in Figure 2-3-5. Notice that every instruction in the pipeline is suspended unlike the situation shown in Figure 2-2-5, in which only the issue of new instruction is suspended. Therefore, we refer to the pipeline suspension in Figure 2-3-5 as *Global Pipeline Suspension* and the situation shown earlier in Figure 2-2-5 *Partial Pipeline Suspension. Global Pipeline Suspension* is used to handle external cache miss and coprocessor busy conditions. *Partial Pipeline Suspension* is used to handle internal instruction cache miss because as explained in Section 2.2.3–*Global Pipeline Suspension* cannot be used due to potential deadlock condition.

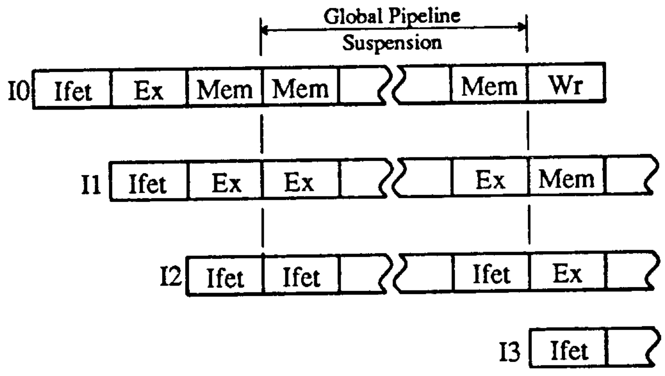


Figure 2-3-5 Global Pipeline Suspension

The SPUR CPU pipeline can be suspended for two reasons: coprocessor (FPU) busy, or cache miss. The first reason is out of the scope of this chapter and is explained in [HaK86]. The second reason can be explained using this figure by assuming *I0* as a Load or Store type instruction. If *I0* causes an external cache miss, then the SPUR CPU pipeline will be suspended until the data is valid.

2.4. The SPUR CPU Controller

The major design theme behind the SPUR CPU controller is decentralization. As described in Section 2.2 above, the Instruction Unit and Execution Unit have their own controllers. Furthermore, using internal instructions *miss*, *trap_call*, and *read_pc*, the Instruction Unit simplifies the control of the Execution Unit by reducing complex control functions such as instruction miss and trap handling into simple instruction sequences that can be executed uniformly by the Execution Unit's four-stage pipeline. Within the Execution Unit, the control responsibility is further delegated to three independent parts:

Priority	Trap Type	Vector (Hex)	Side Effects	Description
0 (highest)	RESET (0)	1000	kpei	power-on initiation
1	ERROR (1)	1010	kpei	bus fault or hardware error
2	WIN_OV (2) WIN_UN (3)	1020 1030		window overflow window underflow
3	FAU_IN (4)	1040	k	page fault or interrupt
4	FPU_EX (5)	1050		FPU exception
5	RUN_ER (6)	1060		Run time software errors: illegal opcode kernel mode violation LISP pointer type violation
6	TAG_TR (7)	1070		Run time tag violation: generation trap LISP data type violation
7 (lowest)	IN_OV (8)	1080		Integer overlow
7 (lowest)	CMP_TR (9)	1090	k	cmp_trap instruction

Table 2-4-1 The SPUR CPU Trap Types

In the table above, illegal opcode includes all the FPU opcodes whenever the the FPU is disabled. LISP pointer type violation occurs when tag fails the "CONS or NIL" test. LISP data type violation occurs when the tags fail the "Both operands are FIXNUM" or "Both operands are FIXNUM or CHAR" test.

Side Effects:

k - changes to kernel mode	e - turn off ERROR detection
p - changes to physical mode	i - disable the Instruction Unit

Cache Controller Interface

This module communicates with the Cache Controller and is out of the scope of this chapter. The cache controller interface is described in [WEG87].

Trap Logic

This module detects the unusual conditions (Table 2-4-1), prioritizes them, and determines which trap type to take. The Trap Logic is discussed in Section 2.4.1.

Control Unit

This module controls the Execution Unit' Upper Datapath and Lower Datapath. The Control Unit is discussed in Section 2.4.2.

2.4.1. Trap Logic

The Trap Logic block must be able to detect thirteen different trap conditions (refer to column "Description" of Table 2-4-1). Once these conditions are detected, they are grouped into ten different trap types that are prioritized into eight priority levels. Each trap type has its own 4-bit trap number which is fed into TrapPC<7:4> (Figure 2-2-2) to form an unique trap vector. The SPUR CPU can be programmed to selectively ignore most of these unusual conditions by writing to the Upsw and the Kpsw (see Appendix A). In fact, whenever the CPU takes a trap, the hardware disables any further traps by turning off the AllEn bit in the Kpsw.

Figure 2-4-1 shows that these enabling, detection, prioritizing, and grouping functions are implement by five logic blocks separated by latches. Trap Enable and Trap Type are the only logic blocks implemented by PLAs. Trap Enable updates the on/off status of the various traps according to the contents of Kpsw and Upsw. Trap Type groups all the detected unusual conditions into trap types and decides which trap type to take according to the priority shown in Table 2-4-1. The three Trap Request blocks are implemented in random logic and together they generate the trap request (assert the signal *trapRequest*) whenever one or more unusual conditions are detected. Pre-Trap Request does the initial set up, Trap Request (AND) and Trap Request (OR) are analogous to the AND and OR plane of a PLA. The reasons why they are not combined into

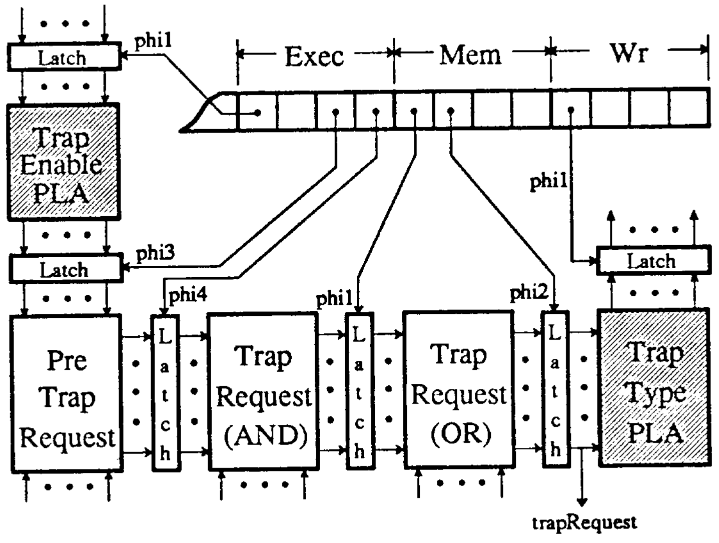


Figure 2-4-1 Trap Logic Block Diagram

The pipeline diagram shows how the Trap Logic operates at different time points with respect to any instruction that may trap. Each pipe stage (clock cycle) consists of four phases and the falling clock edge of these phases are used by the latches to latch in the intermediate results. This is a simplified view because in the real hardware, the clock phases are sometimes "ANDed" with some other control signals before being used by these latches as trigger signals. Out of the five combinational blocks, only the two shaded blocks are implemented by PLA's. Others are custom logic due to timing or area constraints or both.

one logic block are the same as the reasons for not combining the Output and State logic blocks in Figure 2-2-3–it makes the design easier to understand *and* reduces the input-output latency.

2.4.2. Control Unit

The Execution Unit's Control Unit (Figure 2-4-2) is divided into two parts: the Master Control and the Local Decoding Logic. Master Control decodes and buffers the opcode into high level control signals. The Local Decoding Logic then decodes these high level control signals into low level control signals that control the datapath. The coprocessor interface is part of the Master Control but will not be discused here. In a simplified view, the Master Control consists of two parts:

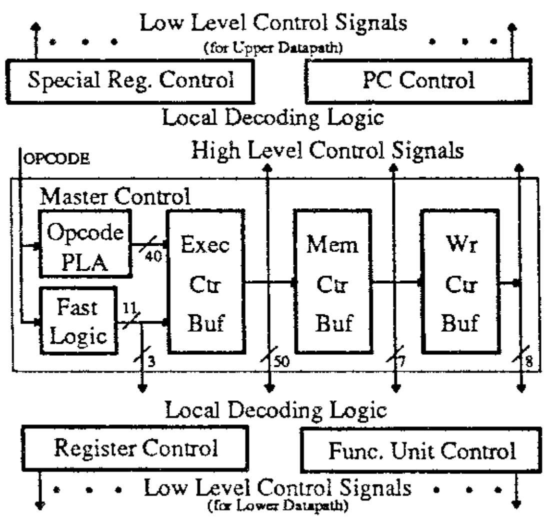


Figure 2-4-2 The Control Unit Block Diagram

The Control Unit can be divided into two parts: Master Control and Local Decoding Logic. In the layout, the Master Control resides in the center of the chip while the Local Decoding Logic blocks are scattered along the Upper Datapath and Lower Datapath close to where the low level control signals are needed.

Opcode PLA and Fast Logic

These modules decode the opcode into high level control signals.

Sequencer

This module sequences the high level signals. Exec-Ctr-Buf, Mem-Ctr-Buf, and Wr-Ctr-Buf, which contain latches and simple logic, together combine and buffer the high level signals into three sets that are responsible for the control of the pipeline stages: Exec, Mem , and Wr, respectively.

There are only three high level signals for the *Ifet* stage of the pipeline. However, these must be provided by fast logic because the opcode arrives at the Control Unit during $\phi 3$ of the *Ifet* stage and these three high level control signals must be valid during the next clock phase ($\phi 4$) of the same stage. The *Exec* stage is the busiest, which is reflected by the large number of high level control signals (50) needed to control it. The *Mem* stage is a null stage except for memory access instructions and requires only seven high level control signals. Finally, the *Wr* stage operation is not that much different for different instructions and it requires only eight high level control signals.

The local decoding logic is organized into four blocks, each of which is specialized in controlling one local area of the datapath. The four blocks, as shown earlier in Figure 2-4-2, are:

Register Control

This block controls the register file and temporary registers: Dst1, Dst2, and Mbr.

Functional Unit Control

This block controls the functional units: Byte Extractor Inserter, the Shifter, and the ALU.

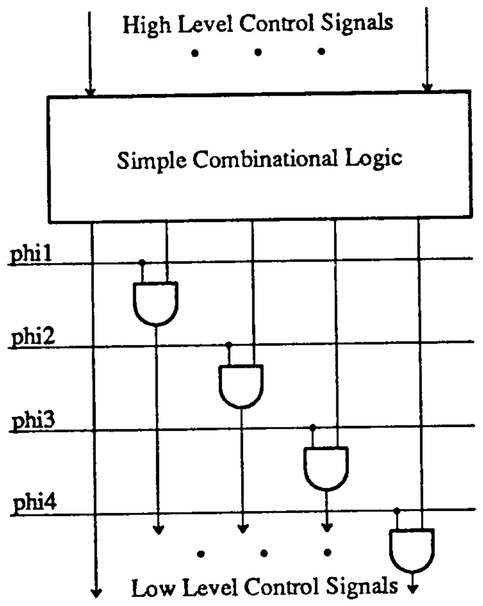

High Level Control Signals
Simple Combinational Logic
phi1
phi2
phi3
phi4
Low Level Control Signals

Figure 2-4-3 Local Decoding Logic

The Simple Combinational Logic blocks are located close to the datapath where the low level control signals are needed. Each of these logic block generally consists of single level of random logic and it also serves as a buffer between the high level and low level control signals.

Special Control

This block controls the special registers: Cwp, Swp, Ins, Kpsw, and Upsw.

PC Control

This block controls the program counter generation logic: ADDER, INC, and the various PCs.

These four blocks are implemented in the generic structure shown in Figure 2-4-3 in which the high level control signals are decoded by the Simple Combinational Logic into low level control signals. There are never more than two levels of logic in the Simple Combinational Logic and its outputs are then either used directly by the datapath or "ANDed" with one of the four phases before they are used.

2.4.3. Controller Design Insights

Parts	Inputs	Outputs	Product Terms	Logic Gates	Implementation Effort (man-month)
Trap Logic: Trap Enable	23	12	14	—	0.25
Trap Type	11	9	11	—	0.25
Pre Trap Req.	14	6	—	24	0.50
Trap Req. (AND)	24	16	—	18	0.50
Trap Req. (OR)	16	10	—	19	0.50
Control Unit: Opcode PLA	8	40	68	—	0.50
Fast Logic	18	14	16	—	0.50
Reg_Ctr	10	9	—	26	0.50
Func_Ctr	19	13	—	26	0.50
Pc_Ctr	17	14	—	41	0.50
Spec_Ctr	27	19	—	23	0.50
Total	—	—	109	177	5.00

Table 2-4-2 The Execution Unit Controller Design Metrics

The implementation metrics of the Trap Logic and the Control Unit are summarized in Table 2-4-2. The implementation of the SPUR CPU Controller can be considered as an experiment which shows that by using internal instructions (Example: *trap_call*) and some satellite logic blocks (Example: Trap Logic), the main control engine that controls the datapath (Example: Control Unit) can be reduced to a simple N-Stage sequential logic structure (Figure 2-4-2) where N is the pipeline length of the machine (Example: N=4 for the SPUR CPU). The term "N-Stage sequential" is used because the outputs depend on the inputs of the previous N cycles only. There is *no* feedback in this structure and therefore it is *not* a state machine. This N-Stage sequential logic block has well defined inputs–the instruction set and internal instructions (mainly the opcode)–and outputs–datapath control signals.

While reducing control functions into internal instruction sequences and designing the satellite logic blocks may still require some human ingenuity, CAD designers should be able to provide CAD tools that can generate the N-Stage sequential logic automatically. Ideally, a VLSI designer would like to have a set of CAD tools that can partition this N-Stage sequential logic into Master Control and Local Decoding Logic, generate them automatically, and route the connections between the two. An optimum solution is hard to define here but as most VLSI designers can tell you, the optimum solution is not necessary as long as the Master Control, the Local Decoding Logic, and the routing between them meet the area, timing, and power constraints.

One final point is that reducing complex control functions by internal instructions gives similar benefits to those found in microprogramming. However, in microprogramming, every instruction (no matter how simple) is turned into a sequence of microinstructions. On the other hand, in the internal instruction approach, *only* complex control functions are turned into sequences of internal instructions. These internal instruction sequence can be quite short because only the most critical steps need to be implemented. The rest of the steps can easily be coded as software routine using the regular RISC–style instructions that are similar to traditional microinstructions. In other words, unlike microprograming, internal instructions allows RISC–style machine to handle complex situation without introducing an overhead on all other instructions. Furthermore, internal instructions will not greatly increase the complexity of the instruction decoding unit because they are similar (if not the same) to those already exist in the RISC–style instruction set. For example, in the SPUR CPU, internal instruction *trap_call* is similar to the regular instruction *call* and *rd_pc* is the same as regular instruction *rd_special ExecPC*.

2.5. REFERENCES

[Dun86] R. R. Duncombe, *The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor*, Master Report, EECS Department, University of California, Berkeley, CA 94720, August, 1986.

[HaK86] P. Hansen and S. Kong, "SPUR Coprocessor Interface Description", Report No. UCB/Computer Science Dpt. 87/308, Computer Science Division, EECS Department, University of California, Berkeley, October 1986.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Dissertation, Computer Science Division, EECS Department University of California, Berkeley, Fall 1987.

[JBH87] D. K. Jeong, G. Borriello, D. A. Hodges and R. H. Katz, "Design of PLL-Based Clock Generation Circuits", *IEEE Journal of Soloid-State Circuits SC-17*, 3 (June 1987).

[Joh88] D. Johnson, "Trap Architectures for Lisp Systems", Report No. UCB/Computer Science Dpt. 88/470, Computer Science Division, EECS Department, University of California, Berkeley, November 1988.

[Lee86] D. Lee, *Datapath Design Considerations for a High Performance VLSI Multiprocessor* , Master Report, EECS Department, University of California, Berkeley, CA 94720, November, 1986.

[LiH83] H. Liiberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects", *Comm. of the ACM 26*, 6 (June 1983).

[Tay85] G. S. Taylor, "SPUR Instruction Set Architecture", in *Proceedings of CS292i: Implementation of VLSI Systems*, R. Katz (editor), Computer Science Division, EECS Department, University of California, Berkeley, September 1985.

[Tay86] G. Taylor et al., *Evaluation of the SPUR Lisp Architecture*, The 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 2-5, 1986.

[Ung84] D. Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", *ACM Software Engineering Notes/SIGPLAN Notices Notices Software Engineering Symposium on Practical Software Development Environments*, Pittsburg, April, 1984.

[WEG87] D. Wood, S. Eggers and G. Gibson, "SPUR Memory System Architecture", Report No. UCB/Computer Science Dpt. 87/394, Computer Science Division, EECS Department, University of California, Berkeley, December 1987.

[ZHH88] B. Zorn, P. Hillfinger, K. Ho, J. Larus and L. Semenzato, "Features for Multiprocessing in SPUR LISP", Report No. UCB/Computer Science Dpt. 88/406, Computer Science Division, EECS Department, University of California, Berkeley, March 1988.

Chapter 3

THE SPUR CPU EXPERIENCE

The biggest performance enhancement is achieved when going from a non-working system to a working system.

John Ousterhout, 1988

In this chapter, I will talk about the SPUR CPU experience. The lessons I learned from this experience is the foundation of my view on the systematic approach to microarchitectural design (Chapter 5) and the future trends (Chapter 6).

3.1. From Chip to System

SPUR's mission is not only to build a VLSI chip but to build a system around three custom VLSI chips. As far as the CPU is concerned, the implications of this ambitious mission are:

- The CPU specifications do not come from expert VLSI designers whose goal is to build the fastest and the most innovative CPU. The specifications come from the system goals.
- We must increase our chance of having a working chip by using as much proven "technology" as possible.
- Instead of experimenting with architectural ideas, we must implement certain features that

building block for the SPUR system.

The first implication is partly responsible for our relatively slow cycle time of 100ns. We did not set the SPUR CPU cycle time goal any faster than 100ns because the SPUR memory system [WEG87] and the SPUR bus [Gib87] cannot run any faster. In the following sections, I will explain the other implications in more details.

3.1.1. The Russian Approach

The phase "use proven technology" means we tried to build the CPU based on previous experience. I called this the "Russian approach" because the Soviet space program is a good example of not feeling ashamed of using old but proven technology. Since our project goal is to build a system based on the SPUR CPU chip, we decided to increase our chances of having a working chip by using as many proven ideas as possible from the two previous generations Berkeley RISC processors: RISC I [Pat82] and RISC II [Kat83], and SOAR [Ung84b]. As mentioned in Chapter 2, the SPUR CPU differs in these five aspects:

Internal Instruction Cache

The SPUR CPU has an on-chip 512-byte direct-mapped instruction cache organized into

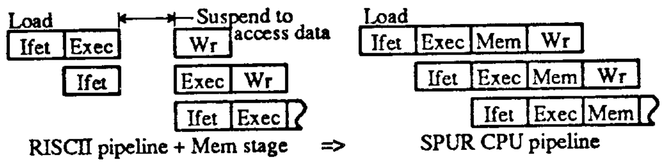


Figure 3-3-1 RISC II Pipeline vs. SPUR CPU Pipeline

Both pipelines consist of instruction fetch (Ifet), execution (Exec), and register write (Wr). The SPUR CPU internal instruction cache allows Ifet in parallel with data access (Mem). Data conflicts in both pipelines are resolved by internal forwarding and branch conflicts are resolved by a single cycle delay branch. The SPUR CPU, however, requires two internal forwarding paths (RISC II and SOAR only require one) due to the extra pipe stage.

sixteen blocks with eight instructions per block.

Four-Stage Pipeline

RISC II and SOAR use the same three-stage pipeline that is shown in Figure 3-1-1. The internal instruction cache essentially provides the SPUR CPU an extra port to memory and enables us to add a memory access stage (Mem) to the RISC II pipeline. This results in the SPUR CPU 4-stage pipeline that does not have to be suspended for LOAD.

Support for LISP

The SPUR CPU supports LISP by three types of hardware tag checking [Tay86]: data type checking for general operations, pointer type checking for list operations, and generation checking for garbage collection [Ung84a] [ZHH88].

Cache Controller Interface

In order to support multiprocessing, the SPUR CPU must communicate constantly with the Cache Controller chip via a cache controller interface [WEG87].

Parallel Coprocessor (FPU) Interface

The SPUR CPU supports a coprocessor interface which allows the FPU to operate in

Feature i	Extra Pipeline Stage (Mem)	On Chip I-Cache	LISP Support	Floating Pt. Support
SPUR CPU vs. SPUR CPU – Feature i	1.12	1.30	1.73	30.0
Performance Improvement (%)	12%	30%	73%	2900%

Table 3-1-1 Contributions to Performance

The performance improvement due to each feature (Row 2) is estimated by comparing the performance of the SPUR CPU against the performance of an imaginary, stripped down, SPUR CPU (Row 1). The details of this analysis can be found in Chapter 4. All numbers are only approximations because they are sensitive to the frequencies of different instructions and the quality of the compiler.

parallel with the CPU [HaK86].

These features' contributions to performance are evaluated in Chapter 4. The results are summarized here in Table 3-1-1. The stripped down CPU for Column 1 uses the RISC II 3-stage pipeline. The stripped down CPU for Column 2 does not have an on-chip instruction cache. Since it is difficult to implement the SPUR 4-stage pipeline without the instruction cache, this stripped down CPU also uses the RISC II 3-stage pipeline. The stripped down CPU for Column 3 does not support hardware tag checking. Similarly, the stripped down CPU for Column 5 does not support the FPU interface. For example in Column 3, we estimate the SPUR CPU to run LISP programs 1.73 times (73%) faster than a similar CPU without hardware tag checking. The performance improvement due to the multiprocessing support features is not included because we believe multiprocessing performance depends more on the shared bus utilization and cache performance [Kat85] [EgK88] than the features in the CPU.

3.1.2. SPUR CPU System Features

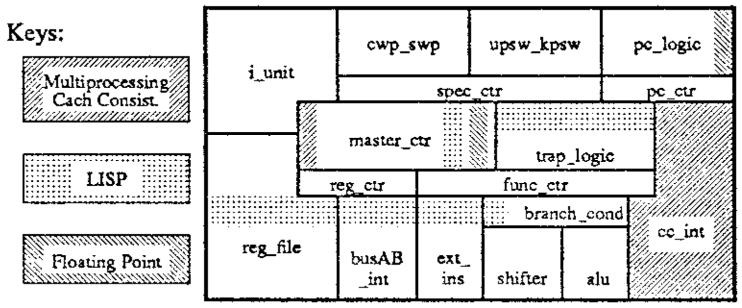


Figure 3-1-2 Impact of the System Features–Graphical

This illustrates qualitatively which portion of which module is affected by the system features.

The SPUR CPU system features are essential to the SPUR system's functionality and performance. The SPUR CPU system features come from three sources [Tay85]:

Multiprocessing and Cache Consistency Support

This requires seven load instructions, three store instructions, and a cache controller interface. Although all load or store instructions are alike internally, the CPU must request different cache operations [KEW85] [WEG87] via the cache controller interface.

LISP Support

This requires four special load instructions, one read tag instruction, one write tag instruction, and eight extra tag bits in the datapath. This tag architecture also adds six branch and five trap conditions.

	Multiprocessing		LISP		Floating Point		Total
Control PLA Outputs	6/54	11%	4/54	7%	3/54	6%	24%
Control PLA Products	2/84	2%	2.2/84	3%	4/84	5%	10%
Chip Area (mm x mm)	2.2/57	4%	4/57	7%	6/57	10%	21%
Transistors (x 1000)	0.8/115	1%	9.9/115	9%	0.4/115	0%	10%
Number of Signal Pins	15/156	10%	8/156	5%	37/156	24%	39%

Table 3-1-2 Impact of the System Features–Quantitative

The first row shows that the master control PLA has 54 outputs. Six of these 54 outputs (11%) are used to control the multiprocessing supporting features, four outputs (7%) are used to control the LISP supporting features, and three outputs (6%) are used to control the FPU supporting features. The total is that 24% of the master control PLA outputs are used to control the system features. Similarly, the second, third, fourth, and fifth row show that the system features are responsible for 10% of the master control PLA product terms, consume 21% of the total active area, 10% of the total transistors, and 39% of the total signal pins.

Floating Point Support

This requires eight load instructions, four store instructions, and a coprocessor interface. The coprocessor FPU also adds two branch conditions and one trap condition.

The end results are 19 load instructions, seven store instructions, and a 40-bit non-standard (not 32-bit) datapath. Furthermore, the CPU must be able to handle 20 branch conditions, nine trap conditions, and support two non-trivial off-chip interfaces. The impact of these features on resources are evaluated quantitatively in Chapter 4. The results are illustrated graphically in Figure 3-1-2 and summarized quantitatively in Table 3-1-2. The complexity of these features is not additive, it is multiplicative! These features must be simulated at the behavioral level to ensure they are implemented correctly. Since the complexity of these features is multiplicative, their simulation effort is also multiplicative.

3.1.3. Simulation Strategy

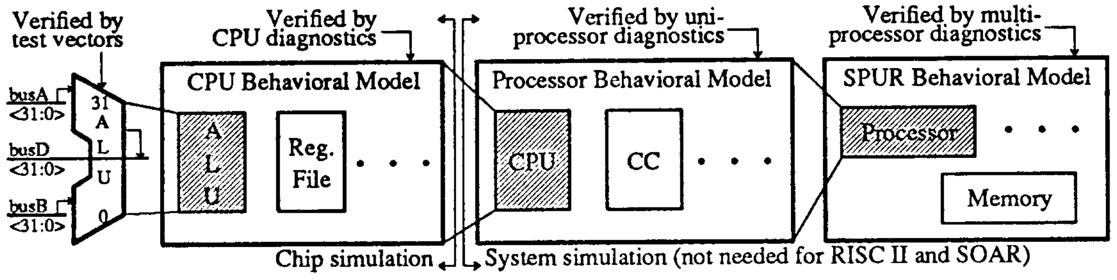

Verified by test vectors
Verified by CPU diagnostics
Verified by uni-processor diagnostics
Verified by multi-processor diagnostics
busA <31:0>
busD <31:0>
busB <31:0>
31
ALU
0
CPU Behavioral Model
ALU
Reg. File
Processor Behavioral Model
CPU
CC
SPUR Behavioral Model
Processor
Memory
Chip simulation
System simulation (not needed for RISC II and SOAR)

Figure 3-1-3 Behavioral Simulation Strategy

The behavioral model of the SPUR system was built from bottom up. The lowest level modules, for example the ALU, were first modeled and a set of test vectors was written to verify its functionality. Once these lowest level modules were verified, they were grouped together to form higher level composite modules following the hardware's hierarchical organization. This "verify and merge" process was repeated until we had the composite module of the SPUR multiprocessor workstation.

At the behavioral level, the complete SPUR system is described in the ISP' hardware description language [KWG87]. As shown in Figure 3-1-3, behavioral simulation is divided into two categories: chip simulation and system simulation. The SPUR CPU behavioral model, which is discussed in more details in Section 3.2, is not only used for chip level simulation but also used as a building block for the processor board behavioral model for system level simulation. This is necessary because not only do we need to verify each individual chip, we also need to verify the interactions among the chips on the processor board and eventually the interactions among processor boards in the multiprocessor. If our mission was just to build a CPU chip–the missions of RISC I, RISC II, and SOAR–the chip simulation would have been sufficient. Since our mission is to build a system, however, we must also complete the system simulation. After the behavioral simulation is completed, the behavioral test vectors for the lowest level modules are converted to switch level vectors to simulate the corresponding layout modules. The layout modules are then merged to form the layout of the CPU which is verified by the same diagnostic programs used for behavioral simulation.

Table 3-1-3 summarizes the switch level simulation we performed. If we just wanted to prove that we knew how to build a CPU chip, the first column is probably all the switch simula-

	General CPU	Cache Controller Interface	Tags & Traps	FPU Interface	Bootstrap Programs	Total
Cycles of Diagnostics	13,113 (24%)	13,875 (25%)	8,675 (16%)	1,543 (3%)	18,310 (33%)	55,516 (100%)
Man-Month of Effort	0.5 (14%)	1.0 (29%)	0.5 (14%)	0.5 (14%)	1.0 (29%)	3.5 (100%)

Table 3-1-3 Switch Level Simulation Summary

The first column verifies the basic function of the CPU. The next three columns verify the utility features. Boot programs are simple bootstrap routines that are used to bootstrap the SPUR processor board.

tion we would have needed. Columns 2 through 5, which constitute 76% of the cycles and 86% of the effort, represent the extra simulation we have to do in order to build a chip to be used in a system. This seems to indicate that it is three to six times the effort to build a system like SPUR than just a chip like RISC II and SOAR. Despite the large amount of effort we spent on simulation, simulation is only the tip of an iceberg–the rest of the iceberg is the design process discussed in the following section.

3.2. The SPUR CPU Design Process

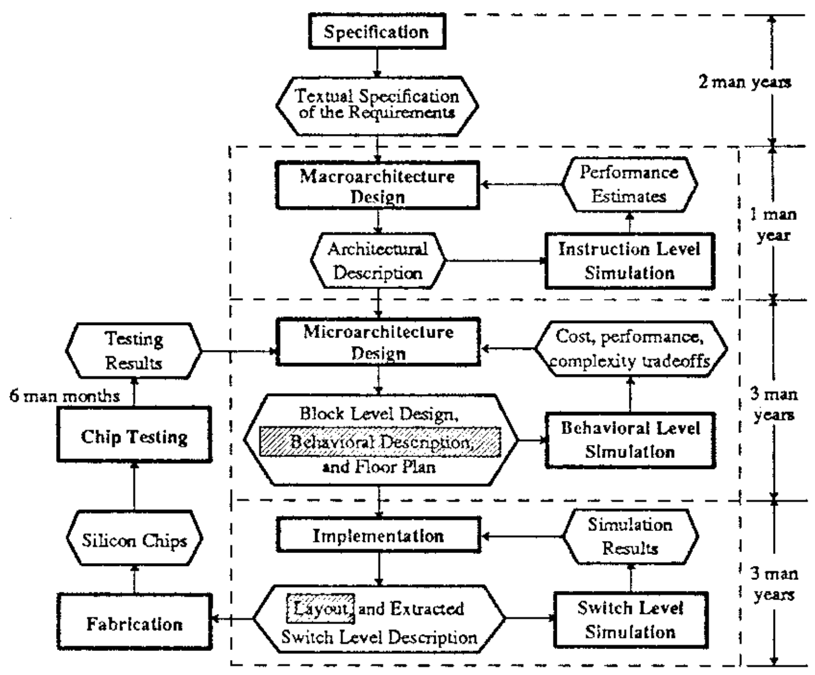

Specification
Textual Specification of the Requirements
2 man years
Macroarchitecture Design
Performance Estimates
1 man year
Architectural Description
Instruction Level Simulation
Testing Results
Microarchitecture Design
Cost, performance, complexity tradeoffs
6 man months
Chip Testing
Block Level Design, Behavioral Description, and Floor Plan
Behavioral Level Simulation
3 man years
Silicon Chips
Implementation
Simulation Results
3 man years
Fabrication
Layout and Extracted Switch Level Description
Switch Level Simulation

Figure 3-2-1 The SPUR CPU Design Process

In this figure, rectangular boxes represent steps in the design process while hexagonal boxes represent products of the design steps. This is a simplified view because we do not show all the interactions between different steps that make iterations necessary. The behavioral description and the layout are the two most important products from the microarchitecture design and implementation steps respectively.

A simplified view of the SPUR CPU design process is shown in Figure 3-2-1. The major steps are: Specification, Macroarchitectural Design, Microarchitectural Design, and Implementation.

Specification

We studied multiprocessor issues and the tradeoffs between different multiprocessor configurations and selected the shared bus configuration (Figure 1-1-1(a)) to fulfill our initial performance goal. Each SPUR processor node was then partitioned into a large cache memory and three custom VLSI chips: CPU, CC, and the FPU (Figure 1-1-1(b)). This Specification step created a textual specification of the requirements for the SPUR CPU chip.

Macroarchitecture Design

We translated the textual specification into the instruction set, interfaces specifications, and algorithms for the Cache Controller, the Floating Point Unit, and the CPU chips. In order to specify each chip in more detail, we further divided each chip into modules which I called the *macro-modules*. The Instruction Unit and the Execution Unit are examples of macro-modules in the SPUR CPU. As far as the CPU is concerned, this step created a machine readable architectural description which enabled us to perform instruction level simulation to evaluate the effectiveness and verify the correctness of this macroarchitecture.

Microarchitecture Design

The microarchitect studied the interactions among the macro-modules and described the interactions in an behavioral description of the SPUR CPU. In describing the SPUR CPU behavior, the microarchitect also expanded the macro-modules into smaller modules which I call *micro-modules* and produced a block level design and a floor plan. The behavioral description, which is the SPUR CPU behavioral model shown earlier in Figure 3-1-3, models the microarchitecture and must be verified by behavioral level simulation.

Implementation.

The behavioral description was translated into logic modules either automatically by CAD tools (PLA) or by the logic designers (gates and latches). The circuit designer then implemented these logic modules by transistors and wires that were eventually translated into layout. The layout was then extracted by a circuit extractor to produce the switch level description that can be verified by switch level simulation.

Strictly speaking, fabrication and testing are not part of the design process. They are included in Figure 3-2-1 for the sake of completeness. Furthermore, in practice the SPUR CPU design process is not a pure sequential process. A lot of work–especially among the microarchitecture design and the implementation steps–can be and were done in parallel. Consequently, the total nine man years required by these four steps were accomplished in approximately four years by five graduate students.† The initial SPUR study was done in the Fall 1983 and the first version of the SPUR CPU was fabricated in the Fall 1987.

The macroarchitectural design and implementation steps of the SPUR CPU design process are discussed in more details in [Tay86] and [Lee86]. This thesis will focus on the microarchitecture design step. The most important product of the microarchitectural design step is the machine readable behavioral description of the CPU. This behavioral description models the microarchitecture and was shown earlier as the SPRU CPU Behavioral Model in Figure 3-1-3 in relation to the behavioral model of the SPUR system. Section 3.2.1 will discuss the construction of this behavioral model. Section 3.2.2 discusses how this behavioral model can be used as a formal specification for logic and circuit designers. Section 3.2.3 discuss how this behavioral model can be used for layout verification. Finally, Section 3.2.4 summarizes some important observations from the SPRU CPU design process.

† George Taylor is the macroarchitect, Shing Kong is the microarchitect, and Dave Lee is the chief circuit designer. Wook Koh and Rich Duncombe are part time logic and circuit designers and Mark Hill is our macroarchitecture consultant.

3.2.1. The Construction of the SPUR CPU Behavioral Model

The SPUR CPU behavioral model [Kon89] was developed using the N.2 hardware modeling package [EEE85]. In the N.2 environment, a piece of hardware can be modeled in two different ways (see Figure 3-2-2):

(1) As a primitive module that is described in ISP', or

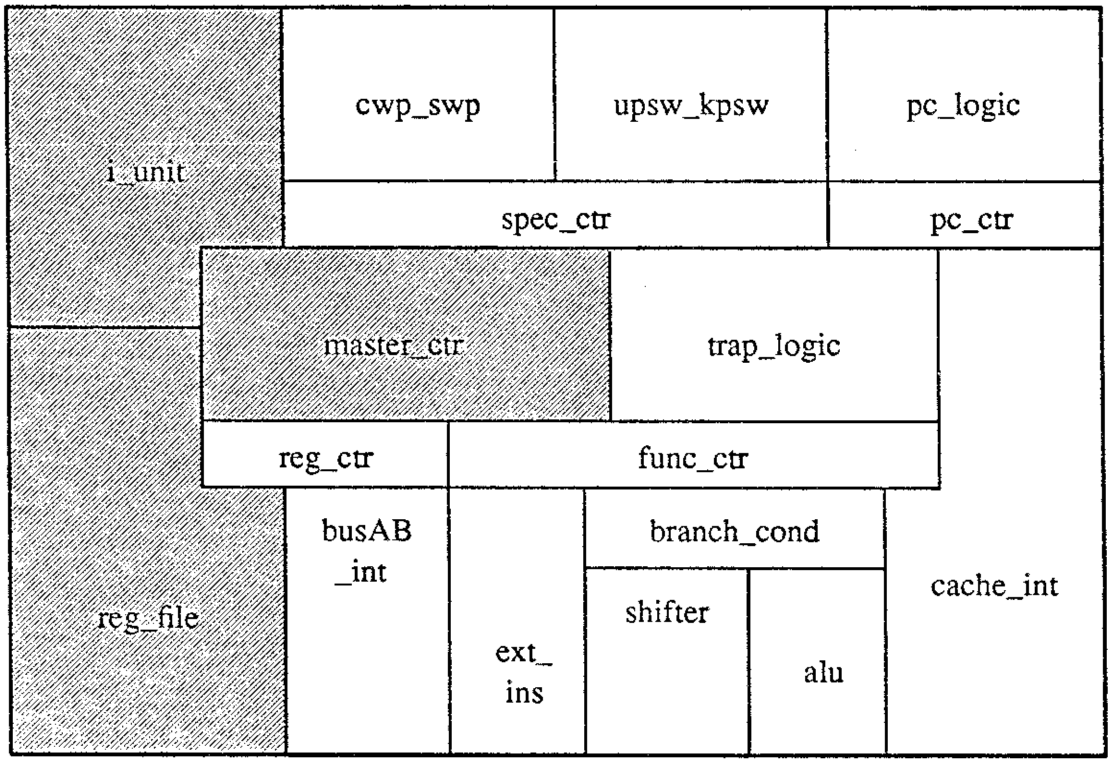


Figure 3-2-2 The Structure of the SPUR CPU Behavioral Model

The three shaded blocks: i_unit, reg_file, and master_ctr are *composite modules*. All other blocks are *primitive modules*. *Primitive module* is a hardware description written in ISP'. *Composite module* is a collection of primitive modules connected together by a topology file. The CPU behavioral model [Kon89] is by definition a high level composite modules which consists of both composite and primitive modules.

(2) As a composite module that consists two or more primitive modules connected together by a topology file.

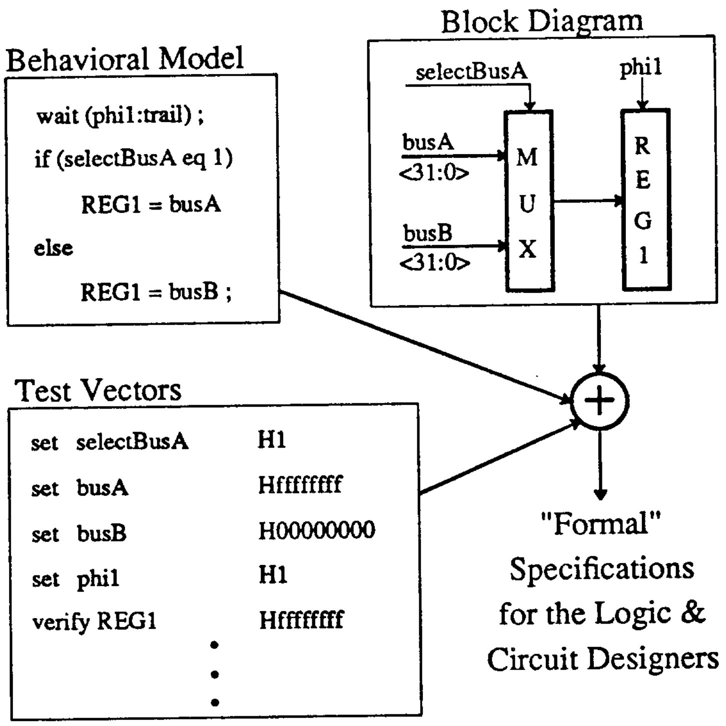
The ISP' hardware description language provides the designer a way to model the behavior while the topology file facility provides the designer a way to model the structure.

The SPUR CPU behavioral model was built using the "meet-at-the-middle" approach. The desired behavior of the microarchitecture was first determined informally and then it was decided how this behavior can be implemented structurally. The next step was to divide the conceived structure hierarchically into modules. Once the modules were defined, the SPUR CPU behavioral model was built from bottom-up. The lowest level modules were described by the ISP' hardware description language as primitive modules and a set of test vectors was written to test each module's functionality. Once these lowest level modules were tested, they were grouped together to form higher level composite modules following the hardware's hierarchical organization. This test and merge process was repeated until the CPU composite module was built. Consequently, this CPU composite module actually models both the behavioral and structural characteristics of the SPUR CPU microarchitecture although it is only called the CPU behavioral model.

The structure of the SPUR CPU behavioral model is shown in Figure 3-2-2. The only composite modules are: *reg_file*, *master_ctr*, and *i_unit*, which model the register file and temporary registers, the master control, and the instruction unit, respectively. Most of the modules in the SPUR CPU behavioral model are primitive modules because we try to ease the behavioral model to silicon transformation by keeping a one-to-one correspondence between the behavioral modules and the prospective layout modules. Consequently, most of the behavioral modules are therefore simple components–ALU, SHIFTER and so on–whose behavior is well understood and can be described easily in single ISP' primitive module.

3.2.2. Behavioral Model–Formal Specification for Logic and Circuit Designers

The behavioral model of the SPUR CPU was not only used for CPU chip and SPUR system verification (Section 3.1.3), but was also used as a "formal" specification for logic and circuit designers. This is illustrated by the example in Figure 3-2-3. The microarchitect prepared a block diagram that showed the input output interfaces and the logic at register transfer level for each module in the behavioral model. Furthermore, as mentioned earlier in Section 3.2.3, a set of test vectors was created for each module to exercise its functionality during the construction of the



```
wait (phi1:trail) ;
if (selectBusA eq 1)
     REG1 = busA
else
     REG1 = busB ;
```

```
set   selectBusA      H1
set   busA            Hffffffff
set   busB            H00000000
set   phi1            H1
verify REG1           Hffffffff
                 •
                 •
                 •
```

Figure 3-2-3 Formal Specification for Logic and Circuit Design Example

In this simple example, there are two 32-bit busses (busA and busB), a two by one multipler (MUX), and a 32-bit register (REG1). The signal *phi1* is a clock signal and *selectBusA* is a control signal. Register REG1 will latch in the value on either busA or busB during every *phi1*. This behavior is described textually by the behavioral model, illustrated graphically by the block diagram, and verified by the test vectors.

behavioral model. The behavioral model of module, the block diagram, and the test vectors together form the specification of the module to be implemented by the logic and circuit designers.

Ideally, we would like to have CAD tools to generate the block diagram automatically from the behavioral model, or vice versa. We would also like to have another tool to generate the test vectors automatically from either the behavioral model or the block diagram. Finally, we would like to have some module generators to generate the layout for us automatically from this formal specification. Alas, such ideal CAD tools were not available for SPUR. The block diagram, the test vectors, and most of the logic design, circuit design, and layout had to be done by hand. The only layout that can be generated automatically from the behavioral description was simple control blocks from the PLA generators.

3.2.3. Behavioral Model–An Aid for Switch Level Simulation

The layout created by hand must be verified by switch level simulation to ensure it is functionally correct. The verification process shown in Figure 3-2-4 ensures the layout behaves the same as specified in the behavioral model. As discussed earlier, there is a one-to-one correspondence between the behavioral module and the layout module and every behavioral module has its own set of test vectors. By running this set of test vectors through the behavioral simulator, we can trace the results, and do a simple format conversion to obtain the switch level test vectors for the layout of that module. After the layout of all the modules was tested individually, they were merged to form the CPU chip.

The behavioral model of the CPU is not verified by test vectors. It is verified by diagnostic programs written in SPUR assembly language. Since the N.2 behavioral simulator supports simulated memories, all we had to do was to generate a memory image using the SPUR assembler and linker, load this image into the simulated memory, and start the execution. This simulated execution was traced and the trace informations was then coverted to switch level test vectors for global switch level simulation of the CPU chip.

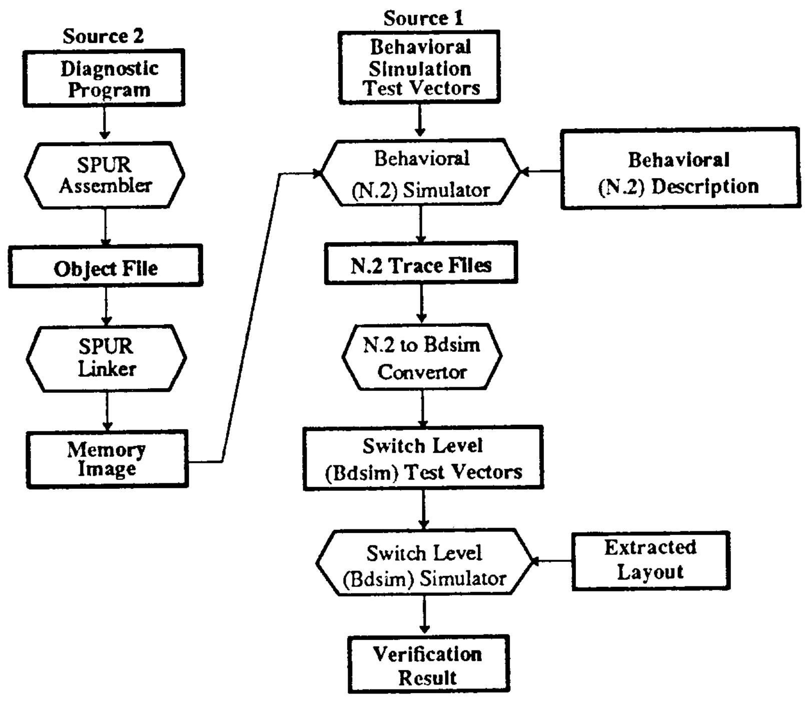


Figure 3-2-4 The Verification Process

The behavioral description on the right can be the description of a primitive module, or a composite module, or even the description of the CPU. The behavioral description is tested by the behavioral simulator. Behavioral simulation guidance can be provided in two different ways. Source 1: Test vectors are used to test the primitive and composite modules' functionality. Source 2: Diagnostic programs that are assembled into a memory image are used to test the description of the complete CPU.

3.2.4. Important Observations

Three important observations can be derived from the SPUR CPU design experience. These three observations will become important considerations as I try to develop a more analytical approach to microarchitectural design in Chapter 5 and predict the future trends in Chapter 6. The three observations are:

(1) A CAD tool that can transform the behavioral model directly to the layout–a silicon compiler–will be extremely useful, but still will not solve all the problems. As illustrated in the SPUR design process (Figure 3-2-1), such a CAD tool will only simplify the

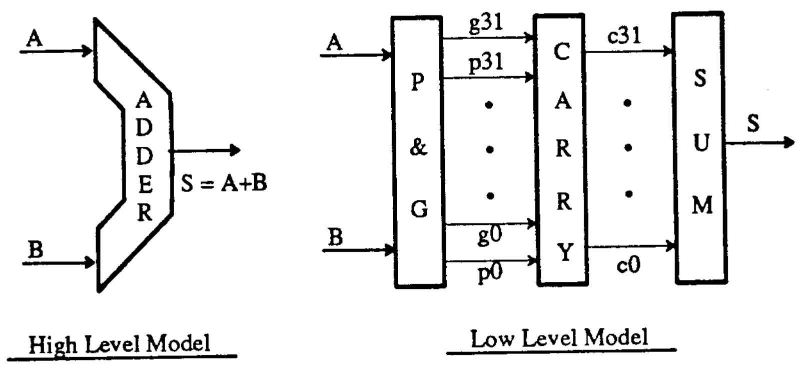


Figure 3-2-5 Conflicting Requirements of the Behavioral Model

The behavioral model can be a good tool to evaluate the effectiveness of alternative microarchitecture if it can be build rapidly. This require the description to be more abstract – higher level. On the other hand, if one want to be able to transform the behavioral description into layout easily or even automatically, it has to be less abstract – low level. Furthermore, if one wants to use the behavioral verification results to drive the switch level simulation effectively, the one-to-one mapping between the Behavioral description and the layout must be carry on to a relatively low level.

implementation step. The microarchitectural design step is still a major task by itself. You may argue that if the same microarchitecture is to be implemented in different technology as different products, then the microarchitectural development cost can be divided among different products. Practical experience showed that to get the highest performance from technology, however, the microarchitecture has to be customized to a technology [Pat89]. Finally, if the microarchitectural design is poor, it will be impossible for any silicon compiler to generate a good implementation from it.

(2) Verification is time consuming–it requires a lot of human interaction time because: (a) the designer has to create all the test cases either directly in the form of test vectors or indirectly via diagnostic programs, and (b) the designer has to interpret the verification result. To make matter worse, verification is usually done repeatedly at different levels. For example, in the SPUR CPU design process (Figure 3-2-1), verification is done after

each major steps by Instruction Level Simulation, Behavioral Level Simulation, and Switch Level Simulation. In order to speed up the design process, the human interaction time needed for verification must be reduced. The options are: (a) reduce the time it takes to generate the test cases, or (b) reduce the redundant verifications among different levels.

(3) There are two conflicting requirements for the description that models the microarchitecture (Figure 3-2-5). The SPUR CPU behavioral model [Kon89] is more towards the low level for two reasons. First of all, we want to use the verification results of the behavioral model to drive our switch level simulation. Secondly, we started writing the behavioral model late–we had already committed on most of the microarchitectural features. Consequently, instead of being a microarchitecture test bed, the behavioral model was used mainly as a specification for logic and circuit designers. Ideally, we would like to model as many alternative microarchitectures as possible such that we can evaluate the effectiveness of each alternative quantitatively. Due to the time we spent in modeling the SPUR CPU at low level, we could not afford major alteration in the SPUR CPU microarchitecture by the time we completed the first SPUR CPU model. In the future, I think VLSI designers should work on a high-level behavioral model earlier as a test bed for microarchitectural ideas. Once the high-level description is completed–hopefully with extensive CAD tools support–the designer can transform it into a low-level description for layout generation and switch level simulation.

3.3. The SPUR CPU Problems

All known SPUR CPU problems and their solutions are listed in Appendix B. This section discusses the more "educational" problems–problems that taught us some valuable lessons. The CPU problems can be classified into three groups:

(1) Microarchitectural Problems

The CPU chip is doing exactly what the microarchitect designed it to do although it is not doing what the microarchitect wanted it to do. The microarchitect has designed it wrong! These problems can be simulated in behavioral and switch level simulation. They were not detected during simulation because we did not cover all possible cases or we did not realize they were problems.

(2) Electrical Problems

The CPU chip is not doing what the microarchitect nor the logic designer designed it to do due to unexpected electrical problems. These problems cannot be simulated in behavioral nor switch level simulation. Careful and in-depth circuit simulation is the only way to detect these problems. These problems exist because the switch level simulation is not low level enough and it is not practical to run circuit simulation for the entire chip.

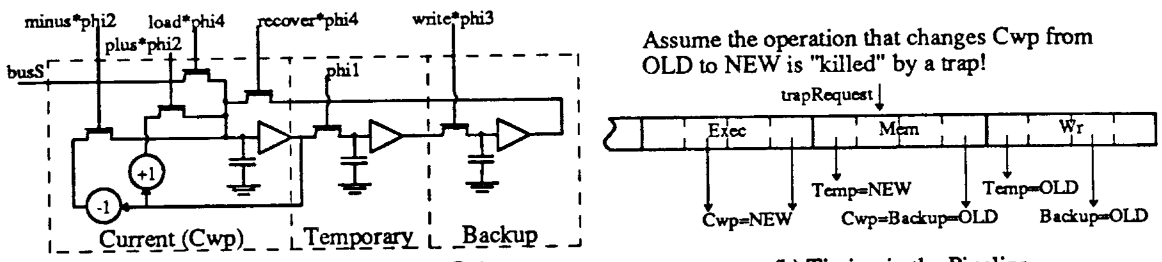
(3) Implementation Problems

The CPU chip is doing exactly what the logic or circuit designer designed it to do although it is not doing what the microarchitect want it to do. The logic or circuit designer implemented something differently than what the microarchitect had in mind! These problems may be detected by comparing the switch level simulation results against behavioral level simulation results if both the switch level and behavioral level descriptions have the proper level of detail. These problems exist because of miscommunication between the microarchitect and the logic or circuit designer.

3.3.1. Microarchitectural Problems

The most educational microarchitectural problem for SPUR is in the design of special registers. The SPUR CPU special registers that have potential problems are:

Cwp	Current register window pointer.
Swp	Save register window pointer.
Kpsw	Kernel processor status word.



(a) Structure of Current Window Pointer

(b) Timing in the Pipeline

Figure 3-3-1 Structure and Timing of the Special Registers

The structure of Cwp is shown in (a). All special registers are similar in that they all consist of two parts: Current and Backup. Any instruction that modifies the special register changes the Current part during either ϕ4 of its Exec stage (for Cwp, it can also be changed during ϕ2 instead of ϕ4) and updates the the Backup during ϕ3 of its Wr stage. (b) shows how the SPRU CPU can recover the special register to its old value during ϕ4 of the Mem stage if the instruction is killed by a trap.

Upsw	User processor status word.
Ins	Insert byte count register.

These special registers are described in details in Appendix A. Figure 3-3-1 shows the structure and the timing of the special register Cwp. Cwp is the most complex special register because it can be loaded from four different sources (see Figure 3-3-1(a)):

(1) Load from busS during ϕ4 for WR_SPECIAL instruction.

(2) Load from its Backup Copy during ϕ4 when there is a trap.

(3) Load from its plus-one copy during ϕ2 for CALL instruction.

(4) Load from its minus-one copy during ϕ2 for RETURN instruction.

All other special registers have similar structure and timing as the Cwp but they can only be loaded during ϕ4 from two sources: busS or the Backup copy. I have simplified Figure 3-3-1(a) by showing all storage nodes as dynamic latches–a simple pass transistor follow by a buffer. In the SPRU CPU, the Current and Backup have to be pseudo-static registers. Furthermore all pass gates in the SPUR CPU are composite pass gates formed by connecting NMOS and PMOS transistors in parallel.

The philosophy behind the special register design is that the Current part is changed as soon as possible such that the next instruction can use the new value. The Backup part is needed to recover the old value if the instruction that changes the special register is "killed" by a trap. This is illustrated in Figure 3-3-1(b) which shows that even if the instruction is "killed" in the last possible time (during $\phi2$ of its Mem stage–see Figure 2-3-4), the SPRU CPU can still recover the special register to its old value during $\phi4$ of the Mem stage.

The first mistake I made in designing the special register can be traced to Figure 3-3-1(b). In the regular register case, if any instruction that modifies regular registers is killed by a trap, its Wr stage is disabled and its destination register is not modified. In the special register case, if any instruction that modifies any special registers is killed by trap, Figure 3-3-1(b) seems to indicate

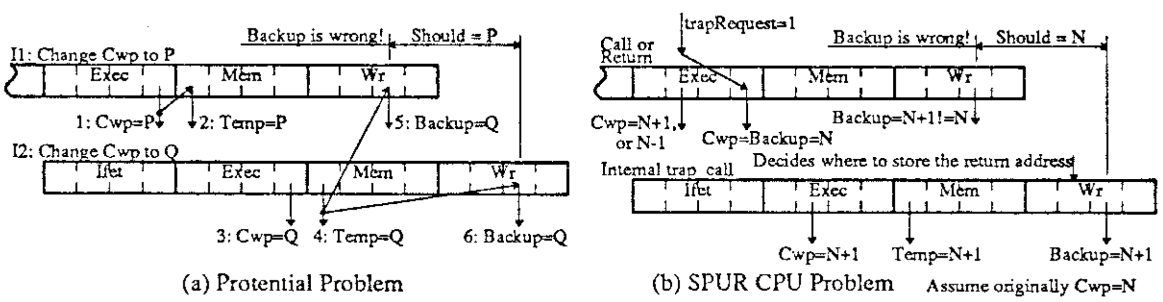


Figure 3-3-2 Problems with the Special Registers

(a) shows the potential problem with the special registers when two consecutive instructions try to modify the same special registers. Since the second instruction (I2) changes the Temp during $\phi1$ of its Mem stage (Step 4) before the first instruction (I1) updates the Backup (Step 5), Backup will get the latest value from Temp one cycle too early (in Step 5 instead of Step 6). (b) shows how this potential problem turn into real problem in the SPUR CPU when a Call or Return instruction is killed by a trap during its Exec stage. Since the internal instruction Trap_call uses the Backup copy of the Cwp to decide where to save the return address during $\phi2$ of its write stage, the return address is saved in the new window (N+1) instead of the desired old window (N).

that writing the Backup copy does not change anything–the Backup remains equal to OLD. Summarizing this error:

Mistake 1

Instead of treating special registers the same way as I treated regular registers, I did not disable the Wr stage (did not set *write*$\times\phi 3$ in Figure 3-3-1(a) to 0) of instructions that modify special registers even if it is killed by a trap. I had too much confidence in Figure 3-3-1(b).

The structure shown in Figure 3-3-1(a) has another potential problem. It will not allow two consecutive instructions to modify the same special register. This is illustrated in Figure 3-3-2(a) where the first instruction wants to set the Cwp to P while the second instruction wants to set the Cwp to Q. Due to the timing and the limitation of the structure, the Backup copy has the wrong value (Q) instead of the correct value (P) during the time period $T_{critical}$:

$$\phi 3 \text{ of I1's Wr stage} < T_{critical} < \phi 3 \text{ of I2's Wr stage}$$

If the second instruction (I2) need to use the Backup during $T_{critical}$, it will get the wrong value. I discovered this problem very early in the design process. I also noticed that this problem can be fixed easily by adding one more temporary latch between the Current and its Backup. Unfortunately, this was not done because I made my second and third mistakes:

Mistake 2

I thought the only time the second instruction used the Backup is when it is trapped in its Mem stage as shown in Figure 3-3-1(b).

Mistake 3

I thought nobody in his right mind will try to change the same register in two consecutive instructions as in Figure 3-3-2(a) because the first instruction can be replaced by an NOOP.

These two mistakes lead me to my fourth mistake:

Mistake 4

Instead of fixing the problem in hardware, I established a software restriction forbidding instruction sequence that has consecutive instructions modify the same special register.

Needless to say, I was very surprised when the potential problem shown in Figure 3-3-2(a) turn into a real problem in the SPUR CPU shown in Figure 3-3-2(b). The two surprises are:

Surprise 1

The discovery of Mistake 3. Two consecutive instruction modifying the same register can happen implicitly when a Call or Return is killed by a trap during its Exec stage. This is shown in Figure 3-3-2(b). The internal instruction Trap_call, which are placed in the pipeline by the *trapRequest* signal (see Figure 2-3-4), modifies the special register Cwp the same way as the regular Call.

Surprise 2

The discovery of Mistake 2. The second instruction, in the case of Figure 3-3-2(b), the internal instruction Trap_call will use the Backup during $T_{critical}$ even it is not trapped. Similar to the regular Call, the Trap_call use the Backup copy to decide which register window to save the return address.

Despite these two surprises, the potential problem shown in Figure 3-3-2(a) still would not have turned into a real problem in the SPUR CPU shown in Figure 3-3-2(b) if I had not made Mistake 1. If I did not make Mistake 1, the Wr stage of Call or Return in Figure 3-3-2(b) would have been disabled by the trapRequest and the Backup would not have been clobbered. Tom Wolfe, in his book "The Right Stuff", observed that many military pilots believed a pilot was never killed by a single mistake. Well, in this case, I sure made enough mistakes to get the CPU into serious trouble!

Fortunately, the case shown in Figure 3-3-2(a) is a very unusual case and it will never happen when the on-chip instruction cache is disabled. Unfortunately, it is so unusual that it was never tested in behavioral simulation and we did not detect this error until we have brought up the operating system and decided to turn on the on-chip instruction cache to increase the speed. As a matter of fact, this is the only reason why we have problem turning on the on-chip instruction cache. The rumor about the on-chip instruction cache's problem has been greatly exaggerated!

Notice that the problem shown in Figure 3-3-2(b) is only a transient error because the Backup Cwp simply has the "correct" value at the wrong time (one cycle too early). Once the Trap_call finishes its execution, it becomes perfectly legal for the Backup to have this latest value. However, this is bad enough to cause the Trap_call to save the return address in R26 of the new register window instead of R26 of the old register window. This problem does have a simple software solution. If by software convention, all procedures and trap handlers must set its R26 to zero before returning to its return address, then the SPUR CPU can check R26 of the new window whenever it takes a trap. If it is not zero, the case shown in Figure 3-3-2(b) must have occurred. The software can then find out what the return address is by reading and saving this register.

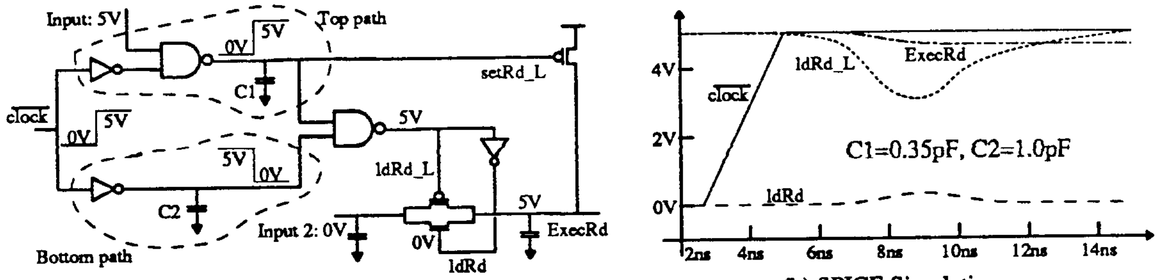
One big lesson we learned here is: Keep it regular! Whenever you make an exception (Mistake 1 and 4), there is likely to be some unexpected cases to get you in the most unexpected way. Another lesson is that there may be many cases you may never thought of during simulation (Mistake 2 and 3) and one must find some easy way to cover more cases in simulation. This is discussed further in Section 3.4.1.

3.3.2. Electrical Problems

The microarchitect is not the only person in SPUR that makes multiple mistakes. The circuit designer also made multiple mistakes at the electrical level that resulted in an electrical problem in the SPUR CPU. This is discussed in Section 3.3.2.1. After discussing all these problems caused by multiple mistakes, Section 3.3.2.2 shows how one single mistake can ruin your whole day!

3.3.2.1. A Hazardous Circuit

The circuit shown in Figure 3-3-3(a) is hazardous because the clock signal $\overline{clock}$ is gated with other inputs in a way that it is forced to pass through two different paths before it is merged



(a) Hazardous Circuit (b) SPICE Simulation

Figure 3-3-3 A Hazardous Circuit

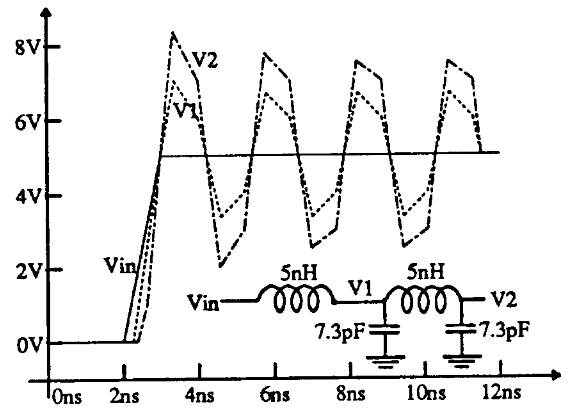
The case we are interested is when $Input = 5V$ and $\overline{clock}$ goes from 0V to 5V. When $\overline{clock}$ equals to 0V, Node ExecRd is charged to 5V. When *clock* switches from 0V to 5V, we want ExecRd to stay at 5V. The only hazard that may discharge ExecRd is that there may be glitches on the ldRd_L and ldRd signals. Our SPICE circuit simulation showed this hazard can occur only if $C_2 > 3 \times C_1$. However, as shown in (b), even if there are glitches on ldRd_L and ldRd, ExecRd is still not destroyed.

again into another signal (ldRd_L). This hazard, however, cannot be detected by a switch level simulator that does not have a good capacitance model. When the effect of the parasitic capacitors C_1 and C_2 are ignored, the bottom path in Figure 3-3-3(a) has one less gate delay and will go 0 before the top path goes to 5V. There will not be any glitch on the control lines ldRd_L and ldRd.

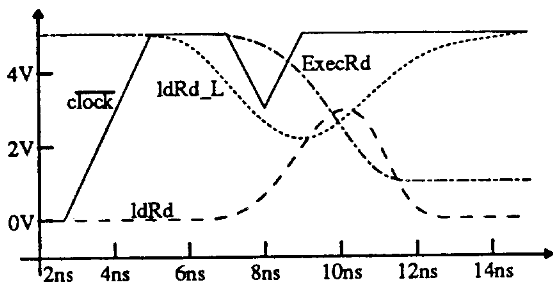
Mistake 1

We believed our switch level simulation and kept a hazardous circuit in our design that combines clock signal with other signals at a place other than at the control point.

This hazard can be detected by careful circuit level simulation using SPICE. If $C_2 > 3 \times C_1$, the top path in Figure 3-3-3(a) will go to 5V before the bottom path go to 0V. There will be glitches on the control lines ldRd_L and ldRd. However, if this is the only problem, Figure 3-3-3(b) shows that ExecRd still will not be discharged unintentionally because the glitches on the control lines ldRd_L and ldRd are not big enough. But then again, as the great philosopher Mur-



(a) Clock Line as a LC Network



(b) SPICE Simulation with Clock Glitch

Figure 3-3-4 Problems of the Hazardous Circuit

The SPUR CPU clock line is modeled as an LC network in (a). This simulation indicates that there will be some ringing in the clock signal *clock*. (b) shows how the glitches on ldRd_L and ldRd, which were insignificant in Figure 3-3-3(b), are now amplified by the glitches in the clock signal. These larger glitches are big enough to discharge ExecRd accidently.

phy had predicted, things usually get worse before getting any better. We made our second mistake:

Mistake 2

Instead of placing the clock generator and clock line drivers in the middle of the chip, they were placed on the left hand side. This resulted in long and unbuffered clock wires.

In the first version of the CPU, the clock line is approximately 8mm long. We estimated it to have one Ohm of resistance, 10nH of inductance, and 14.6pF of capacitance. Although the resistance is relatively small, the inductance and capacitance are big enough to cause some ringing in clock line (Figure 3-3-4(a)). While the ringing in the real clock line will die down due to resistance, Figure 3-3-4(b) shows that the initial ringing on the clock line $\overline{clock}$ are enough to amplify the glitches on the control lines ldRd_L and ldRd such that ExecRd will be discharged unintentionally.

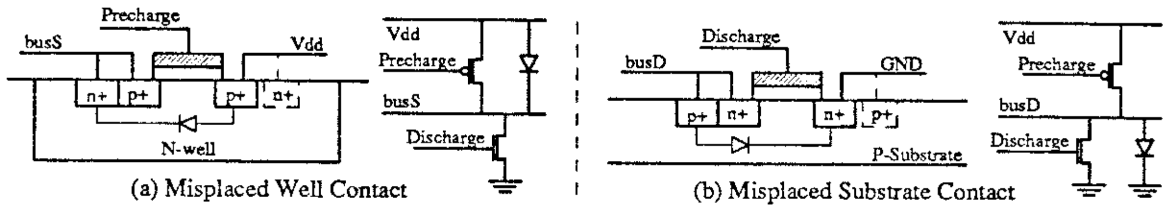


Figure 3-3-5 Misplaced Well and Substrate Contacts

Instead of placing the well and substrate contacts on the right side of the transistor and connects to the power supply V_{dd} and GND respectively (the dotted line), they are placed incorrectly on the left side and connected to the busses. These misplaced contacts form diodes between the V_{dd} and GND that will prevent busS in (a) to go below 4.3V and busD in (b) to go above 0.7V.

This problem caused the Call instruction unable to save the return address. It was first solved in software by emulating the Call instruction. The hardware is fixed in the second version of the CPU chip where the hazardous circuit is redesigned and the clock generator is moved to the center of the chip to reduce the length of the clock wire. The important lessons here is that one should never trust the CAD tools blindly and use any marginal design just because the CAD tools predict it will work. There are just too many second order effects you and the CAD tools may have neglected.

Careful design is still necessary! Despite all rumors, there is still no good substitution for a good electrical engineer knowing what he is doing and working very carefully.

3.3.2.2. Well Problems

The last two problems discussed in Section 3.3.1 and 3.3.2.1 are both caused by multiple mistakes. In this section, I want to show how one single mistake can cause serious problem. One of the supposingly good features of the Magic Layout System [SMH85] we used is that the CMOS layout artists do not have to worry about well placement–Magic will generate the wells automatically. Unfortunately, this simplifying assumption also means the Magic Layout System does not extract the well from the layout when it generates the switch level and circuit level description. Thus any layout verification tools based on the Magic Layout System cannot check the well either. One of the first things most circuit designers will warn you about this approach is that you may end up with floating wells. However, one of the most painful lesson we learned in SPUR is that floating well is not the only possible problem in this approach.

One problem in the SPUR CPU is the misplaced well and subtract contacts shown in Figure 3-3-5. In the N-well process used by SPUR, a misplaced well contact will cause a node to stuck at one (Figure 3-3-5(a)) and a misplaced substrate contact will cause a node to stuck at zero (Figure 3-3-5(b)). In the SPUR CPU, we are fortunate that all misplaced well contacts are at redundant precharge transistors in the lower datapath. As illustrated in Figure 3-3-6, this enabled us to solve the stuck at 1 problem by cutting off the power supply to these precharge devices. The stuck at 0 problem, however, cannot be solved by laser cutting and we were forced to do most primary testing using a crippled 8-bit CPU until the problem was fixed in the second version of

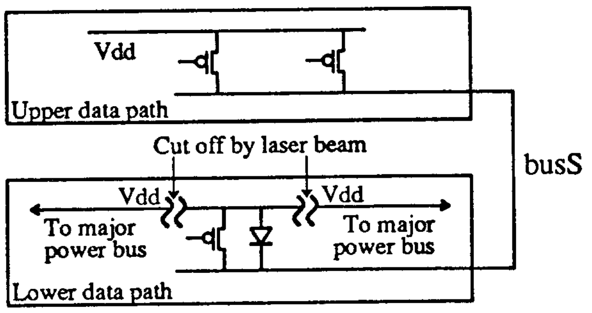


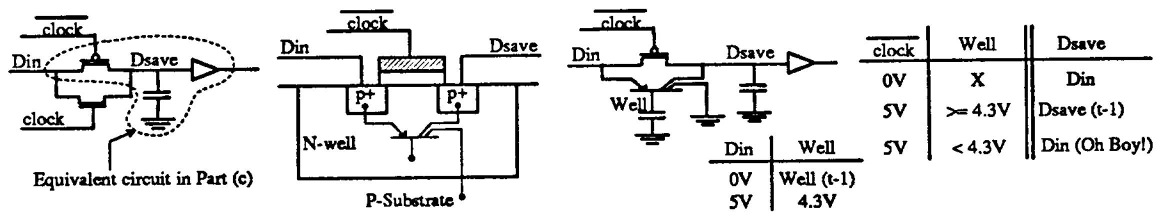
Figure 3-3-6 A Quick Hardware Fix for Misplaced Well Contacts

The misplaced well contacts are all at the redundant precharge PMOS transistors at the lower datapath. These precharge transistors are redundant because busS is also precharged by transistors at the upper datapath. Furthermore, the V_{dd} lines that supply power to these redundant transistors are connected to major power busses on either side. Therefore, by using the laser cutting system at Information Science Institute [Par87] to cut off the V_{dd} supply on both sides, we can isolate these precharging device without affecting the CPU function.

the SPUR CPU.

The floating well is not a problem in the SPUR CPU but it is a problem in the Cache Controller chip. The Cache Controller floating well problem is discussed here because it is quite different from what most people expected from floating well. The first problem come to most people's mind concerning floating wells is latch up. In the Cache Controller case, however, we learned that floating well can also be a problem if there is any dynamic storage node inside the floating well. This is illustrated in Figure 3-3-7. This is ironic because a common technique to build dynamic storage node is to use a pair of pass transistors and the well for one of these transistors is likely to be a floating well because it is usually hard to place a well contact in this congested area. As shown in Figure 3-3-7, if the PMOS pass transistor is in a floating well, the dynamic register will retain its value correctly only if the well is above 4.3V.

One thing we learned after we discovered all the well problems is that all these problems can be detected by Magic if we do some tricks to the Magic technology file. In order to detect these



Din	Well
0V	Well (t-1)
5V	4.3V

$\overline{\text{clock}}$	Well	Dsave
0V	X	Din
5V	>= 4.3V	Dsave (t-1)
5V	< 4.3V	Din (Oh Boy!)

(a) Dynamic Register (b) PNP Transistor (c) Equivalent Circuit

Figure 3-3-7 Floating Well Problem

(a) shows a dynamic storage device in which data is stored dynamically in the node D_{save}. (b) shows how a PNP transistor is formed if the PMOS transistor whose gate is connected to $\overline{clock}$ is in a floating well. (c) shows the equivalent circuit. When $\overline{clock}$ is asserted (0V), the dynamic register latches in the data D_{in} correctly. Unfortunately, when $\overline{clock}$ is disasserted (5V), this dynamic register holds the value correctly only if the Well node is above 4.3V. Since the Well node is charged to to 4.3V whenever D_{in} is 5V and will stay there until leakage current discharge it, this dynamic register will operate correctly most of the time.

errors in Magic, the designer must request Magic to display the wells explicitly. Therefore I concluded that as far as the Magic Layout System is concerned, well-independent design style can be dangerous, although it may seem attractive in theory.

3.3.3. Implementation Problems

The only implementation problem we have is that the backup copies of all special registers (Figure 3-3-1(a)) were implemented incorrectly in dynamic registers instead of static or pseudo-static registers. Since the current copy loads from its backup whenever a trap occurs, the backup copy must retain the correct value at all time. This fact was so obvious to me, the microarchitect, that I did not even bother to specify it explicitly in the documentation. The circuit designer on the other hand did not have the same understanding of the operation and thought a dynamic register was sufficient.

This problem is not detected during switch level simulation because the switch simulation do not simulate leakage current in the dynamic node. Furthermore, since all our test programs have short run time (relative to the real work load), the leakage current is not a problem either. This problem was not discovered until we started debugging the operating system. It was fixed in software by interrupting the CPU regularly to refresh (read and write back) the special registers. The lesson here is that the microarchitect should specify everything explicitly because what is obvious to him may not be obvious to the logic and circuit designers who are looking at the design at a much lower and local level.

3.4. The SPUR CPU Technical Lessons

The SPUR CPU design process and all the problems taught us some valuable lessons. The technical lessons are summarized in this section. The philosophical lessons are summarized Section 3.5.

3.4.1. Simulation and Testing Lessons

The SPUR CPU simulation process consists of two levels: behavioral level and switch level. They have already been discussed in Section 3.1.3 and Section 3.2.3, respectively. The SPUR CPU chip testing process shown in Figure 3-4-1 also consists of two levels: chip test and board test. The switch level simulation vectors were used for initial chip testing. For chip debugging, we found out we must be able to write a new test, run it, and verify the results rapidly. This is accomplished by automating the five-step chip testing process.

The behavioral level diagnostics for the SPUR processor board (see Processor Behavioral Model, Figure 3-1-3) were used for initial board test. In order to debug the processor board, we must also write new tests. Although the board test for SPUR is much more extensive than that for RISC II and SOAR, it is still not the ultimate test. One important lesson we learned is that the ultimate test came when we tried to bring up the operating system [OCD88]. This is the time when we discovered most of the problems. It is interesting to note that the operating system in its first 5ms of operation requires the SPUR CPU to execute more instructions than the total number

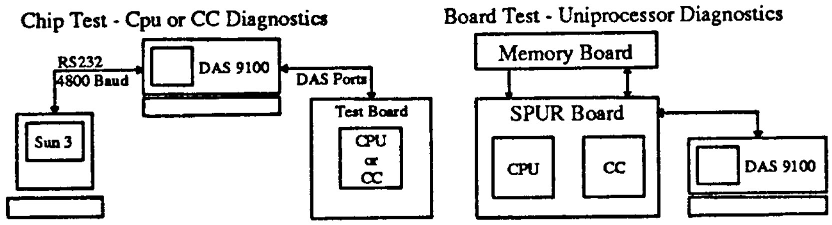


Figure 3-4-1 SPUR CPU Testing Strategy

The chip test is a five-step process: (1) Generate test vectors on the SUN work station by running behavioral diagnostics. (2) Down load the vectors onto the DAS. (3) DAS drives the test board and collects output vectors. (4) DAS sends output vectors back to the SUN. (5) Verify output vectors on the SUN. After the CPU and Cache Controller chips have been tested independently, they are tested together on the SPUR processor board. Uniprocessor diagnostics are loaded onto the memory board and the DAS is used for debugging.

of instructions the switch level simulator has simulated (Table 3-1-3). This is unavoidable because real machines must run much faster than the simulator. It is not a major problem if the diagnostics are well chosen. There were, however, a couple of important lessons we learned concerning simulation and testing.

3.4.1.1. Lesson 1: One Size Does Not Fit All

The switch level simulator is twenty times slower than the behavioral simulator (60 sec/cycle versus 3 sec/cycle). Therefore only a subset of the behavioral diagnostics are used in switch level simulation. Initially, we envisioned that the following process could be fully automated:

(1) Run the diagnostics in the behavioral simulator and trace the input/output ports of the CPU.

(2) Convert the traces into switch level test vectors for switch level simulation.

(3) Convert switch level test vectors to logic analyzer vectors for chip testing.

We encountered two problems in automating this process. First, the behavioral simulator, switch level simulator, and the logic analyzer all have different input/output formats. More importantly, each initializes the chip differently and propagates "don't care" conditions differently. Consequently, we must edit some automatically generated test vectors and examine whether reported errors are real errors. Both tasks are time consuming and error prone. The designers of different simulators and the logic analyzer must work together to avoid this problem. Second, behavioral diagnostics and switch level diagnostics have different requirements. Behavioral diagnostics are verification diagnostics and you want them to be long and general. Switch level and chip testing diagnostics, on the other hand, are debugging diagnostics–you want them to be short and specific. We solved this problem by building long verification diagnostics from short self-testing debugging diagnostics.

3.4.1.2. Lesson 2: The Danger of Simulation

In our simulation world, diagnostics are executed one at a time: Start a diagnostic, finish it, then start the next diagnostic. This is not realistic because in the real world, programs seldom run from start to finish without being interrupted. As a matter of fact, the special registers problem discussed in Section 3.3.1 is one case where the SPUR CPU cannot recover from an interrupt. In order to make behavioral simulation more realistic, we must ensure that each diagnostic can run successfully even if its execution is interrupted randomly. An interesting approach is shown in Figure 3-4-2 where the execution of one diagnostic is interrupted constantly by another diagnostic.

This approach has several advantages. One major reason for multiplicative complexity is the random interaction of different architectural features. This random interaction is caused by

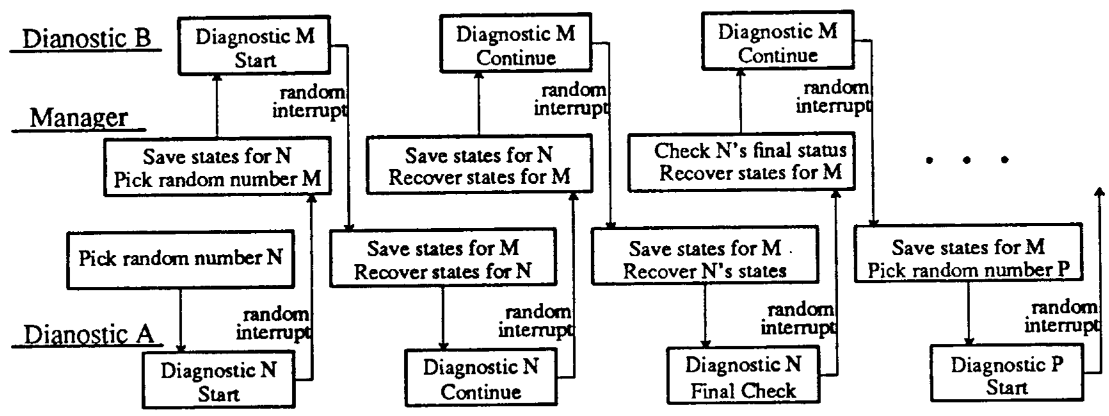

Dianostic B
Manager
Dianostic A
Diagnostic M Start
random interrupt
Save states for N Pick random number M
Pick random number N
Diagnostic N Start
Diagnostic M Continue
Save states for N Recover states for M
Save states for M Recover states for N
Diagnostic N Continue
Diagnostic M Continue
Check N's final status Recover states for M
Save states for M Recover N's states
Diagnostic N Final Check
Save states for M Pick random number P
Diagnostic P Start
• • •

Figure 3-4-2 Random Simulation Algorithm

This example limits the number of active diagnostics to two: A and B. The software manager begins the simulation by randomly starting a diagnostic, say diagnostic N. After a random period of time, the manager interrupts diagnostic N's execution by starting another randomly selected diagnostic, say diagnostic M. The manager then switches between the two diagnostics until one of the diagnostic is completed. The manager then randomly selects another diagnostic: diagnostic P. Each diagnostic must be self checking and the manager should terminate the simulation as soon as any error is detected.

random events such as traps and interrupts and can create a large number of CPU states. It is very time consuming (it may not even be possible) for the designer to visualize all the possible states and write diagnostics to cover them. However, by letting diagnostics interrupt each other randomly, we can explore a large number of CPU states by using only a relatively small set of diagnostics. Furthermore, due to random interaction, each time a new diagnostic is added to the set, the increase in CPU states that can be tested goes beyond the checks in the new diagnostics.

3.4.2. The Nature of Microarchitectural Design

The simulation and testing lessons we learned are only part of the story. The root of the problem is a gap in the computer engineering education. The term microarchitecture, as defined in Chapter 1, is the specification of how the macroarchitecture is implemented in a given technology. Microarchitectural design has been treated more like an art than science. This is unfortunate because you can teach science but you cannot teach art! Consequently, the art of microarchitectural design is not well taught and, as shown in Figure 3-4-3, there is a gap in the computer

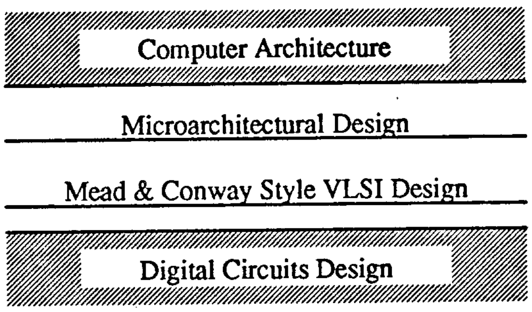


Figure 3-4-3 The Gap in Computer Engineering Education

At the highest level, Computer Science classes are available for computer architecture. At the lowest level, Electrical Engineering classes are available for digital circuit design. There is a big gap between these two levels. In my opinion Mead & Conway style VLSI design class only bridges the gap between these two levels because it is a only a digital circuit design class in in Computer Science perspective.

engineering education. This gap is not as apparent in the past in the academic world because most universities' VLSI projects are either driven from the top–implement architectural innovations, or driven from the bottom–try out fast circuit technology. The only way to close this gap is to make the microarchitectural design process more a science than art by developing a more systematic approach to microarchitectural design.

In my opinion, the key of making the microarchitectural design process into a science is to put more emphasis on the tradeoffs between performance, resources, and complexity. As will be discussed in Section 4.1, one way to measure performance is the $T \times I \times C$ product where T is cycle time, I is the number of instructions it takes to execute certain benchmark programs, and C is the average number of cycle per instruction. Chip area and transistors count are two examples

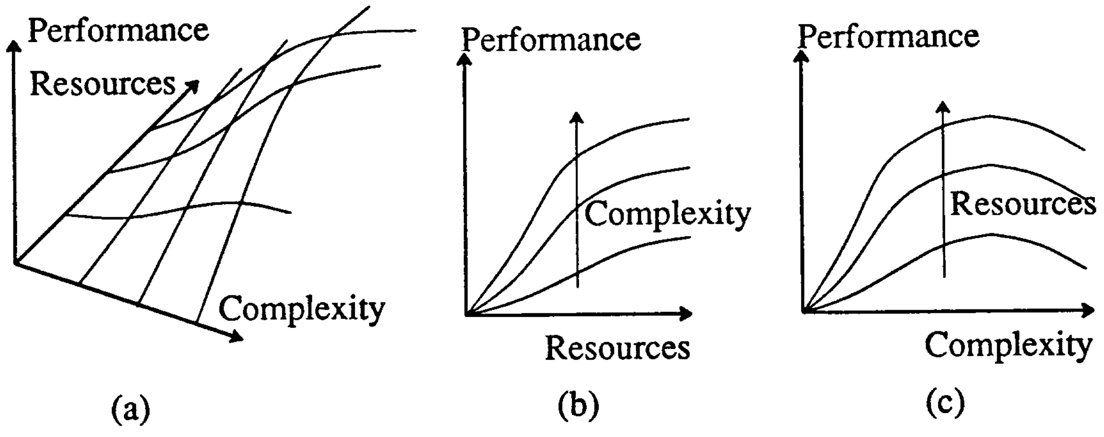


Figure 3-4-4 Performance as a Function of Resources and Complexity

(a) is a three dimensional plot of performance as a function of resources and complexity. (b) and (c) are the two-dimensional projections of this design surface onto the resources and complexity axes respectively. (b) shows that for a fixed amount of complexity, increase the amount of resources will increase the performance. Similarly, as shown in (c), for a fixed amount of resources, increase the amount of complexity will increase the performance. In either case, the rule of diminishing return applies. The RISC argument carries one step further than the rule of diminishing return. RISC proponents suggest that as the complexity gets too high, the performance actually goes down.

of resources metrics. The complexity of a design can be considered qualitatively as a measure of how hard it is to specify and implement that design. The number of cycles of diagnostics and the simulation effort are examples of quantitative complexity metrics.

Performance, resources, and complexity can be considered as three independent dimensions in a multidimensional design space. With other variables such as technology and designer's ability in this multi-dimensional design space being constant, alternative microarchitectures are restricted to points on a three-dimensional design surface shown in Figure 3-4-4(a). Without a high-level design automation system, a designer must go through the process of pruning this design space by making trade-offs. The most systematical way to make these trade-offs is to perform experiments that gives quantitative estimates in the performance, resources, and complexity dimensions. A designer would like to get these estimates with minimal effort and as early as possible in the design process such that more alternative microarchitectures can be evaluated.

Figure 3-4-4(a) is a simplified view of the design space because, in reality, resources and complexity are not completely independent. However, they are not as dependent as most people think either. If resources are measured in terms of area and complexity is defined as the degree of difficulty in understanding the operation of a module, then resource and complexity can be quite independent. For example, a module may be small but its operation can still be very difficult to understand. In Chapter 4, I will evaluate the different SPUR CPU features in terms of the performance, resources, and complexity tradeoffs. Before I move on to the next chapter, I like to list the philosophical lessons I learned.

3.5. The SPUR CPU Philosophical Lessons

In this section, we will summarize some of the philosophical lessons we learned in designing the SPUR CPU. Our experience has shown that these apparently trivial lessons may easily be forgotten.

Keep it Simple. The simplest solution that works is also the most elegant solution because: (1) unless you are willing and able to use the highest performance solutions for all components, the overall performance gain from the improvement of a single component is limited, (2) simple solutions require less design and implementation time and thus can make use of newer technology that may negate many performance advantages of the complex solution, and (3) the simplest solution requires the least human designer time which in a sense is the most limited and expensive resource. Consequently, as long as the simplest solution meets the performance goal and is within the resources available range, the designer should accept the solution and move onto other problems waiting for him to solve. For example, the SPUR CPU uses a simple 4-phase clocking scheme that places a lower limit on the CPU cycle time (approximately 100ns). This is acceptable because the external bus and the memory system cannot run any faster than 100ns.

A Working Whole is Better than a Working Part. A designer should spend his time solving unsolved problems instead of trying to find a better solution for an already solved problem. Professor John Ousterhout at Berkeley once said: "The biggest performance enhancement is achieved when going from a non-working system to a working system." This may sound trivial but whenever a designer is not making any progress in solving a new problem it can be very tempting for him to go back to something he already understands and try to optimize it. For example, we did not attempt to reduce the size of the CPU's master control PLA any further because it can already fit nicely into its assigned space.

The A in CAD Means Aided. The CAD tools are there to help the designer, not to replace him. The result can be catastrophic if the designer does not think nor work carefully and expects the CAD tools to do all his work and catch all his foolish mistakes. For example, switch level simulators or even electrical rules checkers cannot detect many electrical problems such as coupling, charge sharing, and race conditions. They can only be avoided by careful design. Furthermore a VLSI designer should realize that building his own simple tools is the best way to define his problems for CAD tool designers. No matter how simple the tool he build is, it is probably still the best way to define the problem. Once the problem is better defined, it can be explained to CAD tools designers who can then develop tools that are more general, have more features, and more efficient for the problem.

The Rubik's Cube Analogy. One of the most interesting features of the Rubik's Cube puzzle is that each step in solving the puzzle usually has the horrendous effect of destroying some results of previous steps. Similarly the designer must be willing to throw away some of his work that does not perform in order to finish the design project. More importantly, if the designer is unwilling to throw away any of his work, he probably will be unwilling to start until he has all the answers. Unfortunately, in most if not all cases, one will never get all the answers unless one starts. For example, in the beginning of the SPUR project, we did some layout to estimate the relative sizes of various modules. None of this layout was used in the final CPU.

Keep it Regular. The designer must always try to follow the same regular pattern. Our experience in SPUR is that whenever we make an exception to save area, power, or just being lazy, we usually regret it later. For example, in SPUR, everything in the behavioral level is modeled in N.2 [EEE85] except the FPU, which is modeled in SLANG [Van82]. Although the reasons for using SLANG have long been forgotten, none of us forget the grief it caused when we tried to simulate the FPU with the rest of the system.

3.6. REFERENCES

[EEE85] *N.2 Simulator User's Manual*, ENDOT, Inc.,, Cleveland, OHIO, 1985.

[EgK88] S. Eggers and R. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *The 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 30-June 2, 1988.

[Gib87] G. Gibson, "Estimating Performance of Single Bus, Shared Memory Multiprocessors", Report No. UCB/Computer Science Dpt. 87/355, Computer Science Division, EECS Department, University of California, Berkeley, May 1987.

[HaK86] P. Hansen and S. Kong, "SPUR Coprocessor Interface Description", Report No. UCB/Computer Science Dpt. 87/308, Computer Science Division, EECS Department, University of California, Berkeley, October 1986.

[Kat83] G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, Doctoral Dissertation, Computer Science Division, EECS Department, University of California, Berkeley, October 1983.

[KEW85] R. Katz, S. Eggers, D. Wood, C. Perkins and R. Sheldon, "Implementing A Cache Consistency Protocol", *The 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, June 17-19, 1985.

[Kat85] R. Katz, et al., "Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses", Report No. UCB/Computer Science Dpt. 85/221, Computer Science Division, EECS Department, University of California, Berkeley, January 1985.

[KWG87] S. Kong, D. Wood, G. Gibson, R. Katz and D. Patterson, "Design Methodology for a VLSI Multiprocessor Workstation", *VLSI Systems Design VIII*, 2 (February 1987).

[Kon89] S. Kong, "The SPUR CPU Behavioral Model", Report No. UCB/Computer Science Dpt. 89/508, Computer Science Division, EECS Department University of California, Berkeley, May 1989.

[Lee86] D. Lee, *Datapath Design Considerations for a High Performance VLSI Multiprocessor* , Master Report, EECS Department, University of California, Berkeley, CA 94720, November, 1986.

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, "The Sprite Network Operating System", *Computer 21*, 2 (February 1988).

[Par87] B. Parker, Private Communication Information Science Institute, December, 1987.

[Pat82] D. A. Patterson, "A RISCy Approach to Computer Design", *COMPCON 1982*, February, 1982.

[Pat89] D. Patterson, Private Communication Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, January, 1989.

[SMH85] W. Scott, R. Mayo, G. Hamachi and J. Ousterhout, editors. "1986 VLSI Tools: Still More Works by the Original Artists", Report No. UCB/Computer Science Dpt. 86/272, Computer Science Division, EECS Department University of California, Berkeley, CA 94720, December 1985.

[Tay85] G. S. Taylor, "SPUR Instruction Set Architecture", in *Proceedings of CS292i: Implementation of VLSI Systems*, R. Katz (editor), Computer Science Division, EECS Department, University of California, Berkeley, September 1985.

[Tay86] G. Taylor et al., *Evaluation of the SPUR Lisp Architecture*, The 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 2-5, 1986.

[Ung84a] D. Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", *ACM Software Engineering Notes/SIGPLAN Notices*

Notices Software Engineering Symposium on Practical Software Development Environments, Pittsburg, April, 1984.

[Ung84b] D. Ungar et al., ''Architecture of SOAR: Smalltalk on a RISC'', *The 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, June 5-7, 1984.

[Van82] K. VanDyke, *Slang: A Logic Simulation Language*, Master Report, Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, June, 1982.

[WEG87] D. Wood, S. Eggers and G. Gibson, ''SPUR Memory System Architecture'', Report No. UCB/Computer Science Dpt. 87/394, Computer Science Division, EECS Department, University of California, Berkeley, December 1987.

[ZHH88] B. Zorn, P. Hillfinger, K. Ho, J. Larus and L. Semenzato, ''Features for Multiprocessing in SPUR LISP'', Report No. UCB/Computer Science Dpt. 88/406, Computer Science Division, EECS Department, University of California, Berkeley, March 1988.

Chapter 4

MICROARCHITECTURAL EVALUATION

Few things are harder to put up with than the annoyance of a good example.

Mark Twain

This chapter evaluates various features of the SPUR CPU in terms of their impact on performance, resources, and complexity. Resources and complexity are quantified by sets of metrics. Since each feature has different impact on resources and complexity, the metrics used to quantify the impact may be different. Therefore, I will talk about the metrics for each feature separately when I discuss each feature. On the other hand, in order to study the performance impact quantitatively, I must develop a performance model. This is done in Section 4.1. This performance model can then be used to evaluate the performance improvement due to each feature by comparing the performance of the SPUR CPU against the performance of an imaginary stripped down SPUR CPU that does not have that feature. The SPUR CPU features to be evaluated in this chapter are: LISP support in Section 4.2, FPU support in Section 4.3, longer pipeline in Section 4.4, on-chip instruction cache Section 4.5, and multiprocessing support in Section 4.6. Insights base on this evaluation are summarized in Section 4.7.

4.1. The Performance Model

Performance of a microarchitecture can be measured independent of implementation considerations by measuring the number of instructions it takes to execute some benchmarks (I):

$$Performance = \frac{1}{I}$$

However, just as cache performance cannot be measured in terms of hit rate alone [Hil87a] [Hil88], ignoring implementation considerations can be misleading. In order to include implementation considerations, microarchitectural performance can be measured in terms of the TIC product [Hen85]:

$$Performance = \frac{1}{T \times I \times C} \tag{4.1.1}$$

where

T = Cycle time

I = Number of instructions it takes to execute a benchmark

C = Average number of cycles per instruction

Clearly $T \times I \times C \neq I$. Therefore, performance is not simply a function of instruction count. During the design process, the microarchitect must make decisions constantly concerning whether to include certain features in the microarchitecture. In order to make these decisions quantitatively, the microarchitect wants to be able to use the performance model to predict the performance improvement due to each feature under consideration. This can be accomplished by comparing the performance of a base microarchitecture without the feature under consideration against the performance of the enhanced microarchitecture with the feature.

In the following discussion, I will refer to the feature under consideration as $feature_i$. I will also use subscript "o" for the base microarchitecture without $feature_i$ (T_o, I_o, and C_o) and subscript "i" for the enhanced microarchitecture with $feature_i$ (T_i, I_i, and C_i). Using this notation, the performance gain and percentage performance improvement due to $feature_i$ can be defined as:

$$GAIN_i = \text{Performance gain due to } feature_i = \frac{T_o \times I_o \times C_o}{T_i \times I_i \times C_i} \tag{4.1.2}$$

$$IMP_i = \text{Performance improvement (\%) due to } feature_i = \left[GAIN_i - 1 \right] \times 100\% \tag{4.1.3}$$

The T and C terms in the above equations take implementation considerations into account. Therefore, the microarchitect must consider not just how the new feature ($feature_i$) will affect I

(change from I_o to I_i) but also how it will affect T (change from T_o to T_i) and C (change from C_o to C_i). The microarchitect must estimate the effect on T and C based on his experience--or by a systematic approach discussed in Chapter 5. The effect on I can be obtained from the macroarchitect who runs benchmarks on the instruction level simulators for the enhanced (measure I_i) and base architecture (measure I_o). Since architectural features are added to perform a specific function more efficiently, another way to obtain the effect on I is to estimate it indirectly by using Equation 4.1.4, derived below. Before I can explain Equation 4.1.4, I must define the following terms:

I_o = Number of instructions it takes to execute a benchmark without feature i,

I_i = Number of instructions it takes to execute a benchmark with feature i,

U_i = Number of times feature i is used in the benchmark,

F_i = Frequency of feature i in the benchmark = $\frac{U_i}{I_i}$,

M_o = Number of instructions needed to perform the desired function without $feature_i$, and

M_i = Number of instructions needed to perform the desired function with $feature_i$.

Most architectural features are added to reduce the number of instructions to perform certain function. In such cases, $M_o > M_i$. The number of instructions it takes to execute a benchmark without $feature_i$ (I_o) can now be written in terms of I_i, M_i, M_o and F_i:

$$I_o = I_i - F_i \times I_i \times M_i + F_i \times I_i \times M_o = I_i \times \left[1 + F_i \times (M_o - M_i) \right] \qquad (4.1.4)$$

Substituting Equation 4.1.4 into 4.1.2 and 4.1.3, we have:

$$GAIN_i = \frac{T_o}{T_i} \times \frac{C_o}{C_i} \times \left[1 + F_i \times (M_o - M_i) \right] \qquad (4.1.5)$$

$$IMP_i = \left[\frac{T_o}{T_i} \times \frac{C_o}{C_i} \times \left[1 + F_i \times (M_o - M_i) \right] - 1 \right] \times 100\% \qquad (4.1.6)$$

Figure 4-1-1(a) is a plot of Equation 4.1.6 where $M_o = 3$ and $M_i = 1$. This is the case where adding the new feature enables one instruction to perform the same function that used to be performed by three instructions. The p factor in Figure 4-1-1 is defined as the product of the cycle time ratio $\left[\frac{T_o}{T_i} \right]$ and average number of cycles per instruction ratio $\left[\frac{C_o}{C_i} \right]$. Notice that as the p

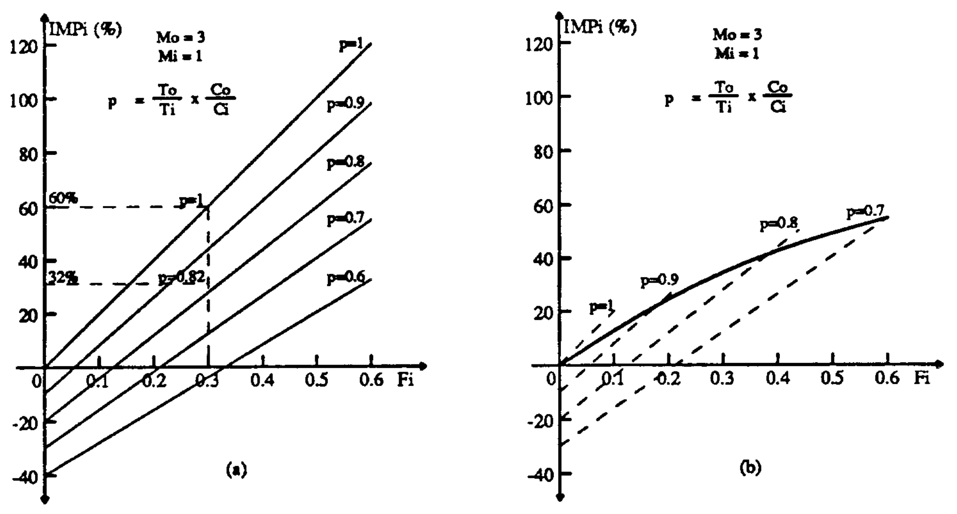


Figure 4-1-1 Performance Graphs

Each line in (a) shows the performance improvement as a function of F_i for $M_o = 3, M_i = 1$, and constant p factor. (b) shows the case where the p factor is a decreasing function of F_i and $p \approx 1$ when F_i is small. In this case the performance improvement will follow the solid curve that is close to $p = 1$ line when F_i is small. As F_i increases, the curve level off and follows the lines with smaller p factor.

factor gets smaller, the advantage of the new feature is reduced in two directions: (1) the line is shifted down and, (2) the slope of the line decreases. For example, if the new feature has no effect on the cycle time ($T_i = T_o$) nor on the average number of cycles per instruction ($C_i = T_o$), then $p = 1$ and there is a 60% performance gain if the new feature is used 30% of the time. However, if this feature increases the cycle time and the average number of cycles per instruction by 10% (T_i =1.1xT_o and C_i =1.1xC_o), then $p = 0.82$ and the performance gain is only 32%.

Figure 4-1-1(b) shows the performance improvement as a function of F_i when the p factor is a decreasing function of F_i. This is the case if the feature you added is a new instruction that takes longer than the original average number of cycles (C_o) to execute. This is more complicated because the new average number of cycles per instruction (C_i) goes up when the new feature (the new instruction) is used more frequently (as F_i increases). When the frequency of this instruction is small ($F_i \approx 0$), the new average will still be close to the original average ($C_i \approx C_o$). However, as the frequency of this instruction increases, the new average becomes bigger than the original average ($C_i > C_o$) and the p factor $\left[p = \frac{T_o}{T_i} \times \frac{C_o}{C_i}\right]$ decreases. In this case, even if the cycle time is not affected by the new feature ($T_i = T_o$), you can enjoy the performance improvement of p=1 only when F_i is small.

4.2. LISP Support Evaluation

The SPUR CPU supports LISP by storing the type and generation information in the 8-bit tag field of the register (Figure 2-1-2) and performing tag checking in parallel with the execution of the following instructions (see Appendix A for a detailed discussion of the instruction set):

Add, Sub, And, Or, Xor, Sll, Sra, and Srl:

The CPU checks both operands to ensure both operands are 32-bit integers (Fixnum).

Store_40:

The CPU checks the generations of the operands to ensure the generation boundary is not crossed.

Cxr and Cxr_ro:

The CPU checks the pointer of the operand to make sure it is either a Cons or Nil.

Cmp_branch and Cmp_trap:

If the branch condition requires comparing the lower 32-bit of the two operands, the CPU checks to make sure either both operands are Fixnum or both operands are Character.

The impact of hardware tag checking on LISP program performance, resources allocation, and complexity are evaluated in Section 4.2.1, Section 4.2.2, and Section 4.2.3, respectively. The results are summarized in Section 4.2.4.

4.2.1. LISP Support–Impact on Performance

In this section, I will use the performance model developed in Section 4.1 to evaluate the impact of hardware tag checking on performance by comparing the performance of the SPUR CPU against an imaginary stripped down SPUR CPU that does not support tag checking. I will use the subscript "i" for the SPUR CPU (T_i, C_i, and M_i) and subscript "o" for the stripped down CPU (T_o, C_o, and M_o). In order to give the stripped down CPU the benefit of the doubt, I assume the stripped down CPU stores the type and generation information at some easy to access location such that the work it take to store and retrieve this information is the same as reading and writing the tag in the SPUR CPU. Furthermore, I assume that if a type or generation violation occurs, both the CPU and the stripped down CPU will handle the unusual cases similarly. Consequently,

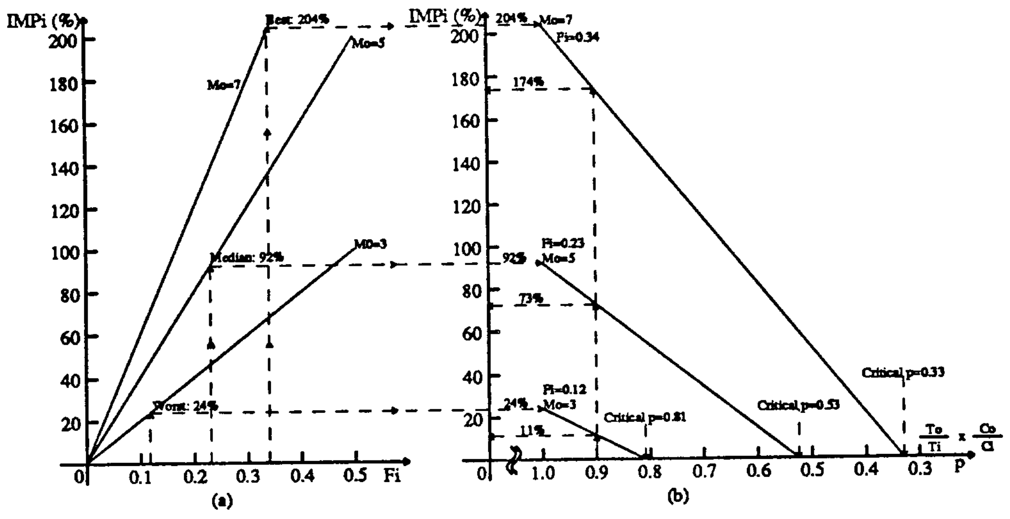


Figure 4-2-1 Performance Improvement due to Tag Checking

(a) shows the performance gain due to hardware tag checking as a function of F_i. The best, ($M_o = 7$, $F_i = 0.34$) median, ($M_o = 5$, $F_i = 0.23$), and the worst arguments ($M_o = 3$, $F_i = 0.12$) for having explicit tag checking are marked in this diagram. (a) assumes $T_i = T_o$ and $C_i = C_o$. (b) shows the effect if hardware tag checking increases the cycle time ($T_i > T_o$), or the average number of cycles to execute a instruction ($C_i > C_o$), or both.

the only difference is that the SPUR CPU checks the type and generation information implicitly in parallel with the execution of certain instructions while programs written for the stripped down CPU must have explicit instructions to do the type and generation checking. When violations occur, the SPUR CPU will trap to the unusual cases handlers while the stripped down CPU will branch to the unusual cases handlers. Since the SPUR CPU checks the tag implicitly whenever the special instructions are executed, the SPUR CPU takes one instruction to perform tag checking:

$$M_i = 1 \qquad (4.2.1)$$

According to the SPUR LISP group [Zor89], LISP programs for the stripped down CPU take between three to seven instructions to do the type or generation checking explicitly:

$$3 \le M_o \le 7 \quad \Rightarrow \quad \text{Median } M_o = 5 \qquad (4.2.2)$$

Since the SPUR CPU checks the tag in parallel with the instruction's execution, the average number of cycles to execute each instructions is NOT affected by adding the feature:

$$C_i = C_o \quad \Rightarrow \quad \frac{C_o}{C_i} = 1 \qquad (4.2.3)$$

The performance improvement, assuming hardware tag checking has no effect on the cycle time $\left[\frac{T_o}{T_i} = 1\right]$, can be estimated using the performance model (Equation 4.1.6) and the values given by Equation 4.2.1, 4.2.2, and 4.2.3. Figure 4-2-1(a) is a plot of the performance improvement as a function of F_i for $M_o = 3$, $M_o = 5$, and $M_o = 7$. According to George Taylor [Tay86], the percentage of instructions that require type and generation checking is between 12% and 34%:

$$12\% \le F_i \le 34\% \quad \Rightarrow \quad \text{Median } F_i = 23\% \qquad (4.2.4)$$

Based on Equation 4.2.2 and Equation 4.2.4, we have

Median argument for having implicit tag checking: $M_o = 5,\ F_i = 0.23$

Best argument for having implicit tag checking: $M_o = 7,\ F_i = 0.34$

Worst argument for having implicit tag checking: $M_o = 3,\ F_i = 0.12$

The performance improvement, assuming hardware tag checking has no effect on the cycle time $\left[\frac{T_o}{T_i}=1\right]$ nor on the average number of cycles per instruction $\left[\frac{C_o}{C_i}=1\right]$ is predicted in Figure 4-2-1(a) to be 204%, 92%, and 24%, respectively. If having hardware tag checking increases the cycle time $(T_i > T_o)$, or the average number of cycles to execute a instruction $(C_i > C_o)$, or both, then Figure 4-2-1(b) predicts the performance improvement to be less than the improvement predicted in Figure 4-2-1(a). For example, the 24% performance gain predicted by the worst argument will be completely nullified if adding hardware tag checking caused an increase in the cycle time such that $\frac{T_i}{T_o} \approx 0.8$. As will be explained at the end of Section 4.2.3, the point $p = 0.4$ can be considered as the place where MIPS-X resides based on Steenkiste's LISP analysis [StH88].

The horizontal axis of Figure 4-2-1(b) is not labeled simply as $\frac{T_o}{T_i}$ although I have argued $\frac{C_o}{C_i} = 1$ in Equation 4.2.3. It is still labeled as $p = \frac{T_o}{T_i} \times \frac{C_o}{C_i}$ to emphasis the fact although hardware tag checking does not affect the average number of cycles per instruction directly (Equation 4.2.3), it may still change the average indirectly. For example, if we do not implement hardware tag checking and somehow can transfer the effort to improve the on-chip instruction cache, this better instruction cache may reduce the average number of cycles per instruction form 1.8 to 1.5. We can then still achieve p=0.8 even if the cycle time is not changed $\left[\frac{To}{Ti}=1\right]$:

$$p = \frac{T_o}{T_i} \times \frac{C_o}{C_i} = \frac{T_o}{T_i} \times \frac{1.5}{1.8} = 0.83 \times \frac{T_o}{T_i}$$

I have defined the critical p factor ($p_{critical}$) as the value of p at which performance improvement of the SPUR CPU over the stripped down CPU is zero. As shown in Figure 4-2-1(b), the critical p factor for the best case is $p_{critical} = 0.33$, the median case is $p_{critical} = 0.53$, and the worst cases is $p_{critical} = 0.81$. Let us give the stripped down CPU the benefit of the doubt and assume the effort we saved in not implementing hardware tag checking can be used to improve the cycle time and average number of cycles per instruction each by 5%:

$$p = \frac{T_o}{T_i} \times \frac{C_o}{C_i} = 0.95 \times 0.95 = 0.90$$

The performance improvement due to LISP support is reduced to 173% for the best case, 73% for the median case, and 12% for the worst case.

4.2.2. LISP Support–Impact on Resources

The hardware tag checking that supports LISP requires six special instructions, eight extra tag bits in the lower datapath, six extra branch conditions and four extra trap conditions. The six special instructions are (see Table A-3-1 and Table A-3-2 in appendix A for a more detail discussion of these instructions and Cache Operations):

(1) LD_40 – Load the 32-bit data and 8-bit tag field of the 40-bit register simultaneously.

(2) LD_40_RO – Similar to LD_40 except it sends a special Cache Operation to the Cache Controller for multiprocessing.

	Six Extra Instructions		Eight Tag Bits		Extra Branch Conditions		Extra Trap Conditions		Total
Control PLA Outputs	4/54	7%	0/54	0%	0/54	0%	0/54	0%	7%
Control PLA Products	2.2/84	3%	0/84	0%	0/84	0%	0/54	0%	3%
Chip Area (mm x mm)	0/57	0%	2.7/57	4.7%	0.1/57	0.2%	1.1/57	1.9%	7%
Transistors (x 1000)	0/115	0%	8.7/115	7.6%	0.1/115	0.1%	1.1/115	0.9%	9%
Number of Signal Pins	0/156	0%	8/156	5%	0/156	0%	0/156	0%	5%

Table 4-2-1 Resources Metrics for Hardware Tag Checking

Each column lists the absolute and percentage impact of each sub-feature on the different resources metrics. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. The total impact of these sub-features, that is the impact due to hardware tag checking, is the sum of four columns and is shown in the right most column.

(3) **CXR** – Special LD_40 instruction that perform LISP pointer type checking in parallel.

(4) **CXR_RO** – Similar to CXR except it sends a special Cache Operation to the Cache Controller for multiprocessing.

(5) **RD_TAG** – Read the 8-bit tag field of the register.

(6) **WR_TAG** – Write the 8-bit tag field of the register.

The six extra branch conditions (Table A-3-9, Appendix A) check the 6-bit type tag (Figure 2-1-2) of the operands:

(1) **EQ_TAG** – Checks whether the 6-bit type tags of the two operands are equal.

(2) **NE_TAG** – Checks whether the 6-bit type tags of the two operands are not equal.

(3) **EQ_38** – Checks whether the 32-bit data and the 6-bits type tags of the two operands are equal.

(4) **NE_38** – Checks whether the 32-bit data and the 6-bits type tags of the two operands are not equal.

(5) **EQ_TC** – Checks whether the 6-bit type tag of the first operand equals a six-bit constant.

(6) **NE_TC** – Checks whether the 6-bit type tag of the first operand equals a six-bit constant.

The four extra trap conditions are (Table A-3-8, Appendix A):

(1) **LISP Pointer Type Violation** – The pointer is neither a CONS nor a NIL.

(2) **LISP Data Type 1 Violation** – Either operand is not a FIXNUM.

(3) **LISP Data Type 2 Violation** – Both operands are not FIXNUM or both operands are not CHARACTER.

(4) **Generation Violation** – Generation of the second operand is greater than the first operand.

The six extra instructions, the eight extra tag bits, the six extra branch conditions, and the four extra trap conditions have different impact on resource allocation. Their different impact must be measured by different resources metrics. The left most column of Table 4-2-1 shows the resources metrics I selected. Since each metric's absolute value has different dimension, comparing different metrics' absolute values is like comparing apple and oranges. Therefore, I find it more useful to look at the dimensionless percentage impact on each metric. The percentage impact is calculated by dividing the absolute impact by the total number of that metric in the SPUR CPU chip. For example, the eight extra tag bits increase the area by $8.7mm^2$ (absolute impact). Since the total active chip area in the SPUR CPU is $57mm^2$, the percentage impact is $\frac{8.7}{57} = 7.6\%$.

The first row of Table 4-2-1 shows that the SPUR CPU master control PLA has a total 54 outputs. Four of these 54 outputs (7%) are used to control the six extra instructions. Since all other sub-features does not affect the number of outputs in master control PLA, this 7% is the total impact due to hardware tag checking. Similarly, the rightmost column of the second, third, fourth, and fifth row show that tag checking for LISP support are responsible for 3% of the master control PLA product terms, consumes 7% of the total active area, 9% of the total transistors, and 5% of the total signal pins. Notice that the six extra instructions' impact on resource (Column 1) is mainly at the Control PLAs. Their impact on chip area and transistors count are minimum. On the other hand, the eight extra tag bits (Column 2) have minimum impact of the Control PLAs but affect the chip area, transistors count, and the number of signal pins. Finally, the extra branch and trap conditions' impact (Column 3 and 4) is mainly on the chip area and transistors count.

4.2.3. LISP Support–Impact on Complexity

The complexity due to the LISP supporting features can be quantified by the effort to verify their correctness by simulation. The LISP supporting features of the SPUR CPU are simulated at both the behavioral and switch level. The diagnostics we used for switch level simulation are:

(1) **cmp-tag_insts** – This diagnostic takes 470 cycles to verify that cmp_branch instructions that use branch conditions involving tag comparison.

	Absolute Impact	Percentage Impact
Cycles of Diagnostics	4,665	4,665/55,516 = 8%
Man-Month of Effort	0.5	0.5/3.5 = 14%

Table 4-2-2 Complexity Metrics for Hardware Tag Checking

The first column lists the absolute impact on the complexity metrics due to hardware tag checking. The second column lists the percentage impact. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU.

(1) **fixnum-trap** – This diagnostic takes 1086 cycles to verify that all LISP data type violations will cause a trap.

(2) **fast-reg-tags** – This diagnostic takes 1169 cycles to verify that the CPU can read and write the tag field of all the registers.

(3) **gen-traps** – This diagnostic takes 509 cycles to verify that all generation violation will cause a trap.

(4) **cxr-traps** – This diagnostic takes 1159 cycles to verify that all LISP pointer type violation will cause a trap.

(5) **trap-psw** – This diagnostic takes 272 cycles to verify that all traps set the processor status work (Kpsw and Upsw) correctly.

The total number of cycles and the man-months of simulation effort are two complexity metrics we can extract from this set of diagnostics. Column 1 of Table 4-2-2 is the absolute value of these two metrics. As I explained in previous section, I found it more useful to look at the dimensionless percentage impact. The percentage impact of each metric is calculated in Column 2. The total number of cycles of diagnostics we run for the SPUR CPU's switch level simulation is 55,516 cycles. The total switch level simulation effort is 3.5 man month. These two numbers

	Pessimistic View	Realistic View	Optimistic View
Performance Impact	Worst 11%	Median 73%	Best 174%
Resources Impact	Biggest 9%	Median 6%	Smallest 3%
Complexity Impact	Biggest 14%	Median 11%	Smallest 8%

Table 4-2-3 Three Different Views of the Tradeoffs

The pessimistic view uses the smallest performance impact and the biggest resource and complexity impact. The realistic view uses the median performance impact, median resource impact, and median complexity impact. The optimistic view uses the biggest performance impact and the smallest resources and complexity impact.

are used in Column 2 to calculate the percentage of the total effort.

4.2.4. LISP Support–Impact Summary

The impact of hardware tag checking on LISP program performance, resources allocation, and complexity are illustrated in Figure 4-2-1, Table 4-2-1, and Table 4-2-2, respectively. The results of these figure and tables are summarized in Table 4-2-3, giving three different views of the performance, resources, and complexity tradeoffs. These three views of the tradeoffs are shown graphically in Figure 4-2-2. The median and optimistic views both seems to indicate the hardware tag checking is a good feature because the percentage improvement in performance is big while the percentage increases in resources and complexity are relatively small. Furthermore, as illustrated in Figure 4-2-1(b), both the best and median performance improvement arguments have a relatively small critical p factors ($p_{critical}$=0.53 for the median case and $p_{critical}$=0.33 for the best case). In other words, even if the cycle time or the average number of cycles per instruction or both are affected moderately by having hardware tag checking, the SPUR CPU having tag checking will still come up a winner.

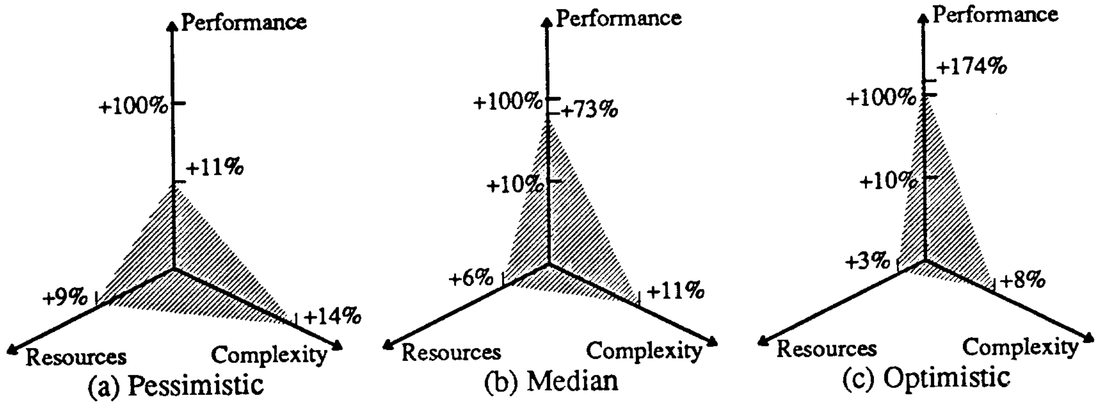


Figure 4-2-2 Three Different Views of the Tradeoffs

(a) shows the pessimistic view which uses the smallest performance impact and the biggest resources and complexity impact. (b) shows the median view which uses the median performance impact, median resources impact, and median complexity impact. (c) shows the optimistic view which uses the biggest performance impact and the smallest resources and complexity impact. For clarity, logarithmic scale is used for the performance axis.

Finally, the performance analysis shown in Figure 4-2-1 can also be used to compare the LISP performance between the SPUR CPU and the MIPS-X. Both the SPUR CPU and the MIPS-X are RISC style load-store processors aim for single cycle execution. The MIPS-X, however, does not have hardware tag checking. The MIPS-X has a cycle time of 50ns and its average number of cycles per instruction is also lower than the SPUR CPU mainly due to its higher internal instruction cache hit rate. Therefore for a first order approximation, MIPS-X can be considered as a stripped down SPUR CPU that does not have hardware tag checking but have a faster cycle time and lower number of cycles per instruction. More specificly, assuming the MIPS-X cycle time is 50ns and the average number of cycle per instruction is 1.2 (80% better than the SPUR CPU), we have:

$$p = \frac{T_o}{T_i} \times \frac{C_o}{C_i} = \frac{50}{100} \times \frac{1.2}{1.5} = 0.40$$

The performance improvement due to LISP support is now reduced to 22% for the best case, –23% for the median case, and –50% for the worst case. In other words, for the best case (SPUR CPU's point of view), the MIPS-X performance is only 82% (1/1.22) of the SPUR CPU's performance. However, for the median and worst case, the MIPS-X is 30% (1/0.77) and 100% (1/0.5) faster. Incidently, these numbers agrees with Steenkiste's analysis [StH88].

These numbers, however, should not be used to draw the conclusion that hardware tag checking is a bad idea. It will be unfair because hardware tag checking should not be blamed for the SPUR CPU's relatively slow cycle time (100ns) and small internal instruction cache (128 instructions). The SPUR CPU cycle time is limited by system considerations, conservative circuit design, and the conservative 4-phase non-overlap clocking scheme. The main reason why the SPUR CPU has a small internal instruction cache is that we try to be conservative and build the instruction cache using the relatively large static RAM cells that were used in the register file. A better conclusion is that the MIPS-X designers, who are willing to take more risks, used more aggressive circuit designs to lower the cycle time and increase the size of the internal instruction cache. Under most circumstance, these improvements are enough to offset the SPUR CPU's hardware tag checking.

4.3. FPU Support Evaluation

The SPUR CPU supports floating point arithmetic by a coprocessor–the Floating Point Unit (FPU). The FPU is connected to the SPUR CPU via a parallel coprocessor interface [HaK86]. Detailed discussions of coprocessor interface and FPU design can be found in [Han88] and [Bos88], respectively. The floating point instructions supported by the FPU are:

(1) F_ADD – Floating point add.

(2) F_SUB – Floating point subtract.

(3) F_MUL – Floating point multiply.

(4) **F_DIV** – Floating point divide.

(5) **F_CMP** – Floating point compare.

(6) **F_MOV** – Floating point move.

(7) **F_ABS** – Find the absolute value of a floating point number.

(8) **F_NEG** – Negate a floating point number.

(9) **CVTS** – Convert a double precision number to single precision.

(10) **CVTD** – Convert a single precision number to double precision.

(11) **SYN** – Synchronize the CPU and the FPU.

The CPU treat all these instructions as NOOP when the FPU is disabled and treat them as illegal instructions when FPU is enabled. F_ADD, F_SUB, F_CMP, F_MUL, and F_DIV can be considered as major FPU instruction because it is the FPU's goal to provide the ADD, SUB, MUL, DIVIDE and CMP operations. The other FPU instructions can be considered as supporting instructions because these instructions are provided to make the FPU operate more efficiently. Section 4.3.1 will focus on the performance impact of the major FPU instructions. Section 4.3.2, and Section 4.3.3 discuss FPU support's impact on resources and complexity in the SPUR CPU's perspective. Section 4.3.4 summarizes the results.

4.3.1. FPU Support–Impact on Performance

This section I will use the performance model developed in Section 4.1 to evaluate the impact of FPU support on floating point intensive program's performance by comparing the SPUR CPU to an imaginary stripped down SPUR CPU that does not support floating point operation. As before, I use the subscript "i" for the SPUR CPU (T_i, C_i, and M_i) and subscript "o" for the stripped down CPU (T_o, C_o, and M_o). As far as the SPUR CPU is concerned, each floating point operation is supported by a floating point instruction to be executed by the FPU, therefore:

	Precision	Execution Time (us)	Number of Cycles (N_{RISC})	Approx. Number of Instruction (M_{RISC})
Add/Sub	S	63	472.5	80
	D	83	622.5	100
Multiply	S	110	825	140
	D	680	5100	850
Divide	S	191	1432.5	240
	D	712	5340	900

Table 4-3-1 RISC I Floating Point Operations Measurements

Column 2 are the execution time of the various floating point operations. Using the numbers in Column 2 and Equation 4.3.2, we can calculate the numbers in Column 3. Then using the numbers in Column 3 and Equation 4.3.3, we can calculate the numbers in Column 4.

$$M_i = 1 \tag{4.3.1}$$

On the other hand, in the stripped down CPU, the floating point operations must be performed by floating point routines. The execution time of various floating point routines on a RISC I simulator running at (7.5/4)MHz were measured by Sippel [Sip82]. I have divided the 7.5MHz quoted in Sippel's report by four because RISC I has a four-phase clock and 7.5 MHz is the frequency between phases. Since we know the average number of cycles per instruction for RISC I is approximately 1.5 ($C_{RISC} = 1.5$), we can calculate the approximate number of instructions RISC I takes to emulate the various floating point operations ($M_{RISC}I$) by using Equation 4.3.2 and 4.3.3. The results are summarized in Table 4-3-1:

$$N_{RISC} = ET_{RISC} \times \frac{1}{T_{RISC}} = ET_{RISC} \times \frac{7.5}{4} MHz \tag{4.3.2}$$

$$M_{RISC} = \frac{N_{RISC}}{C_{RISC}} \approx \frac{N_{RISC}}{1.5} \tag{4.3.3}$$

where

N_{RISC} = Number of cycles to execute the floating point operation in RISC I

ET_{RISC} = Execution time of the floating point operation in RISC I

T_{RISC} = Cycle time of RISC I simulator = $\frac{4}{7.5MHz}$

M_{RISC} = Number of instructions to execute the floating point operation in RISC I

C_{RISC} = Average number of cycles per instruction in RISC I $\approx$ 1.5

Assume the number of instructions to execute the various floating operations in the stripped down CPU are similar to those in RISC I and the number of operations to execute a floating point compare is similar to that of floating point subtract, we have:

$$80 \leq M_o \text{ (Add/Sub/Cmp)} \leq 100$$

$$140 \leq M_o \text{ (Multiply)} \leq 850$$

$$240 \leq M_o \text{ (Divide)} \leq 900$$

The coprocessor interface that connects the SPUR CPU and the FPU allows them to operate together in two different modes:

Sequential Mode

After issuing a FPU instruction, the CPU must wait until the FPU finishes before continuing its operation. Since it takes the FPU longer to execute any FPU instruction than the CPU takes to execute an integer instruction, the average number of cycles per instruction will increase when a FPU instruction is executed ($C_i > C_o$).

Parallel Mode

After issuing a FPU instruction, the CPU continues to execute integer instructions and will stall only if it encounters another FPU instruction and the FPU is still busy from a previous FPU instruction. For the best scenario, there are enough integer instructions in between FPU instructions and the average number of cycles per instruction will NOT increase when FPU instruction is executed ($C_i = C_o$).

In order to derive the equation that relates C_i and C_o for the sequential and parallel mode, I need to define the following terms:

E_j = Number of cycles to execute instruction type j

F_j = Frequency of instruction type j

Assume we have N types of CPU (integer) instructions and P type of floating point instructions, then the average number of cycles per instruction for the SPUR CPU operating in sequential mode with the FPU is C_i:

$$C_i = \sum_{j=1}^{N} F_j \times E_j + \sum_{j=N+1}^{N+P} F_j \times E_j \tag{4.3.4}$$

For the stripped down CPU that does not support floating point instructions, the average number of cycles per instruction is C_o:

$$C_o = \sum_{j=1}^{N} \bar{F}_j \times E_j \tag{4.3.5}$$

The term $\bar{F}_j$ (j=1,2,3 ... N) is used in Equation 4.3.5 to emphasis the fact that the frequency of the integer instructions may change when floating point instructions are eliminated. Let us assume the changes in frequencies are small then:

	F_j		E_j	$F_j \times E_j$		M_j		$F_j{*}M_j$	
	Min	Max		Min	Max	Min	Max	Min	Max
FADD	2.4%	3.3%	4	0.096	0.132	80	100	1.92	3.30
FSUB	1.7%	2.4%	4	0.068	0.096	80	100	1.36	2.40
FCMP	1.4%	1.9%	4	0.056	0.076	80	100	1.12	1.90
FMUL	3.2%	4.5%	7	0.224	0.315	140	850	4.48	38.25
FDIV	1.3%	1.9%	19	0.247	0.361	240	900	3.12	17.10
Total	10.0%	14.0%	–	0.691	0.980	–	–	12.0	62.95

Table 4-3-2 Impact of FPU Support on Performance

Every column, except Column E_j, is divided into two sub-columns that corresponds to the minimum and maximum values of that column. The minimum sub-column of Column $F_j \times E_j$ is calculated from (Min F_j) $\times E_j$. The maximum sub-column of Column $F_j \times E_j$ is calculated from (Max F_j) $\times E_j$. The minimum sub-column of Column $F_j \times M_j$ is calculated from (Min F_j) $\times$ (Min M_j). The maximum sub-column of Column $F_j \times M_j$ is calculated from (Max F_j) $\times$ (Max M_j).

$$\sum_{j=1}^{N} \bar{F}_j \times E_j \approx \sum_{j=1}^{N} F_j \times E_j \qquad \textbf{(4.3.6)}$$

Consequently, by combining Equation 4.3.4, 4.3.5, and 4.3.6, we have for sequential mode:

$$C_i \approx C_o + \sum_{j=N+1}^{N+P} F_j \times E_j \qquad (4.3.7)$$

Similarly for parallel mode:

$$C_i \approx C_o + (1 - POR_{par}) \times \sum_{j=N+1}^{N+P} F_j \times E_j \qquad (4.3.8)$$

where

POR_{par} = Portion of FPU operations in parallel with CPU operations

Table 4-3-2 summarized the typical values for the frequency of various floating point operations–F_j [Pat89] [Tay89], the number of cycles it takes the SPRU FPU to execute these operations–E_j [Bos88], and the number of instructions the stripped down CPU takes to emulate

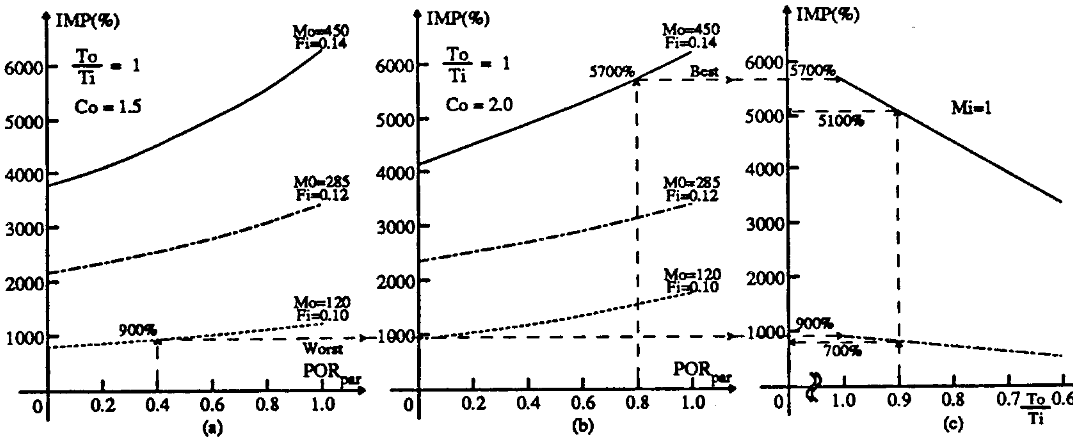


Figure 4-3-1 Performance Improvement due to FPU Support

(a) and (b) show the performance improvement as a function of the frequency of the coprocessor (FPU) instructions (F_i) and the portion of FPU instructions that can be executed in parallel with CPU instructions (POR_{par}). I have assumed the average number of cycles per CPU instruction to be 1.5 for (a) and 2.0 for (b). (c) shows the effect on the best and worst case if supporting the FPU degrades the CPU cycle time ($T_i > T_o$).

these operations–M_j (Table 4-3-1). The products $F_j \times E_j$ and $F_j \times M_j$ are also calculated in this table. The numbers in Table 4-3-2 can be used to calculate the numbers needed by the performance model (Equation 4.1.6) using the following formulas:

$$\sum \text{Min}\, F_j \le F_i \le \sum \text{Max}\, F_j$$

$$\Rightarrow \quad 10\% \le F_i \le 14\% \quad \Rightarrow \quad \text{Median}\, F_i = 12\% \tag{4.3.9}$$

$$\frac{\sum \text{Min}\,(F_j \times M_j)}{\sum \text{Min}\, F_j} \le M_o \le \frac{\sum \text{Max}\,(F_j \times M_j)}{\sum \text{Max}\, F_j}$$

$$\Rightarrow \quad 120 \le M_o \le 450 \quad \Rightarrow \quad \text{Median}\, M_o = 285 \tag{4.3.10}$$

$$\sum \text{Min}\,(F_j \times E_j) \le \sum F_j \times E_j \le \sum \text{Max}\,(F_j \times E_j)$$

$$\Rightarrow \quad 0.691 \le \sum F_j \times E_j \le 0.980 \quad \Rightarrow \quad \text{Median} \sum F_j \times E_j = 0.836 \tag{4.3.11}$$

In the formulas above, the summation indexes $j = N+1$ and $j = N+P$ are dropped for clarity. The values given by Equation 4.3.11 are used in Equation 4.3.8 to calculate the effective C_i in terms of C_o. These numbers can then be used with the performance model (Equation 4.1.6) to calculate the performance improvement due to the FPU supporting features. This is shown in Figure 4-3-1 as a function of the frequency of FPU instructions (F_i), the portion of FPU instructions that can be executed in parallel with CPU instructions (POR_{par}), and the cycle time ratio $\left[\frac{T_o}{T_i}\right]$.

Each line in Figure 4-3-1(a) and (b) assume a fixed frequency of FPU instructions (10%, 12% and 14%) and a fixed average number of cycles per CPU instruction–1.5 in (a) and 2.0 in (b). Notice that Figure 4-3-1(a) and Figure 4-3-1(b) predict the same amount of performance improvements when $POR_{par} = 1.0$ because all FPU instructions are executed in parallel with the CPU instructions. Consequently, $\frac{C_o}{C_i} = 1$ at $POR_{par} = 1$ for both graphs (a) and (b). Assume the portion of FPU instructions that can be executed in parallel with the CPU integer instructions is between 40% and 80%, we have the following best and worst case scenarios:

Worst Case:

Frequency of FPU instruction is only 10% ($F_i = 0.10$), the average number of instructions the stripped down CPU takes to emulate the FPU operations is 120 ($M_o = 120$), and the average number of cycles per CPU instruction is 1.5 ($C_o = 1.5$).

Best Case:

Frequency of FPU instruction is 14% ($F_i = 0.14$), the average number of instructions the stripped down CPU takes to emulate the FPU operations is 450 ($M_o = 450$), and the average number of cycles per CPU instruction is 2.0 ($C_o = 2.0$).

Figure 4-3-1(a) predicts even at the worst case, the FPU support improves the floating point performance by 900%–the SPUR CPU is 10 times faster than the stripped down CPU. Figure 4-4-1(b) predicts at the best case, the FPU support improves the floating point performance by 5700%–the SPUR CPU is 58 times faster than the stripped down CPU. Both these two cases have assumed the FPU support has no effect on the CPU cycle time. However, as shown in Figure 4-3-1(c), even if the cycle time is degraded by 10%, the performance improvement is still respectable: 5100% for the best case and 700% for the worst case.

4.3.2. FPU Support–Impact on CPU Resources

In addition to the 11 floating point instructions listed in the beginning of this section, the FPU support capabilities also requires eight load instructions, four store instructions, a coprocessor interface, two extra branch conditions, and one trap condition. As far as the SPUR CPU is concerned, these extra FPU load and store instructions are similar to the CPU load and store instructions except that the FPU will receive or provide the data. The SPUR CPU is still responsible for generating the effective address and send the proper Cache Operation code to the Cache Controller. The two extra branch conditions are:

(1) FPU_TRUE – Branch if the floating point compare instruction results in a true condition.

(2) FPU_FALSE – Branch if the floating point compare instruction results in a false condition.

The extra trap condition is:

	FPU Instructions		Coprocessor Interfaces		Total
Control PLA Outputs	3/54	6%	0/54	0%	6%
Control PLA Products	4/84	5%	0/84	0%	5%
Chip Area (mm x mm)	0/57	0%	6/57	10%	10%
Transistors (x 1000)	0/115	0%	0.4/115	0%	0%
Number of Signal Pins	32/156	21%	5/156	3%	24%

Table 4-3-3 Resources Metrics for FPU Support

Each column lists the absolute and percentage impact of each sub-feature on the different resources metrics. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. The total impact of these sub-features, that is the impact due to FPU support, is summarized in the right most column.

(1) FPU_EX – FPU exception.

The impact of FPU support on resources are quantified into several resources metrics in Table 4-3-3. The impact of the two extra branch conditions and the one extra trap condition on resources is so small that it is not listed in the table. The number of transistors consumed by the FPU support is also negligible. Area consumption is mainly due to the coprocessor interface which include suspension logic in the master control, a special register FpuPC, and routing the instruction busI onto the output pads. FPU support also consumes 37 of the total 156 signal pads. The SPUR CPU must broadcast the instruction currently being fetched via 32 of these pads. This is the only way the FPU can find out the current instruction because the internal instruction caches makes the instruction currently being fetched invisible to the outside world.

4.3.3. FPU Support–Impact on CPU Complexity

The complexity due to the FPU supporting features can be quantified by the simulation

	Absolute Impact	Percentage Impact
Cycles of Diagnostics	1,543	1,543/55,516 = 3%
Man-Month of Effort	0.5	0.5/3.5 = 14%

Table 4-3-4 Complexity Metrics for FPU Support

The first column lists the absolute impact on the complexity metrics due to FPU support. The second column lists the percentage impact. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU.

effort. The diagnostics we used for switch level simulation are:

(1) **fpu-fpu_busy** – This diagnostic takes 190 cycles to verify that the CPU will stall suspend the pipeline when the FPU is busy and the CPU wants to issue a new FPU instruction.

(2) **fpu-enable_fpop** – This diagnostic takes 142 cycles to verify that the CPU will treat FPU operations instructions as illegal instruction when the FPU is disabled and treat them as FPU instructions when the FPU is enabled.

(3) **fpu-enable_ld** – This diagnostic takes 144 cycles to verify that the CPU will treat FPU load and store instructions as illegal instruction when the FPU is disabled and treat them as FPU instructions when the FPU is enabled.

(4) **fpu-fpc** – This diagnostic takes 148 cycles to verify that the address of the last FPU instruction issued by the CPU is stored in the special register FpuPC.

(5) **fpu-serial** – This diagnostic takes 173 cycles to verify that the CPU and FPU can operate in sequential mode.

(6) **fpu-sync** – This diagnostic takes 142 cycles to verify that the CPU and FPU operations can be synchronized by the SYNC instruction.

(7) **fpu-cpu_trap** – This diagnostic takes 153 cycles to verify that the FPU operation can survive a CPU trap.

(8) **fpu-fpu_trap** – This diagnostic takes 144 cycles to verify that the FPU operation can interrupt the CPU operation via a FPU exception.

(9) **fpu-cmp** – This diagnostic takes 146 cycles to verify that the CPU can correctly execute a cmp_branch that uses the FPU branch conditions.

(10) **fpu-allopcodes** – This diagnostic takes 161 cycles to verify that the CPU can correctly identify all FPU instructions.

The total number of cycles and the man-month of simulation effort are two complexity metrics we can extracted from this set of diagnostics. These are summarized in Table 4-3-4.

4.3.4. FPU Support–Impact Summary

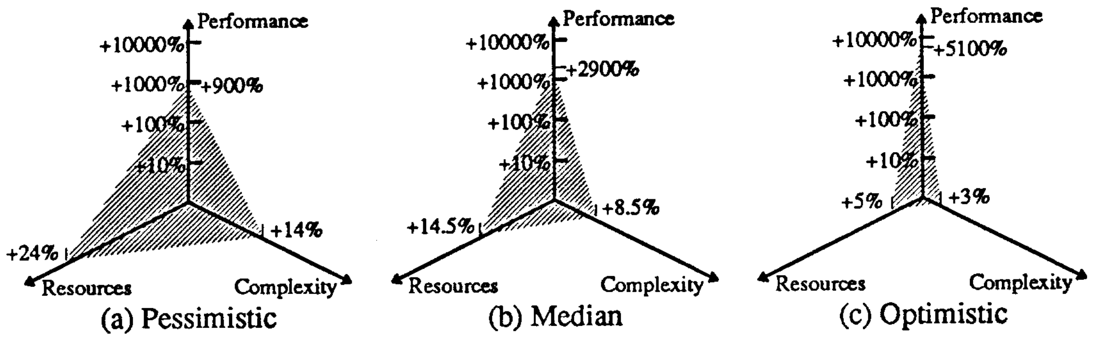


Figure 4-3-2 Three Different Views of the Tradeoffs

(a) shows the pessimistic view which uses the smallest performance impact and the biggest resources and complexity impact. (b) shows the median view which uses the median performance impact, median resources impact, and median complexity impact. (c) shows the optimistic view which uses the biggest performance impact and the smallest resources and complexity impact. For clarity, the performance axis is on a logarithmic scale.

The FPU support's impact on performance (Figure 4-3-1), resources (Table 4-3-1), and complexity (Table 4-3-2) can be summarized in the following inequalities:

$$700\% \leq \text{Performance Impact} \leq 5100\% \quad => \quad \text{Median Performance Impact} = 2900\%$$

$$5\% \leq \text{Resources Impact} \leq 24\% \quad => \quad \text{Median Resource Impact} = 14.5\%$$

$$3\% \leq \text{Complexity Impact} \leq 14\% \quad => \quad \text{Median Complexity Impact} = 8.5\%$$

In Table 4-3-3, the FPU support's impact on transistors is 0%. For conservative analysis, this is not used as the minimum resources impact. Instead, the next smallest increase (5%) is used. These numbers indicate the use of the coprocessor FPU via a coprocessor interface to support floating point operations is a good idea because the FPU support only increases the resource and complexity by a small amount but improves floating point intensive programs drastically. This is of course only the CPU's perspective. A lot of resources and complexity not included in this tradeoffs analysis are involved in the design and implementation of the coprocessor FPU [Bos88] and the coprocessor interface. Furthermore, the performance improvement is for floating point intensive programs only. If a program does not have any floating point operations, the coprocessor interface and the FPU will not improve the program's performance.

4.4. Extra Pipeline Stage Evaluation

The only difference between the SPUR CPU pipeline and the RISC II pipeline is shown in Figure 4-4-1. The SPUR CPU pipeline has an extra memory access stage (Mem) that allows the SPUR CPU to execute LOAD without suspending the pipeline. The extra pipeline stage's impact on performance, resources, and complexity are evaluated in Section 4.4.1, 4.4.2, and 4.4.3 respectively. The results are summarized in Section 4.4.4.

4.4.1. Extra Pipe Stage–Impact on Performance

This section evaluates the extra pipeline stage's impact on performance by comparing the performance of the SPUR CPU against an imaginary stripped down SPUR CPU that uses the RISC II three stage pipeline. I will use subscript "i" (T_i, C_i, M_i) for the SPUR CPU and the

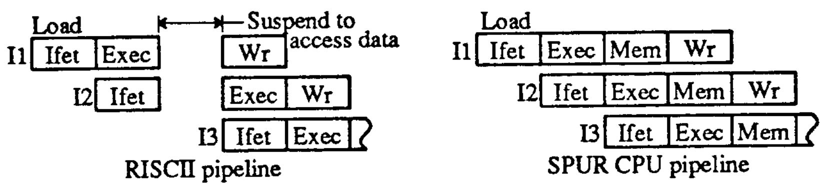


Figure 4-4-1 RISC II Pipeline vs. SPUR CPU Pipeline

This figure assumed the external cache will provide the data within one cycle. Under this assumption, the SPUR CPU 4-stage pipeline will execute LOAD without pipeline suspension. However, since the execution stage (Exec) of the instruction following the LOAD (I2) overlaps the memory access stage (Mem) of the LOAD, I2 cannot use the LOAD's destination register. We call LOAD a delay instruction and I2 the delay slot. On the other hand, the RISC II pipeline is suspended for one cycle whenever LOAD is executed. Due to this one cycle suspension, the execute stage (Exec) of I2 is delayed until after the data access phase of the LOAD and I2 can uses the LOAD's destination register.

subscript "o" (T_o, C_o, M_o) for the stripped down CPU. In the SPUR CPU 4-stage pipeline, the instruction after the LOAD (I2 in Figure 4-4-1) cannot use the destination register of the LOAD. This delay slot must be filled by a NOOP unless we can find a instruction that does not use the the destination register of the LOAD. Therefore in the worst case, the load function is performed by two instructions–the LOAD and the NOOP in the delay slot:

$$M_i = 2 - POR_{fill} \qquad (4.4.1)$$

POR_{fill} = Portion of the delay slot is filled by an useful instruction

On the other hand, in the 3-stage pipeline, the instruction immediately after the LOAD can use the destination register of the LOAD. Therefore, the number of instructions it takes to perform the load function is just one–the LOAD instruction:

$$M_o = 1 \qquad (4.4.2)$$

The average number of cycles per instruction for the SPUR CPU (C_i) is different from the average number of cycles per instruction for the stripped down CPU (C_o) because the 3-stage pipeline must be suspended for data access whenever a LOAD is executed. In order to look at this difference quantitatively, I define the following terms:

N_{load_4} = Average number of cycles to execute LOAD in the 4-stage pipeline,

N_{load_3} = Average number of cycles to execute LOAD in the 3-stage pipeline,

N_{other} = Average number of cycles to execute other instructions in either pipeline,

U_{load} = Number of LOAD instruction in the benchmark,

I_i = Number of instructions it takes the SPUR CPU to execute the benchmark,

I_o = Number of instructions it takes the stripped down CPU to execute the benchmark,

F_i = Frequency of LOAD instructions in the SPUR CPU = $\frac{U_{load}}{I_i}$, *and*

$\bar{F}_{load}$ = Frequency of LOAD instructions in the stripped down CPU = $\frac{U_{load}}{I_o}$.

I assume the number of LOAD instructions (U_{load}) and the average number of cycles to execute instructions other than LOAD (N_{other}) to be the same for programs that are written for either pipeline. Since the SPUR CPU can execute LOAD without pipeline suspension and I assume the external memory can provide data within a cycle (if not, it will affect either pipeline equally), the average number of cycles to execute LOAD in the SPUR CPU 4-stage pipeline is the same as all other instructions:

$$N_{load\_4} = N_{other} \quad => \quad C_i = N_{other} \tag{4.4.3}$$

Since the RISC II 3-stage pipeline always suspend the pipeline for one cycle whenever LOAD is executed, the average number of cycles to execute LOAD is just one more cycles than the average for other instructions:

$$N_{load\_3} = N_{other} + 1$$

$$C_o = \bar{F}_{load} \times N_{load\_3} + (1 - \bar{F}_{load}) \times N_{other} = N_{other} + \bar{F}_{load} \tag{4.4.4}$$

Combining Equation 4.4.3 and Equation 4.4.4, we have:

$$C_o = C_i + \bar{F}_{load} \tag{4.4.5}$$

In general, the frequency of LOAD in the stripped down CPU ($\bar{F}_{load}$) will be slightly bigger than the frequency of LOAD in the SPUR CPU (F_i) because programs written for the SPUR CPU will have some extra NOOPs in the delay slots of LOAD. These extra NOOPs increase the total number of instructions to execute the benchmark (I_i) and since the number of LOAD (U_{load}) is constant, the frequency is lower. The number of NOOPs in the LOAD's delay slots is U_{noop}:

$$U_{noop} = I_i \times F_i \times (1 - POR_{fill})$$

The number of instruction it takes the stripped down CPU to execute a benchmark can be calculated by subtracting the extra NOOPs in the LOAD's delay slot from the number of instruction it takes the SPUR CPU to execute the same benchmark:

$$I_o = I_i - U_{noop} = I_i \times \left[1 - F_i \times (1 - POR_{fill}) \right]$$

The frequency of LOAD in the stripped down CPU can now be calculated as:

$$\tilde{F}_{load} = \frac{U_{load}}{I_o} = \frac{U_{load}}{I_i \times (1 - F_i \times (1 - POR_{fill}))} = \frac{F_i}{1 - F_i \times (1 - POR_{fill})} \qquad (4.4.6)$$

Before we go any further, let us perform couple sanity checks on Equation 4.4.6. Assume $POR_{fill} = 1$, Equation 4.4.6 gives $\tilde{F}_{load} = F_i$. This makes sense because this is the case when all LOAD delay slots are filled with useful instructions in the SPUR CPU. As a second check, assume $POR_{fill} = 0$. The maximum F_i possible for the SPUR CPU 4-stage pipeline with $POR_{fill} = 0$ is 0.5 because there must be a NOOP for every LOAD–half of the instructions are NOOP. Using $POR_{fill} = 0$ and $F_i = 0.5$, Equation 4.4.6 predicts $\tilde{F}_{load} = 1$. This again makes sense because in the stripped down CPU, LOAD does not have to be separated by NOOP. Therefore our two checks show Equation 4.4.6 to be "sane".

Combining Equation 4.4.5 and Equation 4.4.6, we have:

$$C_o = C_i + \frac{F_i}{1 - F_i \times (1 - POR_{fill})}$$

$$\frac{C_o}{C_i} = 1 + \frac{F_i}{C_i \times (1 - F_i \times (1 - POR_{fill}))} \qquad (4.4.7)$$

Conventional wisdom says that longer pipeline usually has shorter cycle time because longer pipeline usually means there will be less thing to do in each pipe stage [Kog81]. This, however, is not the case when we add one more stage to the RISC II 3-stage pipeline to form the

SPUR CPU 4-stage pipeline because:

(1) The increase in the number of stages is not a result of dividing the tasks into smaller pieces. The original task (Ifet, Exec, and Wr) are the same for both pipelines.

(2) The extra Mem stage is a delay stage for all instructions other than LOAD. This extra pipe stage increases the complexity of the datapath and control logic.

Since (1) states that the orginal tasks are not getting any simplier and (2) states that the extra task increases the complexity, the cycle time of the SPUR 4-stage pipeline (T_i) is likely to be bigger than the cycle time of the RISC II 3-stage pipeline (T_o):

$$T_i \geq T_o \quad => \quad \frac{T_o}{T_i} \leq 1 \qquad (4.4.8)$$

The performance improvement of the SPUR CPU 4-stage pipeline over the stripped down 3-stage pipeline is plotted in Figure 4-4-2 as a function of the portion of the delay slot being filled

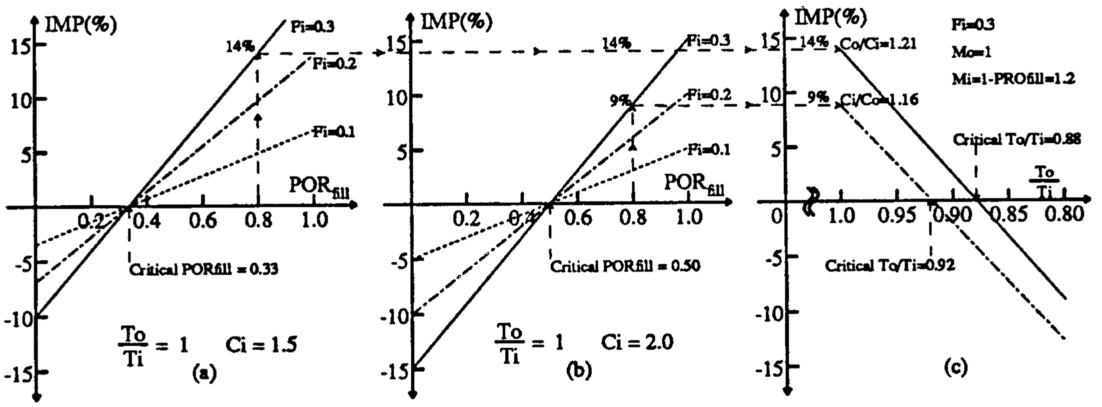


Figure 4-4-2 Performance Improvement due to the Extra Pipe Stage

(a) and (b) show the performance improvement as a function of the portion of the LOAD delay slot being filled (POR_{fill}) by useful instructions and the frequency of the LOAD (F_i). I have assumed the average number of cycles per non-LOAD instruction to be 1.5 for (a) and 2.0 for (b). (a) and (b) assume the pipeline with the extra pipe stage has the same cycle time as the shorter pipeline ($T_i = T_o$). (c) shows the effect if the longer pipeline has a longer cycle time ($T_i > T_o$).

by useful instructions (POR_{fill}), frequency of LOAD (F_i), and the cycle time ratio $\left[\frac{T_o}{T_i}\right]$. This is the result of applying Equation 4.4.1, 4.4.2, 4.4.7, and 4.4.8 to the performance model (Equation 4.1.6). Each line in Figure 4-4-2(a) and (b) assume a fixed frequency of LOAD (10%, 20% and 30%) and a fixed average number of cycles per instruction for all non-LOAD instructions–1.5 in (a) and 2.0 in (b). There are two things worth noticing:

(1) The performance improvement is negative when POR_{fill} is 0 because the 3-stage pipeline only requires the pipeline to suspend for one cycle while the 4-stage pipeline will waste C_i cycles ($C_i > 1$) to execute the NOOP in the delay slot due to misses in the internal instruction cache.

(2) The portion of the delay slot that must be filled (the critical POR_{fill}) in order for for the 4-stage pipeline to have the same performance as the 3-stage pipeline ($IMP_i = 0\%$) is a function of the average number of cycles per non-LOAD instructions (C_i) only. The breakeven point depends on POR_{fill} and not on the LOAD frequency (F_i) because the number of cycles the SPUR CPU wasted whenever a LOAD is executed ($SPUR_{waste}$) is:

$$SPUR_{waste} = C_i \times \left[1 - POR_{fill}\right]$$

At the critical POR_{fill}, the number of cycles the SPRU CPU wasted equals to the number of cycles the 3-stage pipeline must be suspended whenever a LOAD is executed. In the RISC II 3-stage pipeline, this number of cycles is one. Therefore:

$$C_i \times \left[1 - (\text{Critical } POR_{fill})\right] = 1$$

$$\text{Critical } POR_{fill} = 1 - \frac{1}{C_i} \qquad (4.4.9)$$

Using Equation 4.4.9, we have:

$$\text{Critical } POR_{fill}\ (C_i{=}1.5) = 0.33 \qquad \text{Critical } POR_{fill}\ (C_i{=}2.0) = 0.50$$

In the discussions above, we have assumed the only reason why non-LOAD instructions take more than one cycle to execute is due to misses in the internal instruction cache. This is true for the SPUR CPU in which the average number of cycles per non-LOAD instruction (C_i) is estimated to be somewhere between 1.5 and 2.0. Furthermore, let us give the 4-stage pipeline the benefit of the doubt and assume:

(1) The pipeline with the extra pipe stage has the same cycle time as the shorter pipeline.

(2) The frequency of LOAD is 30% ($F_i = 0.3$).

(3) The LOAD delay slot is filled by useful instruction 80% of the time ($POR_{fill} = 0.8$).

The performance improvement, as shown in Figure 4-4-2(a) and (b), is still only between 9% and 14%. However, as discussed earlier (Inequality 4.4.8), the SPUR CPU cycle time is likely to be bigger than that for the stripped down CPU. As shown in Figure 4-4-2(c), the cycle time of the stripped down CPU only has to be approximately 10% smaller than the SPUR CPU cycle time (between $1 - 0.92 = 8\%$ and $1 - 0.88 = 12\%$ to be exact) to neutralize out all the performance advantage of the SPUR CPU 4-stage pipeline.

4.4.2. Extra Pipe Stage–Impact on Resources

The resources impact of the extra pipe stage can be estimated by comparing the resources needed to implement the SPUR CPU 4-stage pipeline to the resources needed to implement the RISC II 3-stage. Relative to the resources needed to implement the RISC II 3-stage pipeline, the

	Extra Temp. Register		Extra PC		Extra Comparators		Extra Control Stage		Total
Chip Area (mm x mm)	1.1/57	2.0%	0.2/57	0.3%	0.1/57	0.2%	0.3/57	0.5%	3%
Transistors (x 1000)	1.8/115	1.6%	0.4/115	0.3%	0.2/115	0.2%	0.4/115	0.3%	2%

Table 4-4-1 Resources Metrics for the Extra Pipe Stage

Each column lists the absolute and percentage impact of each components on the different resources metrics. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. The total impact of these sub-features, that is the impact due to the extra pipe stage, is summarized in the right most column.

extra pipe stage requires:

(1) One extra temporary register (Dst2) in the lower datapath (Figure 2-3-1).

(2) One extra program counter register (MemPC) in the upper datapath (Figure 2-3-2).

(3) Two extra comparators for the internal forwarding logic. Internal forwarding is discussed in Figure 2-1-6.

(4) One extra stage (MemCtrBuf) in the Sequencer of the Master Control (Figure 2-4-3).

All these components increase the area and transistors count of the SPUR CPU. Their impacts are summarized in Table 4-4-1. Their impacts on control PLA's output, control PLA's product terms, and output signal pins, however, are either none or negligible. In order to keep the table simple, these negligible impacts are not shown.

4.4.3. Extra Pipe Stage–Impact on Complexity

Simulation effort cannot be used directly as complexity metric for the extra pipe stage because we do not have a separate set of diagnostics designated just to test the extra pipe stage. However, every SPUR CPU diagnostic checks this extra stage implicitly. I estimated that 15% of all the diagnostics cycles were spent in checking this stage. Furthermore, I also believed that this

	Absolute Impact	Percentage Impact
Cycles of Diagnostics	–	– = 15%
Man-Month of Effort	1	1/3.5 = 29%

Table 4-4-2 Complexity Metrics for the Extra Pipe Stage
The first column lists the absolute impact on the complexity metrics due to the extra pipe stage. The second column lists the percentage impact. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. In Row 1, I do not have the exact values of the Absolute Impact but I can estimate the Percentage Impact.

extra pipe stage makes all diagnostics more complex and increase the simulation effort by one man-month–29%. These two numbers are summarized in Table 4-4-3.

4.4.4. Extra Pipe Stage–Impact Summary

The extra pipe stage's impact on performance (Figure 4-4-1), resources (Table 4-4-1), and complexity (Table 4-4-2) can be summarized in the following inequalities:

$$9\% \leq \text{Performance Impact} \leq 14\% \quad \Rightarrow \quad \text{Median Performance Impact} = 11.5\%$$

$$2\% \leq \text{Resources Impact} \leq 3\% \quad \Rightarrow \quad \text{Median Resource Impact} = 2.5\%$$

$$15\% \leq \text{Complexity Impact} \leq 29\% \quad \Rightarrow \quad \text{Median Complexity Impact} = 22\%$$

The 4-stage pipeline does not use up that many resources but it does increases the complexity. Notice that the best performance improvement is estimated to be only around 14%. This is assuming the SPUR CPU pipeline with the extra stage will have the same cycle time as the stripped down CPU's 3-stage pipeline. As we can see from Figure 4-4-2(c), this performance improvement will disappear quickly if the cycle time of the SPUR CPU is only slightly bigger than the cycle time of the stripped down CPU. Consequently, I do not think the SPUR CPU 4-

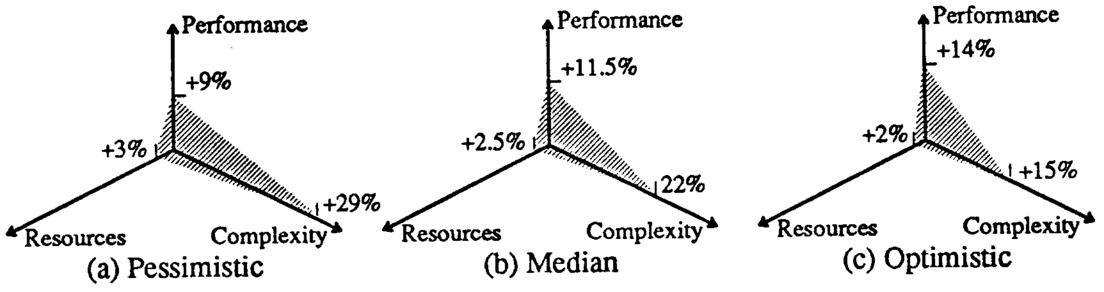


Figure 4-4-3 Three Different Views of the Tradeoffs

(a) shows the pessimistic view which uses the smallest performance impact and the biggest resources and complexity impact. (b) shows the realistic view which uses the median performance impact and median resources and complexity impact. (c) shows the optimistic view which uses the biggest performance impact and the smallest resources and complexity impact.

stage pipeline is a winning feature.

In general, I think designer should be very careful whenever they add "delay" stages in the pipeline to avoid pipeline suspension due to structural conflicts [Kog81] because the performance improvement may be relatively small. There are two reasons for this caution:

(1) Due to data or branch hazard, adding a delay stage in the pipeline is likely to end up creating a delay instruction and the delay slot must be filled for the longer pipeline to be an advantage.

(2) Adding a delay stage increases the length and thus the complexity of the pipeline without finer division of the original task. This may result in a longer cycle time.

4.5. On-Chip Instruction Cache Evaluation

The design and implementation of on-chip instruction cache are discussed in detail by [Hil87a] and [Dun86], respectively. Section 4.5.1, 4.5.2, and 4.5.3 discuss on-chip instruction cache's impact on performance, resources, and complexity of the SPUR CPU. Section 4.5.4 summarizes the results.

4.5.1. On-Chip Instruction Cache–Impact on Performance

This section evaluates the on-chip instruction cache's impact on performance by comparing the performance of the SPUR CPU against an imaginary stripped down SPUR CPU that does not have an internal instruction cache. I will use subscript "i" (T_i, C_i, M_i) for the SPUR CPU and the subscript "o" (T_o, C_o, M_o) for the stripped down CPU. The on-chip instruction cache does not affect the number of instruction to perform any function directly. However, without the on-chip instruction cache, instruction fetch and data access cannot occur in parallel and it becomes impossible to implement the SPUR CPU 4-stage pipeline. Therefore the stripped down CPU must use the shorter RISC II 3-stage pipeline. Based on the discussion in Section 4.4, we have:

$$M_i = 2 - POR_{fill} \tag{4.5.1}$$

POR_{fill} = Portion of the LOAD delay slot is filled by an useful instruction

$$M_o = 1 \tag{4.5.2}$$

The stripped down CPU's average number of cycles per instruction or its cycle time must be bigger than the CPU because it does not have an on-chip instruction cache and must fetch every instruction from the slower external cache. Let us assume:

Assumption 1

The external cache is so big that the instruction miss rate is negligible. It always takes one cycle to fetch an instruction unless instruction fetch is blocked by data access. In other words, the stripped down CPU without an on-chip instruction cache runs slower so that it can fetch and execute one instruction per cycle under most situations.

However, no matter how slow the stripped down CPU runs, it still cannot fetch and execute one instruction every cycle because its 3-stage pipeline must be suspended whenever a LOAD or STORE instruction is executed to avoid data access and instruction fetch conflict. Based on the discussion in Section 4.4 that results in Equation 4.4.5, we get the following equation for stripped down CPU's average number of cycles per instruction (C_o):

$$C_o = \tilde{C}_i + \tilde{F}_{load} + \tilde{F}_{store} \tag{4.5.3}$$

$\tilde{C}_i$ = C_i if the SPUR CPU has a perfect on-chip instruction cache

$\tilde{F}_{load}$ = Frequency of LOAD instructions in the stripped down CPU

$\tilde{F}_{store}$ = Frequency of STORE instructions in the stripped down CPU

The term $\tilde{C}_i$ is used instead of C_i in Equation 4.5.3 because of Assumption 1. Assume the external cache miss rate is low, the term $\tilde{C}_i$ can be expressed as:

$$\tilde{C}_i = C_i - (1 - HIT_{icache}) \times PEN_{icache} \tag{4.5.4}$$

HIT_{icache} = Hit rate of the SPUR CPU on-chip I-cache

PEN_{icache} = Miss penality of the SPUR CPU on-chip I-cache

In order to simply the derivation, I will write the frequency of the STORE instruction as a function of the LOAD instruction:

$$\bar{F}_{store} = \beta \times \bar{F}_{load} \qquad (4.5.5)$$

Applying Equation 4.5.4 and Equation 4.5.5 to Equation 4.5.3, we have:

$$C_o = C_i + (1+\beta) \times \bar{F}_{load} - (1 - HIT_{icache}) \times PEN_{icache} \qquad (4.5.6)$$

As shown previously in Equation 4.4.6, frequency of LOAD in programs written for the stripped down CPU ($\bar{F}_{load}$) can be expressed in terms of the frequency of LOAD in programs written for the SPUR CPU (F_i). Consequently, Equation 4.5.6 can be written as:

$$\frac{C_o}{C_i} = 1 + \frac{(1+\beta) \times F_i}{C_i \times (1 - F_i \times (1 - POR_{fill}))} - \frac{(1 - HIT_{icache}) \times PEN_{icache}}{C_i} \qquad (4.5.7)$$

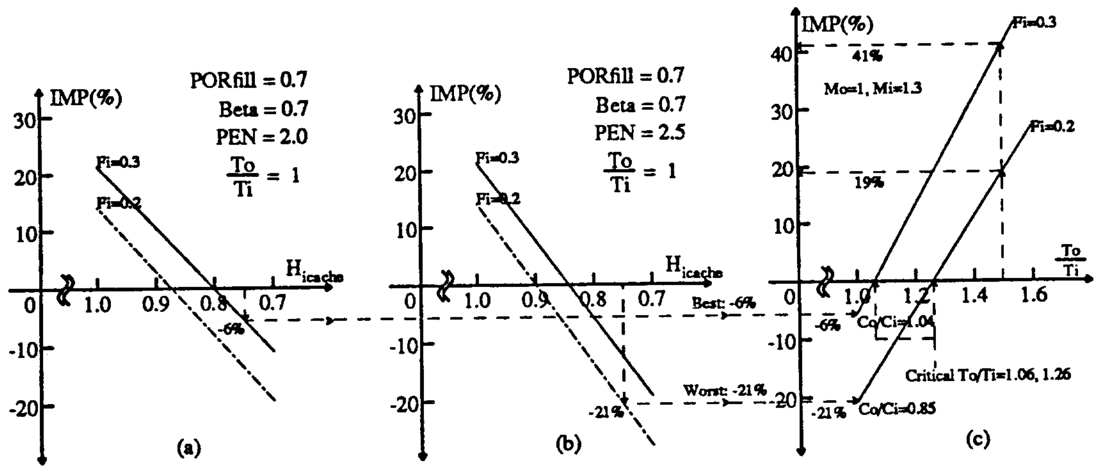


Figure 4-5-1 Performance Improvement due to the On-Chip Instruction Cache

(a) and (b) show the performance improvement as a function of the instruction cache hit rate (H_{icache}) and the frequency of the LOAD (F_i). I have assumed the on-chip instruction cache miss penalty to be 2.0 for (a) and 2.5 for (b). (a) and (b) assume the on-chip instruction cache has no effect on the cycle time. The effect of reduction of cycle time ($T_i < T_o$) due to the on-chip instruction cache is shown in (c).

The performance improvement of the SPRU CPU over the stripped down CPU that does not have an on-chip instruction cache is plotted in Figure 4-5-1 as a function of the probability of the I-cache hit rate (H_{icache}), frequency of LOAD (F_i), and the cycle time ratio $\left[\frac{T_o}{T_i}\right]$. This is the result of applying equations 4.5.1, 4.5.2, and 4.5.7 to the performance model (Equation 4.1.6). In order to reduce the number of variables in the graph, I have assumed:

(1) 70% of the LOAD delay slot are filled by useful instruction ($POR_{fill} = 0.7$) for Equation 4.5.1.

(2) The frequency of STORE is 70% of the LOAD frequency ($\beta = 0.7$) for Equation 4.5.7.

Each line in Figure 4-5-1(a) and (b) assume a fixed frequency of LOAD (20% and 30%) and a fixed instruction cache miss penalty–2.0 in (a) and 2.5 in (b). The lower limit of I-cache miss penalty is two cycles. This limit ($PENALTY = 2.0$) can only be achieved if the external cache can supply the missing instruction within a cycle. Therefore in Figure 4-5-1(b), where the I-cache miss penalty is two and half cycles ($PENALTY = 2.5$), the extra half cycle can be considered as miss penalty of the SPUR CPU external cache. Therefore Figure 4-5-1(b) is biased against the SPUR CPU (worst case) because it assumes the SPUR CPU has to pay an external cache miss penalty while this penalty is assumed to be negligible for the stripped down CPU (see Assumption 1). Notice that Figure 4-5-1(a) and (b) predict the same amount of performance improvement when $H_{icache} = 1$ because neither case has to pay the the I-cache miss penalty when the hit rate is 100%.

Assumption 1 essentially states that all things being equal, the SPUR CPU average number of cycles per instruction will always be higher than the stripped down CPU unless the SPUR CPU has a perfect instruction cache (100% hit rate). Therefore, Figure 4-5-1(a) and (b), which neglect the on-chip instruction cache's effect on cycle time $\left[\frac{T_o}{T_i} = 1\right]$, should predict the performance improvement to be negative for all cases where $H_{icache} < 1$. This is not the case because "all things are not equal" for the SPUR CPU and the stripped down CPU. Besides reducing the cycle time, the internal instruction cache also enable the SPUR CPU to execute LOAD and STORE without suspending the pipeline. This is the only advantage of the on-chip instruction cache when its effect on cycle time is neglected. However, this advantage disappear quickly as the hit rate (H_{icache}) decreases.

Figure 4-5-1(c) shows the on-chip instruction cache's impact on performance when the cache's effect on cycle time is taken into account. We believe the on-chip instruction cache improve the SPUR CPU cycle time by 50% $\left[\frac{T_o}{T_i} = 1.5\right]$. Mark Hill estimated the instruction cache hit rate to be 75% [Hil87b]. George Taylor [Tay86] estimated the frequency of LOAD to be between 20% and 30%. Based on these numbers, we have:

Best Case

30% LOAD, $0.7 \times 30\% = 24\%$ STORE, two cycles miss penalty.

Worst Case

20% LOAD, $0.7 \times 20\% = 14\%$ STORE, two and half cycles miss penalty.

As shown in Figure 4-5-1(c), the performance improvement of these two cases are 41% and 19% respectively if the on-chip instruction cache can improve the cycle time by 50%.

4.5.2. On-Chip Instruction Cache–Impact on Resources

The on-chip instruction cache in the SPUR CPU is organized into an Instruction Unit (Section 2.2) that can be divided into two parts: (1) datapath, and (2) controller. Table 4-5-1 shows the resources impact of the on-chip instruction cache can be estimated by counting the area and transistors consumed by the Instruction Unit. Table 4-5-1 has three columns: Column 1 shows resources consumed by the datapath of the Instruction Unit. Column 2 shows resources consumed by the controller of the Instruction Unit, and Column 3 shows the total resources consumed by the Instruction Unit.

	I-Unit Datapath		I-Unit Controller		Total
Chip Area (mm x mm)	16.7/57	29%	2/57	4%	33%
Transistors (x 1000)	36.4/115	32%	1.2/115	1%	33%

Table 4-5-1 Resources Metrics for the On-Chip Instruction Cache

Each column lists the absolute and percentage impact of each components on the different resources metrics. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. The total impact of the datapath and the controller is the impact due to the on-chip instruction cache and is summarized in the right most column.

The datapath of the Instruction Unit, which consists of the cache and tag array, consumes 32% of the number of transistors but only 29% of the chip area. This is a result of the regularity of the cache and tag arrays. On the other hand, the controller of the instruction cache is not as regular. Consequently, it consumes 4% of the chip area although it only represents 1% of the number of transistors. Neither the controller nor the datapath of the Instruction Unit has any significant impact on the PLA's output nor product terms. This indicates the Instruction Unit is relatively independent from the Execution Unit. In order to keep the Table 4-5-1 simple, these negligible impacts are not shown.

4.5.3. On-Chip Instruction Cache–Impact on Complexity

The complexity due to the on-chip instruction cache can be quantified by the simulation effort. The on-chip instruction cache of the SPUR CPU is simulated at both the behavioral and switch level. The diagnostics we used for switch level simulation are:

(1) **cc-IB_disabled** – This diagnostic takes 364 cycles to verify that the SPUR CPU can at least run with the Instruction Unit disabled.

	Absolute Impact	Percentage Impact
Cycles of Diagnostics	4,994	4,994/55,516 = 9%
Man-Month of Effort	0.25	0.25/3.5 = 7%

Table 4-5-2 Complexity Metrics for the On-Chip Instruction Cache
The first column lists the absolute impact on the complexity metrics due to the on-chip instruction cache. The second column lists the percentage impact. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU.

(2) **cc-IB_fetch** – This diagnostic takes 356 cycles to verify that the SPUR CPU can run with the Instruction Unit enabled but prefetching disabled.

(3) **cc-IB_fetch** – This diagnostic takes 261 cycles to verify that the SPUR CPU can run with the Instruction Unit and prefetching enabled.

(4) **cc-IB_stuck** – This diagnostic takes 4013 cycles to verify that there is no "stuck-at" errors in the Instruction Unit.

The total number of cycles and the man-month of simulation effort are two complexity metrics we can extracted from this set of diagnostics. Column 1 of Table 4-5-2 is the absolute value of these two metrics. The percentage impact of each metric is calculated in Column 2. Notice that the increase in complexity due to the Instruction Unit is relatively small.

4.5.4. On-Chip Instruction Cache–Impact Summary

The impact of the on-chip instruction cache on performance, resources allocation, and complexity are illustrated in Figure 4-5-1, Table 4-5-1, and Table 4-5-2 respectively. The results of these figure and tables can be summarized as:

19% ≤ Performance Impact ≤ 41% => Median Performance Impact = 30%

33% ≤ Resources Impact ≤ 33% => Median Resources Impact = 33%

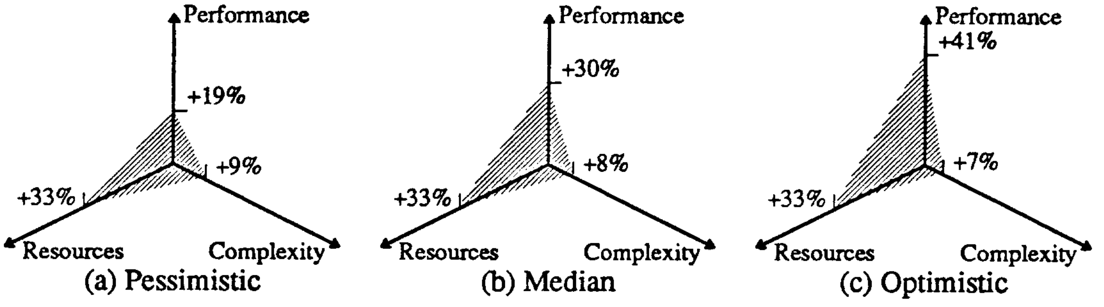


Figure 4-5-2 Three Different Views of the Tradeoffs

(a) shows the pessimistic view which uses the smallest performance impact and the biggest resources and complexity impact. (b) shows the median view which uses the median performance impact and median resources and complexity impact. (c) shows the optimistic view which uses the biggest performance impact and the smallest resources and complexity impact.

$$7\% \leq \text{Complexity Impact} \leq 9\% \quad \Rightarrow \quad \text{Median Complexity Impact} = 8\%$$

These ranges of impact numbers can be used to formulate the optimistic, median, and pessimistic views of the performance, resources, and complexity tradeoffs. These three views are shown graphically in Figure 4-5-2. Notice that although the Instruction Unit consumes a large amount of resources, its impact on complexity is relatively small.

4.6. Multiprocessing Support Evaluation

The SPUR CPU supports multiprocessing by communicating with the Cache Controller Chip [Woo86] via a specialized coprocessor interface [WEG87]. The performance model developed in Section 4.1 cannot be used here because it is developed for uniprocessor performance analysis only. Uniprocessor's performance is a function of cycle time, instruction count, and average number of cycles per instruction. These factors are not as significant in a multiprocessor environment where many processors work in parallel. Consequently, multiprocessor's performance depends more on the number of processors [Kat85], bus traffic [Gib87], and cache behavior [EgK88]. Since all these factors are outside the scope of this thesis, the multiprocessing support's impact on performance will not be studied here. Instead, we will concentrate on multiprocessing support's impact on resources in Section 4.6.1, and complexity in Section 4.6.2. The results is summarized in Section 4.6.3.

4.6.1. Multiprocessing Support–Impact on Resources

	Instructions		Cache Controller Interfaces		Total
Control PLA Outputs	6/54	11%	0/54	0%	11%
Control PLA Products	2/84	2%	0/84	0%	2%
Chip Area (mm x mm)	0/57	0%	2.2/57	4%	4%
Transistors (x 1000)	0/115	0%	0.8/115	1%	1%
Number of Signal Pins	0/156	0%	15/156	10%	10%

Table 4-6-1 Resources Metrics for Multiprocessing Support

Each column lists the absolute and percentage impact of each sub-feature on the different resources metrics. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU. The total impact of these sub-features, that is the impact due to multiprocessing support, is summarized in the right most column.

The multiprocessing support requires seven load instructions, three store instructions, and a Cache Controller Interface. Although all load or store instructions are alike internally, the CPU must request different cache operations for different load or store instructions via the cache controller interface. The impact of multiprocessing support on resources are quantified into several resources metrics in Table 4-6-1. The instruction's impact are mainly on the number of control PLA outputs. On the other hand, the Cache Controller Interface's major impact is on the number of signal pins. The number of transistors and active chip area consumed by the multiprocessing support are relatively small.

4.6.2. Multiprocessing Support–Impact on Complexity

The complexity due to the multiprocessing support can be quantified by the simulation effort. The cooperation of the SPUR CPU and the Cache Controller is simulated in both the behavioral and switch level. The diagnostics we used for switch level simulation are:

(1) **cc-epromLd32** – This diagnostic takes 260 cycles to verify that the SPUR CPU can work together with the Cache Controller to load 32-bit data from the external word.

(2) **cc-ldSt40** – This diagnostic takes 1184 cycles to verify that the SPUR CPU can work together with the Cache Controller to load and store 40-bit data from and to the external world.

(3) **cc-short_hit** – This diagnostic takes 2131 cycles to verify that the SPUR CPU can work together with the Cache Controller to handle a cache hit situation.

(4) **cc-shortMissPF** – This diagnostic takes 1392 cycles to verify that the SPUR CPU can work together with the Cache Controller to handle a cache miss that results in a page fault.

(5) **cc-ptetMissPF** – This diagnostic takes 1585 cycles to verify that the SPUR CPU can work together with the Cache Controller to handle a cache miss that results in a page fault

	Absolute Impact	Percentage Impact		
Cycles of Diagnostics	13,875	13,875/55,516	=	25%
Man-Month of Effort	1.0	1.0/3.5	=	29%

Table 4-6-2 Complexity Metrics for Multiprocessing Support
The first column lists the absolute impact on the complexity metrics due to hardware tag checking. The second column lists the percentage impact. The percentage impact is calculated by dividing the absolute impact by the total value of that metric in the SPUR CPU.

and the page table entry is not in the cache.

(6) **cc-short_si** – This diagnostic takes 1805 cycles to verify that the SPUR CPU can work together with the Cache Controller to handle an interrupt.

(7) **catch-fault** – This diagnostic takes 5518 cycles to verify that the SPUR CPU can work together with the Cache Controller to handle page faults that caused by store operations.

The total number of cycles and the man-month of simulation effort are two complexity metrics we can extracted from this set of diagnostics. The metrics are summarized in Table 4-6-2. Notice that the increase in complexity due to multiprocessing support is relatively big.

4.6.3. Multiprocessing Support–Impact Summary

The impact of the on-chip instruction cache on resources allocation and complexity are illustrated in Table 4-6-1 and Table 4-6-2 respectively. The results of these figure and tables can be summarized as:

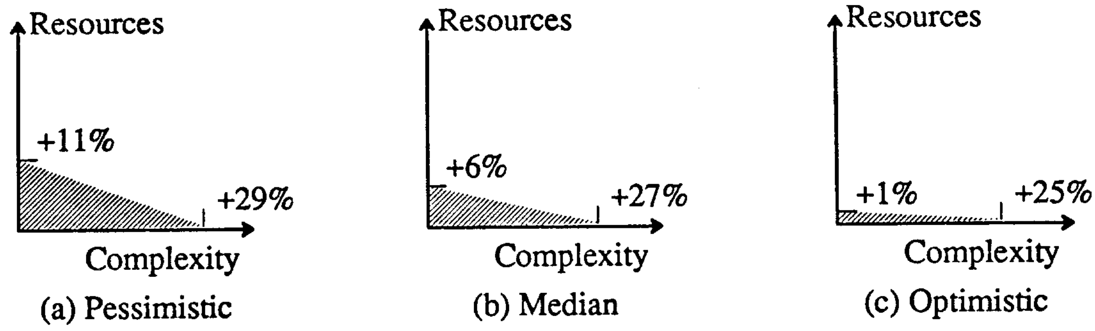


Figure 4-6-1 Three Different Views of the Tradeoffs

(a) shows the pessimistic view which causes the biggest resources and complexity impact. (b) shows the median view which causes the median resources and complexity impact. (c) shows the optimistic view which causes the smallest resources and complexity impact.

$$1\% \leq \text{Resources Impact} \leq 11\% \quad => \quad \text{Median Resources Impact} = 6\%$$

$$25\% \leq \text{Complexity Impact} \leq 29\% \quad => \quad \text{Median Complexity Impact} = 27\%$$

These ranges of impact numbers can be used to formulate the optimistic, median, and pessimistic views of the resources and complexity impact. These three views are shown graphically in Figure 4-6-2. Notice that while the multiprocessing support does not consume a large amount of resources, its impact on complexity is large.

4.7. Microarchitectural Evaluation Summary

The performance model introduced in Section 4.1 allows us to study performance quantitatively. Section 4.7.1 summarizes how this simple model was used to study the performance improvement caused by different SPUR CPU features. Section 4.7.2 gives a quantitative argument for keeping the cycle time and average number of cycles per instruction low. Finally, Section 4.7.3 discusses a systematic approach to the performance, resources, and complexity tradeoffs.

4.7.1. Versatility of the Performance Model

The performance model introduced in Section 4.1 allows us to study different microarchitectural features' impact on performance by comparing the performance of the advanced microarchitecture with that feature against a stripped down microarchitecture without that feature. This performance model has only five parameters:

(1) M_i: The number of instruction it takes to perform a certain function with the architectural feature under consideration.

(2) M_o: The number of instruction it takes to perform a certain function without the architectural feature under consideration.

(3) F_i: The frequency of the architectural feature being used.

(4) $\frac{T_o}{T_i}$: The cycle time ratio.

(5) $\frac{C_o}{C_i}$: The average number of cycles per instruction ratio.

This simple performance model is versatile enough to let us evaluate the performance impact of different SPUR CPU features although they affect the performance very differently. All we have to do is find the proper values or, in more complex cases, find the proper expression for the five parameters. For example, in Section 4.4 the number of instructions it takes to execute a LOAD (M_i) is expressed in terms of the proportion of the LOAD delay slot being filled by useful instruction PRO_{fill}. Below is a summary of how the different features of the SPUR CPU affect these five parameters. In all cases, I have used the subscript "i" for the SPUR CPU (T_i, C_i, M_i) and subscript "o" for the stripped down CPU (T_o, C_o, M_o).

LISP Support:

$M_i = 1$. $M_o > 1$ and depends on the number of instructions it takes to do the explicit tag checking. F_i is the frequency of the instructions that requires tag checking. Finally, $\frac{C_o}{C_i}$ and $\frac{T_o}{T_i}$ are not affected directly.

FPU Support:

$M_i = 1$. $M_o > 1$ and depends on the number of instructions it takes to emulate the floating point operations. F_i is the frequency of the floating point operations. $\frac{C_o}{C_i} < 1$ because even with the FPU coprocessor, the number of cycles it takes the CPU-FPU combination to execute a floating point operation is still bigger than the average number of cycles per CPU instruction. Finally, $\frac{T_o}{T_i}$ is not affected directly.

Extra Pipeline Stage:

$M_o = 1$. $M_i > 1$ and depends on the proportion of the LOAD delay slot being filled by useful instruction. F_i is the frequency of the LOAD. $\frac{C_o}{C_i} > 1$ because the stripped down CPU must suspend the pipeline for one cycle whenever LOAD is executed. Finally, $\frac{T_o}{T_i}$ is not affected directly.

On-Chip Instruction Cache:

$M_o = 1$. $M_i > 1$ and depends on the proportion of the LOAD delay slot being filled by useful instruction. F_i is the frequency of the LOAD and STORE. $\frac{C_o}{C_i} > 1$ because the stripped down CPU must suspend the pipeline for one cycle whenever LOAD and STORE is executed. Finally, $\frac{T_o}{T_i} > 1$ because the on-chip instruction cache eliminate the need for going off-chip to fetch every instruction.

I must point out that the performance improvement due to LISP and FPU support are very program dependent. The FPU support will not benefit any program that does not contain any floating point operation. Similarly, the LISP support feature will not benefit any program that is not written in LISP.

4.7.2. The Need for Speed

Figure 4-5-1(c) shows that the performance improvement predicted by the best argument for having on-chip instruction cache increases 47% (from –6% to 41%) while the worst argument increases only 40% (from –21% to 19%) when the cycle time is improved by 50%. In general, the best argument for having a particular feature improves faster than the worst argument when the cycle time, the average number of cycles per instruction, or both are improved by that feature. From the opposite viewpoint, the best argument for having a particular feature degrades faster than the worst argument when the cycle time, or the average number of cycles per instruction or both are degraded by that feature. This is illustrated in Figure 4-2-1(b), Figure 4-3-1(c), and Figure 4-4-2(c). For example, in Figure 4-2-1(b) when the p factor (the product of the cycle time ratio and the average number of cycles per cycle ratio) is reduced from 1.0 to 0.9, the best argument for having tag checking drops 30% (from 204% to 174%), the median argument only drops

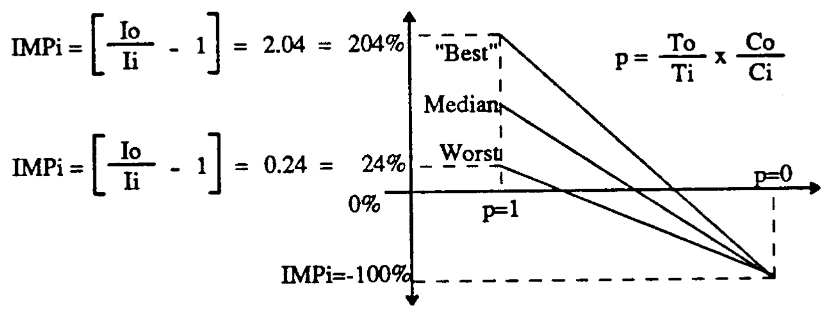


Figure 4-7-1 The Effect of Degrading P Factor

This is a simplified version of Figure 4-2-1(b) which shows how the performance improvement due to hardware tag checking is affected by degrading cycle time or average number of cycles per instruction or both. The *p* factor is defined as the product of the cycle time ratio and the average number of cycles per cycle ratio. Notice that as *p* decreases, the performance improvement predicted by the best argument (top line) drops off faster than the median argument (middle) which drops faster than the worst argument (bottom). This make sense mathematically because all these lines must merge at minus 100% when $p = 0$.

19% (from 92% to 73%), while the worst argument drops even less, only 13% (from 24% to 11%). As illustrated in Figure 4-7-1, this make sense mathematically. However as engineers, we were taught not to believe the mathematics unless it makes sense physically!

In order to understand why the top line drops faster than the bottom line (Figure 4-7-1), we have to look at the case $p = 1$. When $p = 1$, we are ignoring the effects of the new feature on cycle time and the average number of cycles per instruction. A big performance improvement at $p = 1$ (top line) means adding that feature can reduce the number of instructions (I) by a large amount. Seen another way, it means getting rid of the feature will increase the number of instructions by a large amount. In our LISP example Figure 4-7-1(b), the best argument for having tag checking (top line) therefore predicts an increase of 204 instructions for every 100 instructions if tag checking is removed. The worst argument predicts only an increase of 24 extra instructions for every 100 instruction.

At p=1, the gap between the best and worst arguments represents this difference in number of instructions (204 – 24 = 180) because we are neglecting hardware tag checking's effect on cycle time and average number of cycles per instruction. On the other hand, at $p<1$, the gap between the best and worst arguments represents the time the stripped down CPU takes to execute the extra 180 instructions. If by getting rid of hardware tag checking we can reduce the cycle time or the average number of cycles per instruction or both (smaller p), the time it takes the stripped down CPU to execute the extra instructions is reduced. Consequently as we move towards the p=0 point, the gap between the best and worst arguments gets smaller and smaller. Finally, at p=0, the stripped down CPU's cycle time and average number of cycles per instruction is so much faster than the SPUR CPU that the time to execute the extra instructions is negligible–the gap disappears. As a matter of fact, at p=0, the time the stripped down CPU takes to execute the benchmark is practically zero compare to the execution time of the SPUR CPU. The performance improvement is therefore −100%.

In my opinion, this is a good quantitative argument for keeping the cycle time and average number of cycles per instruction low at all cost because they benefit all instructions–not just one particular instruction. However, I must also point out that when I say reduce the cycle time, it does not mean just reduce the CPU cycle time. The environment surrounding the CPU–such as the memory system and the I/O devices–must also speed up. Otherwise the extra wait states will lower the performance improvement. This, of course, is one version of Amdahl's law [Amd67] which states that the speed of any computation is limited by its slowest part.

4.7.3. Performance Resources and Complexity Tradeoffs

Resources and complexity are two separate ways we can pay for performance. As we can see from our analysis, resources and complexity are quite independent. For example, the on-chip instruction cache has a large impact on resources but only a small impact on complexity. On the other hand, the multiprocessing support has large impact on complexity but only a small impact on resources. The microarchitect has many options to achieve the desired performance by

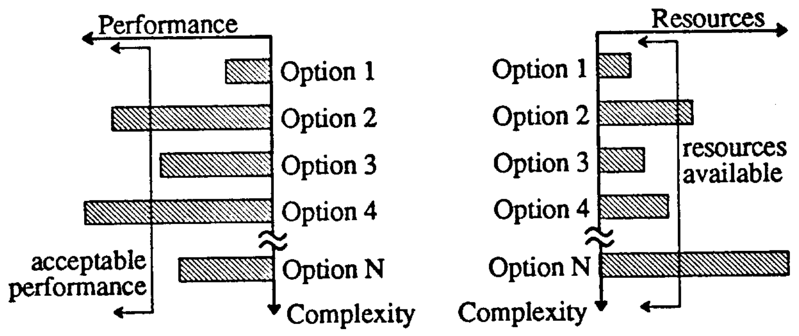


Figure 4-7-2 Performance Resources and Complexity Tradeoffs

The options, which correspond to different features that can be included in the microarchitecture, are placed in increasing complexity on the vertical axis. The performance and resources needed for these options are plotted on the horizontal axes. Each module has its own minimum performance requirement which is a direct result of the overall performance goal. This performance requirement place an "acceptable performance" bound on the Performance axis.

selecting different features for the microarchitecture. All options involve tradeoffs between performance, resources, and complexity.

This type of tradeoff is shown graphically in Figure 4-7-2. Since we are considering what features to be included in the CPU, Option 1 could be a basic CPU that has minimum features. It is simple in complexity, low cost in resources, and low performance. Option 2 could be a CPU similar to Option 1 but the microarchitect uses resources to pay for more performance by adding a very large instruction cache. This large instruction cache increases the resources a lot but only increases the complexity slightly. Option 3 could be a CPU similar to Option 1 but the microarchitect uses complexity to pay for more performance by using a very long pipeline. This long pipeline increases the complexity a lot but only increases the resources slightly. Option 4 could be a combination of Option 2 and Option 3. The microarchitect uses both resources and complexity to pay for more performance by using a moderate size instruction cache and a moderate length pipeline. The interaction between the pipeline and the instruction cache further increases the complexity. However, since the cache is much smaller than the cache in Option 2, the resources it consumes is less. Finally, Option N could be a machine that is so complex that its performance is less than a simple machine that uses far fewer resources.

The performance requirement for the CPU places an "acceptable performance" bound on the performance axis (Figure 4-7-2). Given this performance requirement, the microarchitect must include enough features in the microarchitecture such that the performance requirement is met while at the same time stay within the resources and complexity constraints. Here is the recommended approach:

(1) Make an educated guess on how many resources you are willing to spend or are available to the CPU. This places a "resource available" bound on resources axis in Figure 4-7-2.

(2) Within this bound, pick the simplest option available.

(3) If this option's performance is within the acceptable range, then mission accomplished. Otherwise, go to Step 4.

(4) If there are any other options within the resource bound, pick the next more complex option and go back to Step 3. Otherwise go to Step 5.

(5) If possible, go to Step 1 and increase the resources available bound. Otherwise, you may need to reduce the performance expectation.

Using Figure 4-7-2 as an example, Step 2 of this procedure will pick Option 1. However, in Step 3, we will find out Option 1's performance is below the acceptable range. In Step 4, Option 2 will not be considered because it is beyond the resources available bound. Option 3 will be chosen because it is less complex than Option 4. Unfortunately, in Step 3, we will again find out Option 3's performance is not acceptable. This will lead us back to Step 4 and select Option 4. This time, when we get back to Step 3, we will find out Option 4's performance is acceptable.

This kind of tradeoff should always be in a microarchitect's mind and I see absolutely no reason why we can build CAD tools to place and route million of gates but cannot build tools to help designer to make tradeoffs in this fashion. In this chapter, the performance resources and complexity tradeoffs are done on the complete CPU. In chapter 5, I will show how this tradeoffs can be extended into lower level modules that form the CPU. There are two things I want to point out:

(1) We always pick the simplest option because a complex option requires the most expensive resource–the human designer. The simplest solution can also make use of the newest technology. When technology is improving fast it can negate the performance advantage of complex solution that takes longer to implement and debug.

(2) We do not pick the highest performance option because any option within the performance specification is as acceptable as the highest performance option. The reason is that the increase in CPU performance alone will not improve the system performance drastically unless you speed up all components of the system. This is another version of Amdahl's law [Amd67].

In this chapter, we performed the performance resources and complexity tradeoffs analysis *after* the SPUR CPU was built. This is educational, but may be too late unless we are willing to build multiple prototype to correct our mistakes. Next chapter, I will show a systematic approach that will perform this type of analysis earlier in the design process.

4.8. REFERENCES

[Amd67] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *Proceedings AFIPS 1967 Spring Joint Computer Conference 30*, Atlantic City, New Jersy, April, 1967.

[Bos88] B. K. Bose, *VLSI Design Techniques for Floating-Point Computation*, Doctoral Dissertation, Computer Science Division, EECS Department, University of California, Berkeley, November 1988.

[Dun86] R. R. Duncombe, *The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor*, Master Report, EECS Department, University of California, Berkeley, CA 94720, August, 1986.

[EgK88] S. Eggers and R. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *The 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 30-June 2, 1988.

[Gib87] G. Gibson, "Estimating Performance of Single Bus, Shared Memory Multiprocessors", Report No. UCB/Computer Science Dpt. 87/355, Computer Science Division, EECS Department, University of California, Berkeley, May 1987.

[HaK86] P. Hansen and S. Kong, "SPUR Coprocessor Interface Description", Report No. UCB/Computer Science Dpt. 87/308, Computer Science Division, EECS Department, University of California, Berkeley, October 1986.

[Han88] P. M. Hansen, *Coprocessor Architectures for VLSI*, Doctoral Dissertation, Computer Science Division, EECS Department, University of California, Berkeley, November 1988.

[Hen85] J. Hennessy, "VLSI RISC Processor", *VLSI Systems Design VI*, 10 (October 1985).

[Hil87a] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Dissertation, Computer Science Division, EECS Department University of California, Berkeley, Fall 1987.

[Hil87b] M. Hill, Private Communication Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, December, 1987.

[Hil88] M. Hill, "A Case for Direct-Mapped Caches", *Computer 21*, 12 (December 1988).

[Kat85] R. Katz, et al., "Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses", Report No. UCB/Computer Science Dpt. 85/221, Computer Science Division, EECS Department, University of California, Berkeley, January 1985.

[Kog81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981.

[Pat89] D. Patterson, Private Communication Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, January, 1989.

[Sip82] T. N. Sippel, *Floating RISC: Implementation and Analysis of Floating Point on RISC I*, Master Report, Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, August, 1982.

[StH88] P. Steenkiste and J. Hennessy, "Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization", *Computer 21*, 7 (July 1988).

[Tay89] G. Taylor, Private Communication Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, January, 1989.

[Tay86] G. Taylor et al., *Evaluation of the SPUR Lisp Architecture*, The 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 2-5, 1986.

[WEG87] D. Wood, S. Eggers and G. Gibson, "SPUR Memory System Architecture", Report No. UCB/Computer Science Dpt. 87/394, Computer Science Division, EECS Department, University of California, Berkeley, December 1987.

[Woo86] D. A. Wood et al., "An In-Cache Address Translation Mechanism", *The 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 2-5, 1986.

[Zor89] B. Zorn, Private Communication Computer Science Division, EECS Department, University of California, Berkeley, CA 94720, January, 1989.

Chapter 5

A SYSTEMATIC APPROACH TO MICROARCHITECTURAL DESIGN

I shall never believe the God plays dice with the world.

Albert Einstein, 1947

The goal of this thesis, as stated in Chapter 1, is to provide a quantitative way to evaluate microarchitectures and a systematic way to design them. In Chapter 4, I have used the SPUR CPU as an example to show how microarchitecture can be evaluated quantitatively. In this chapter, I show how to approach the microarchitectural design problem systematically.

5.1. The Microarchitectural Design Problem

This section discusses the microarchitectural design problem. Section 5.1.1 formally defines the term microarchitecture and then the phrase "microarchitectural design." Section 5.1.2 introduces a set of important issues that are important to microarchitectural design. Section 5.1.3 shows a systematic approach to these microarchitectural issues.

5.1.1. Microarchitectural Design–The Definition

The term microarchitecture was defined informally in Chapter 1 as the specification of how the macroarchitecture is implemented in a given technology. More formally, the term microarchitecture can be defined with respect to Gajski's tripartite representation [Gaj85] as all the information the designer knows about the design at its microarchitectural level. As shown in Figure 5-1-1, the microarchitectural level is one of the five possible design levels in Gajski's tripartite

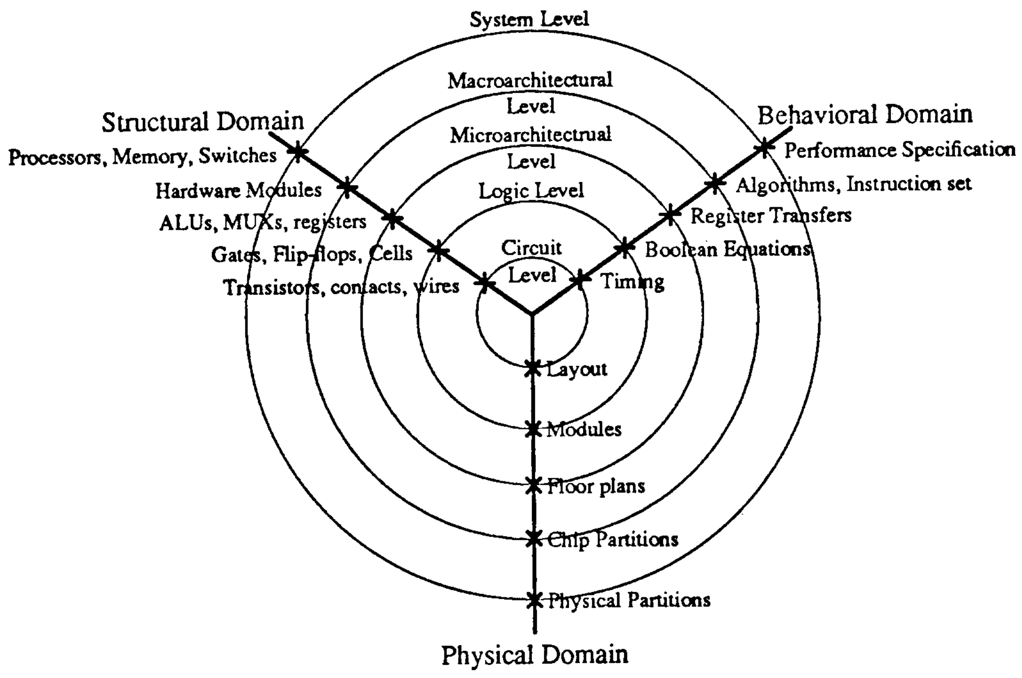


Figure 5-1-1 The Tripartite Representation of a Design

In Gajski tripartite representation, a design can be described in three separate domains: behavioral, structural, and physical. Each domain is represented by one of the three axes that form the Y chart. Within each domain, there are five design levels. I have added concentric circles to Gajski original tripartite representation to show the five design levels graphically. The design levels can be viewed as different levels of abstraction and each design level represents all the informations known about the design at some point in the design process.

representation. Since each design level is shown in Figure 5-1-1 to span all three domains, the term microarchitectural design refer to the step at which the designer specifies all the microarchitectural level features in the behavioral, structural, and physical domains.

The microarchitectural design step of the SPUR CPU is shown as an example in Figure 5-1-2. This figure shows the SPUR CPU design processing discussed in Section 3.2 (Figure 3-2-1) with respect to the tripartite representation. This representation idealizes the SPUR CPU design process into a purely sequential process that starts in the performance specification in the behavioral domain and spirals toward the final product–the layout at the physical domain. This is

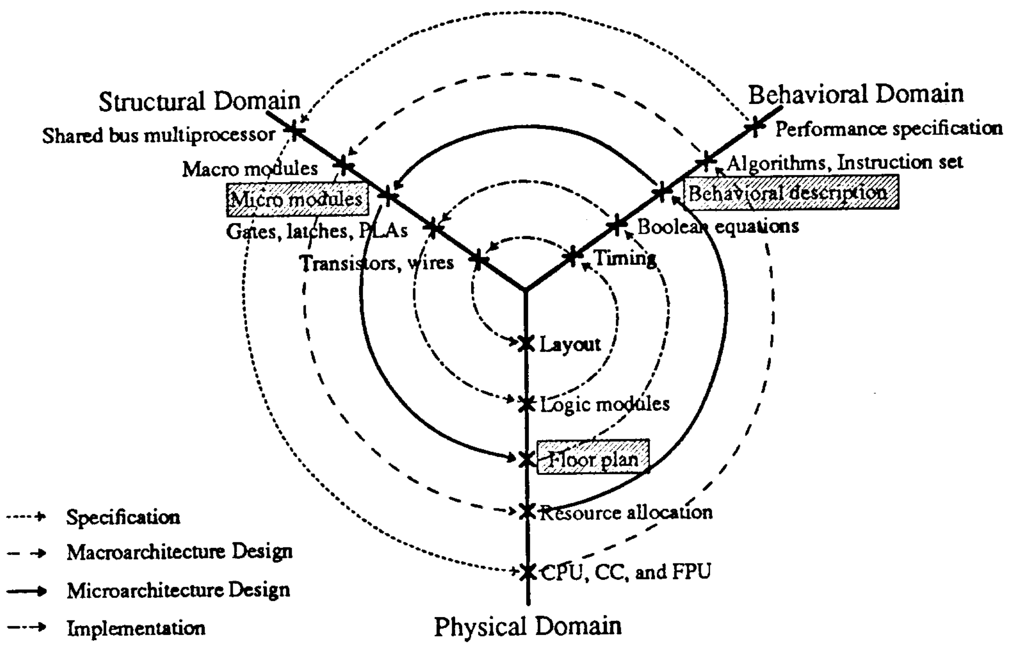


Figure 5-1-2 The Tripartite Representation of the SPUR CPU Design Process

The four major steps in the SPUR design process–Specification, Macroarchitectural Design, Microarchitectural Design, and Implementation–are shown here with respect to the tripartite representation. The three products of the microarchitectural design step are the behavioral description in the behavioral domain, a set of micro-modules (specifies in block diagrams) in the structural domain, and a floor plan in the physical domain. The ALU and Shifter are examples of micro-modules. On the other hand, the Operand Supplier and Functional Unit are examples of macro-modules.

an idealized picture because, in practice, the sub-steps and products within the four major steps cannot be defined as clearly as shown.

5.1.2. Microarchitectural Design Issues

The most general approach for handling the microarchitectural design problem involves two steps:

(1) Design the datapath, and

(2) Design a controller that controls the datapath.

This procedure, however, is so general that telling it to an inexperienced designer is about as helpful as telling someone who is afraid of flying that the only danger in aviation is hitting the ground. There are just too many tasks in these two steps. Fortunately, I can give more direct advice: All the tasks involve making decisions concerning certain issues. The designer can approach these two complex steps systematically by asking himself what the important issues are and finding solutions to them. Based on the SPUR CPU design experience, I think these are the six important issues affecting microarchitectural design:

(1) Off-chip Communication

Off-chip communication has always been a bottleneck due to the limited number of pins and it is worse for output ports because of power considerations [PaS80]. This problem is more severe for modern microprocessors which communicate not only with memory but also have to communicate with coprocessors, memory management units, and other microprocessors in a multiprocessor system.

(2) Pipeline and Clocking

A longer pipeline usually leads to a shorter cycle time. The performance gain from a shorter cycle time, however, may be lost due to the increased cost in branches and data hazards. Alternatively, a shorter pipeline is easier to control and the penalties are smaller for branches and data hazards. But a shorter pipeline usually requires more clock phases per cycle, leading to a more complicated clock distribution network, a more severe clock skew problem, and ultimately a longer cycle time.

(3) Micro-Modules Selection

The microarchitect must select a set of micro-modules to implement the instruction set and other macroarchitectural features specified by the macroarchitect.

(4) Resources Allocation

On-chip storage trade-offs such as the size of register file versus the size of the instruction cache is a very interesting problem by itself [GoH86]. This problem can be generalized to include the process of allocating resources to the set of micro-modules.

(5) On-chip Interaction

A clean microarchitectural design can restrict most on-chip interactions to internal busses. Certain functions such as trap handling, inherently involve many on-chip components and interaction is unavoidable. Certain instructions also have the tendency to involve many on-chip components and these instructions should be avoided.

(6) Floor Planning

The microarchitecture must eventually be implemented on the limited area of a silicon chip. The microarchitect must decide how to place the set of macro-modules according to their sizes, aspect ratios, and connections between the macro-modules.

I call these issues the microarchitectural issues. They can be grouped into three groups with respect to the three domains in Gajski tripartite representation.

Behavioral Issues

Off-chip communication, and pipeline and clocking are behavioral issues.

Structural Issues

Micro-modules selection, on-chip interaction, and resources allocation are structural issues.

Physical Issue

Floor planning is a physical issue.

A microarchitect must answer some tough questions concerning these issues when he designs the datapath and the controller. His decisions on these issues will have a direct effect on the performance, resources, and complexity tradeoffs.

5.1.3. A Systematic Approach to Microarchitectural Issues

A systematic approach to microarchitectural design must begin with a systematic approach to the microarchitectural issues. Ideally, the microarchitect would like to tackle one microarchitectural issue at a time. Unfortunately, all microarchitectural issues are interrelated. Decisions concerning one issue usually lead to (or restrict) decisions concerning the others issues. A systematic approach to these issues must take these interrelations into account. Here is a systematic approach I recommended:

Before making any important decisions concerning any microarchitectural issue or issues, the microarchitect should:

(1) List all the unanswered questions concerning each issue.

(2) Construct a model that can isolate the issue or issues.

(3) Use the model to conduct quantitative experiments to answer the questions.

The modeling language does not have to be a hardware description language. The goal is to isolate certain aspects of the microarchitectural design at a time. The model the microarchitect constructs and the experiments he conducts must take into account the characteristics of the underlying technology and implementation considerations. There are a couple of interesting questions concerning this approach:

- How can the microarchitecture be modeled such that the microarchitect can examine a subset of the microarchitectural issues at a time?
- What are the important parameters to be measured in the experiments such that the microarchitect can make quantitative decisions concerning the issue?

Before I try to answer these questions, I review background studies on systematic approaches to the general design problem.

5.2. Background Studies

Hardware description languages and silicon compilers are two areas to look for ideas that can help us in developing a systematic approach to microarchitectural design, because microarchitecture is just one possible representation of the hardware to be implemented in silicon. Silicon compiler research can be considered as an extension of hardware description language research because once the hardware is described in a machine readable form, you probably want to generate the silicon automatically–the goal of silicon compiler research. Hardware description language research, on the other hand, does not limit its focus to integrated circuit. Hardware description language research must also study hardware at board and system levels. Furthermore, hardware description language must also investigate problems such as formal verification, compilation facilities, access to program libraries, version control, and standardization. Therefore with respect to the scope of their research, silicon compiler research can also be considered as a subset of the hardware description language research.

5.2.1. Hardware Description Languages

Hardware description languages can be defined as computer languages for describing, documenting, simulating, and synthesizing digital systems with the aid of a computer [Su77]. According to Chu [Chu74], describing digital system in computer language is nothing new:

> The use of computer languages to describe digital system designs can be traced back to Shannon's work on switching circuit in 1939, Aiken's work on switching theory at Harvard in the 1940's, the logic diagrams at MIT and the National Bureau of Standards in the late 1940's, the flipflop equations in the 1950's, and the register languages in the 1960's.
>
> Yaohan Chu, Why Do We Need Computer Hardware Description Languages?
> Computer, December 1974, Page 18

Hardware description languages' application, however, was not wide spread until the early 1970s when researchers started using them as documentation, simulation, and teaching tools. By the late 1970s [Su77] hardware description languages were well established as documentation and simulation tools. There was also an attempt to use hardware description language to describe a new invention at that time–the microprocessor [Lip75]. Two interesting research topics in hardware description languages emerged in the late 1970s are:

Digital System Analysis and Evaluation

The goal was to find ways to evaluate the effectiveness and prove the correctness of a proposed digital system by simply analyzing the system's description in a hardware description language. This would eliminate the need for time consuming simulation.

Structured Design of Digital System

A digital system described in a well designed hardware description language should be easy to partition into modules that are easy to build and design. This will encourage designer to use relatively independent modules.

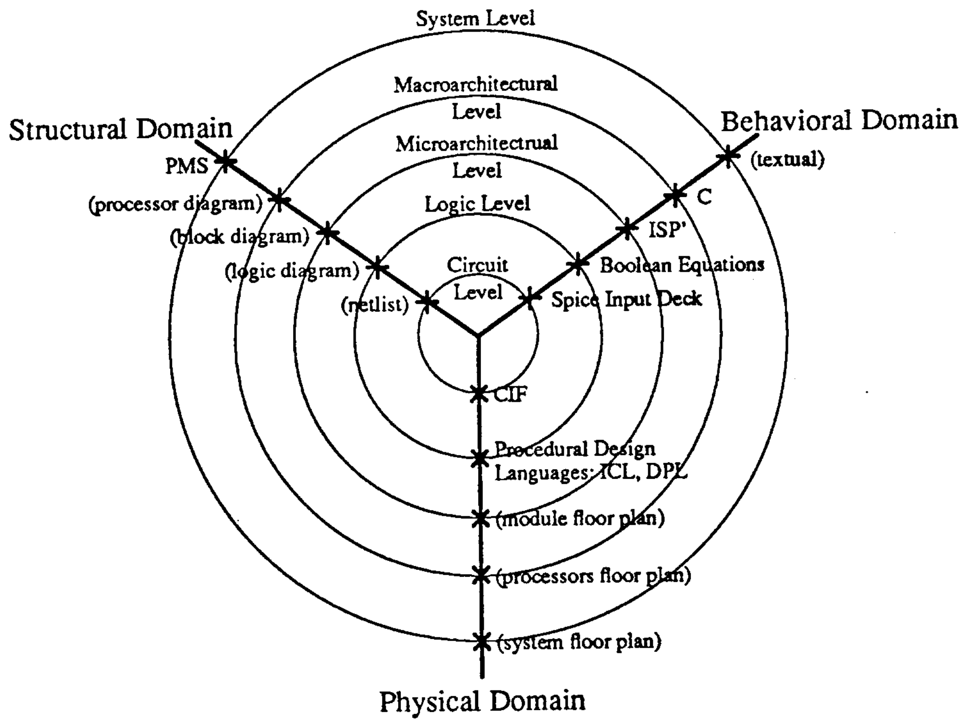


Figure 5-2-1 Classification of Hardware Description Languages

At each domain, there are certain design levels where no formal hardware description language exists. These levels are usually described informally and the informal ways to describe them are shown inside parenthesis. The behavioral domain is the domain most covered by hardware description languages. As a matter of fact, most so call hardware description languages are languages that describe the microarchitectural level of the the behavioral domain.

These two topics remain important driving forces in hardware description language research in the 1980s. Furthermore, the trend of the 1980s is more formal applications of hardware description languages and synthesis of hardware from formal machine description. A standardized universal hardware description language was a goal since the early 1970s but was never realized. I think the goal of having a standardized universal hardware description language is hard to achieve because there are different requirements for different applications. This is illustrated in Figure 5-2-2, where I have borrowed Gajski's ideas (Figure5-1-1) and classified hardware description languages into five levels and three domains.

In the behavior domain, the system level behavior is usually specified informally in textual form as performance specification. The macroarchitectural level behavior can be modeled by an instruction level simulator written in a high level programming languages such as C. The microarchitectural level behavioral can be described by register transfer languages such as ISP'. Logic level behavior can be described by Boolean equations. Finally, the SPICE input deck can be used to describe the circuit level behavior.

In the physical domain, circuit and logic levels are probably the only levels need to be described in machine readable form. CIF is the most common language for describing physical characteristics at the circuit level–layout (see Figure 5-1-1). On the other hand, procedural design languages such as ICL and DPL are common languages for describing physical characteristics at the logic level. All other levels are usually described informally in floor plans that have different levels of details.

In the structural domain, PMS at the system level is the only well known hardware description language. All other levels are described informally by diagrams and netlists. Here are three reasons for this lack of hardware description languages in the structural domain:

- Most hardware description languages are modeled after high level programming languages. They are good at capturing a design's behavior, but contain little structural information.

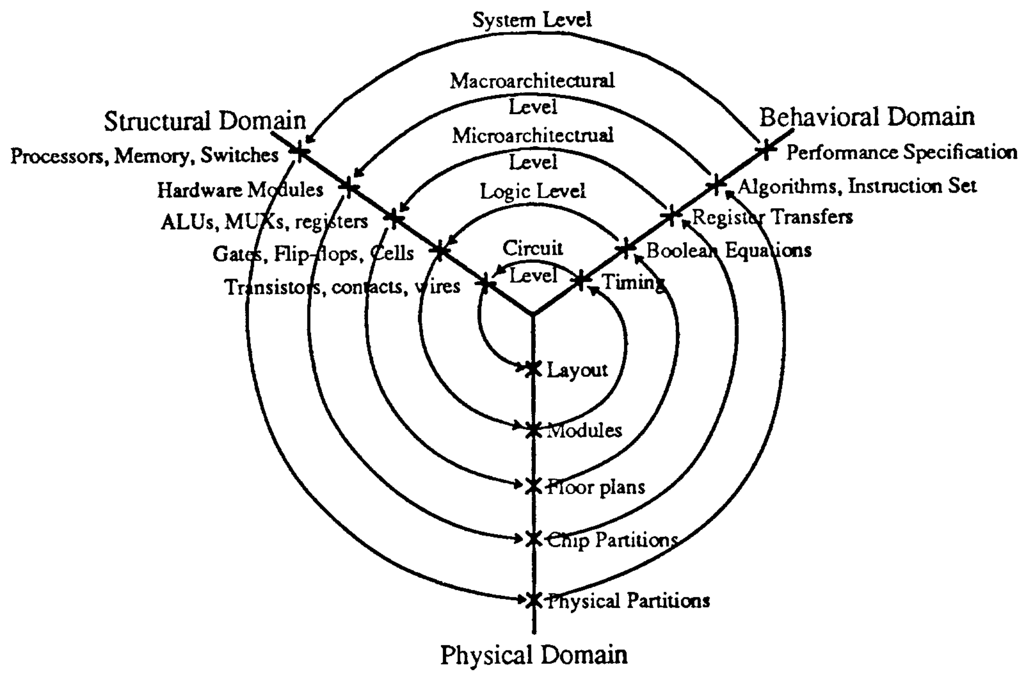


Figure 5-2-2 Pure Top Down Design Methodology

The steps of the pure top down design methodology form an inward spiral. For simplicity, feedback paths between each step are not shown. However, these feedback paths are the reasons why iteration is necessary. The pure bottom up design methodology is exactly opposite–an outward spiral. The reader can get a mental picture of the pure bottom up design methodology by reversing the arrow heads.

- Diagrams consisting of black boxes and connections are natural ways to describe structure. These diagrams may have different levels of detail at different design levels but are similar structurally. Therefore it is conceivable to use PMS for all levels in the structural domain.

- Most current custom design methodologies require the designer to work in the structural domain. The graphical CAD tools they use then capture the structural information implicitly and eliminate the need for explicit description of the structure using formal description languages. For example, the Magic .ext files [SMH85] can be considered as an implicit hardware description languages that describe the netlist.

Human designers prefer to work in the behavioral domain, but their work in the behavioral domain must eventually be transformed to the physical domain. Since work in the physical domain is tedious (but necessary), designers prefer this transformation to be done automatically. This transformation, however, will not be efficient unless important structural information is provided because the structural domain is the bridge between the behavioral and the physical domain. Unfortunately, the lack of hardware description languages in the structural domain makes it hard to express structural information in machine readable form. Consequently, designers end up doing more work in the structural and physical domains than they prefer. In order to reduce the manual labor in the structural and physical domains, researchers must pay more attention to structural information representation.

5.2.2. Silicon Compilers

The term silicon compiler was first used by Dave Johannsen of Caltech back in 1981 [Joh81]. The silicon compiler concept was inspired by the pure top-down design methodology. In Figure 5-2-2 above, I have drawn my view of the pure top-down design methodology with respect to Gajski's tripartite representation (Figure 5-1-1). In this view, the steps of the pure top down design methodology form an inward spiral that starts at the system level of the behavioral domain (performance specification) and ends at the circuit level of the physical domain (layout). The ultimate goal of silicon compiler is to carry out this inward spiral automatically.

Early silicon compilers were proposed to carry out the entire synthesis process. In order to simplify this complex task, a target technology was usually assumed and a fixed floor plan was chosen by human designers. In fact, as illustrated in Figure 5-2-3, the highest level input the early primitive silicon compilers could accept was the logic level description in the behavioral domain. Notice that the human designer must carry out all the design steps manually up to the logic level. According to Newton and Sangiovanni-Vincentelli [NeS86], the most important contributions of the early silicon compiler research is the development of the *Procedural Design Languages*–hardware description language in the physical domain (see Figure 5-2-1).

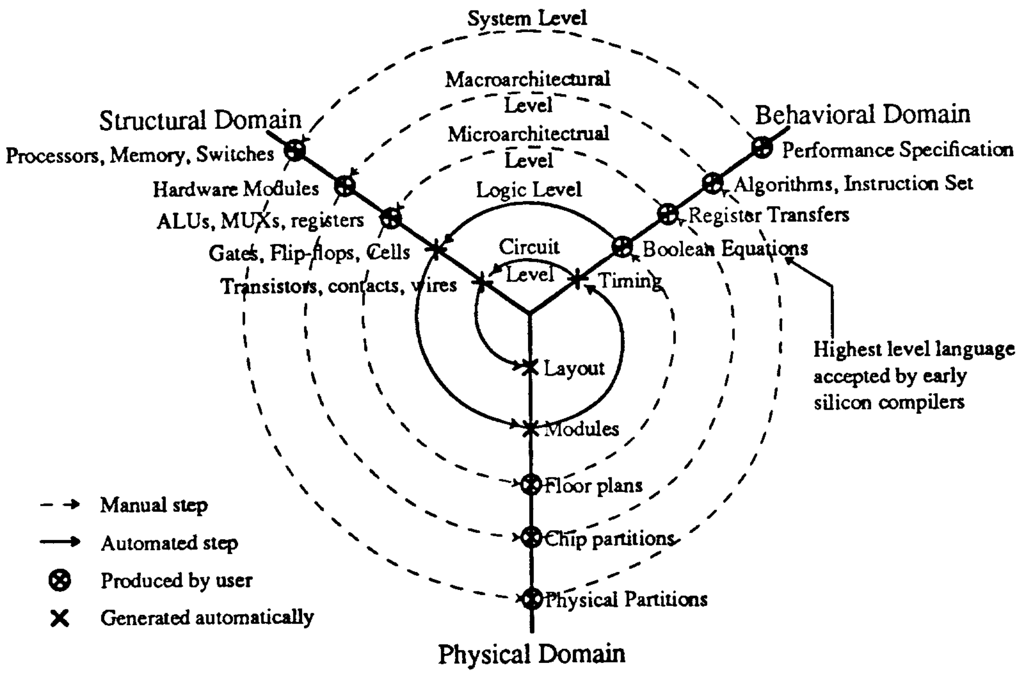


Figure 5-2-3 Primitive Silicon Compilers

Logic level description was the usual input to the early silicon compilers. This description can be in the behavioral domain for the most powerful compilers, in the structural domain for the less powerful compilers, or in the physical domain for the least powerful compilers. In this figure, I have shown the most powerful compilers that can accept inputs from the behavioral domain. In this case, the designer must carry out all the steps and generate all the products manually up to the Boolean equations.

The current goal of most silicon compiler research is no longer to carry out the entire synthesis process by one single program. The current emphasis is to create a "silicon compiler design environment." This environment is illustrated in Figure 5-2-4 in which the synthesis process is divided into stages. CAD tools are then developed to optimize resource allocation and performance at each stage, and to automate the transformation from one stage to the other. The common input of a modern silicon compiler design environment is the register transfer description. The Yorktown Silicon Compiler at IBM [Cam87] is one example. Other more ambitious projects, such as the Design Automation Assistant (DAA) at AT&T Bell Laboratories [Kow85], accept input at the macroarchitectural level of the behavioral domain–algorithmic description.

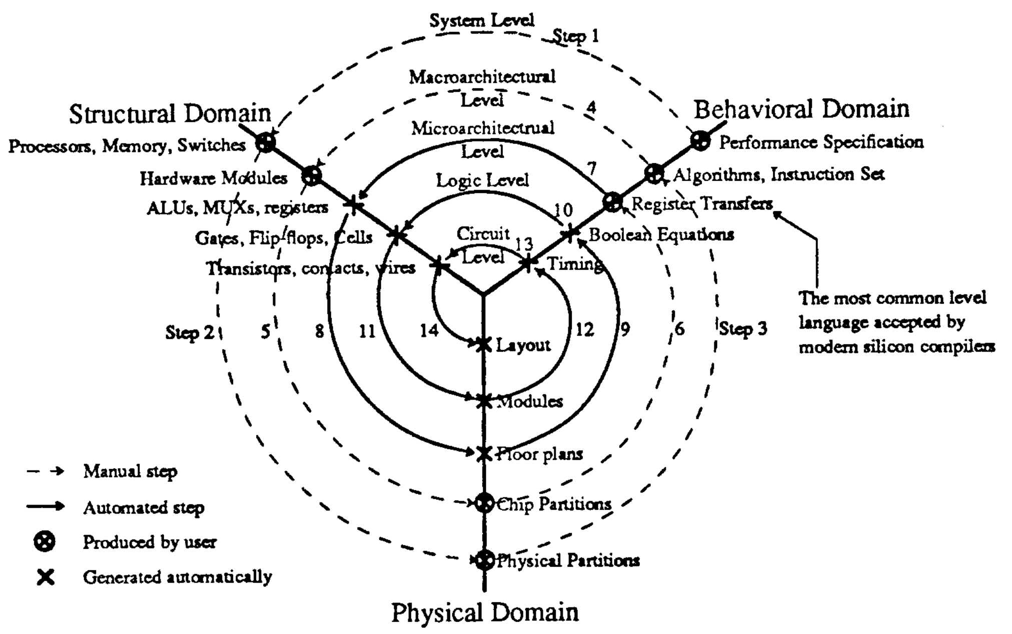


Figure 5-2-4 Modern Silicon Compiler Design Environment

The key words here are "design environment." In the silicon compiler design environment, a set of CAD tools are available to optimize and automate each step of the synthesis process. Most modern silicon compiler design environment can accept inputs at the microarchitectural level of the behavioral domain and generate layout automatically. For some applications such as digital signal processing, and some ambitious silicon compiler projects, they can even accept inputs at the macroarchitectural level of the behavioral domain.

The translation from the algorithmic description to the register transfer description (Step 4, 5, and 6 in Figure 5-2-4) is not a trivial task except for some very specific application such as digital signal processing. For more complex applications such as CPU design, this translation usually requires the use of some knowledge-based expert system programming techniques. Newton and and Sangiovanni-Vincentelli [NeS86] said that in the future, procedural design systems and knowledge-based expert systems are crucial for the development of future synthesis system. They also believe that the major components of a synthesis system are:

(1) Procedural Design and Module Generation–Step 7, Step 8, and Step 9 in Figure 5-2-4.

(2) Logic synthesis–Step 10, Step 11, and Step 12 in Figure 5-2-4.

(3) Physical synthesis–Step 13 and Step 14 in Figure 5-2-4.

5.2.3. Meet in the Middle Approach

There is one major philosophical difference between the goal of this chapter and research in hardware description languages and silicon compilers which is based strongly on computer science theory. Their goal is to introduce a new theory-based approach to the design process and ultimately automate it. This chapter, on the other hand, is based on the design process that was used to create the SPUR CPU. I will look at ways to make this process more systematic and

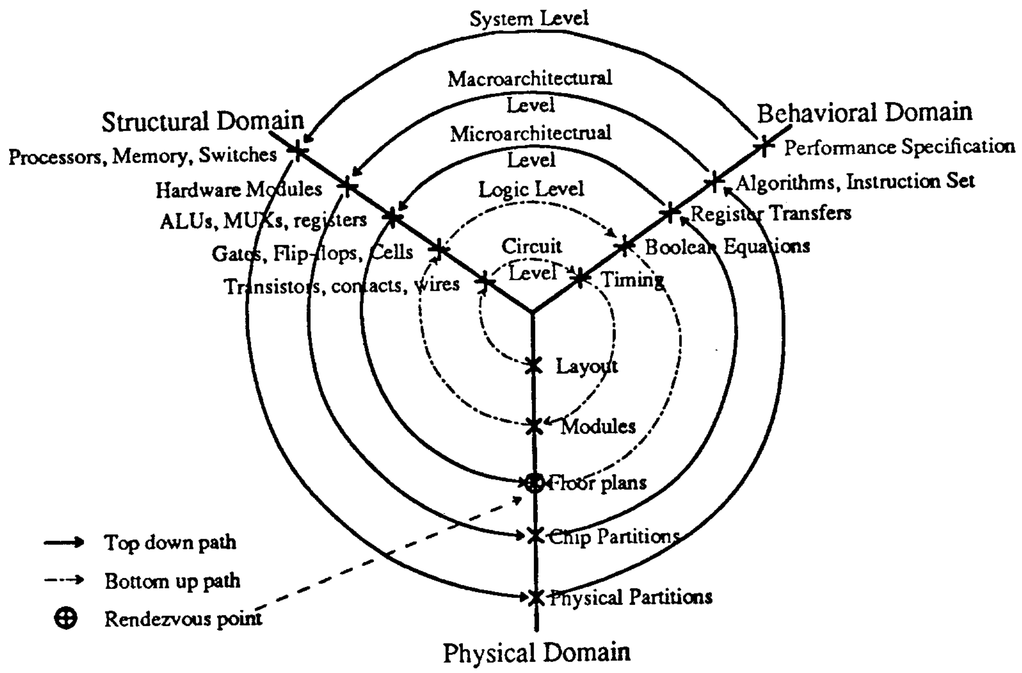


Figure 5-2-5 Meet In The Middle Approach

In the meet in the middle approach, system designers start at the performance specification in the behavioral domain and work their way down. At the same time, logic and circuit designers start at the layout in the physical domain and work their way up. They meet at the microarchitecture level of the physical domain.

efficient based on the lessons I learned. Furthermore, the goal of most hardware description language and silicon compiler research is to develop CAD tools that can automate the top-down design methodology. However, as many custom VLSI chip designers have learned, the top-down design methodology is not as practical as the "meet-in-the-middle" approach. My view of the "meet-in-the-middle" approach is shown in Figure 5-2-5.

Cathedral [DRS86] is a silicon compiler for digital signal processing chips that is based on the meet at the middle approach. The user of Cathedral must provide "structural hints" to the compiler to aid the behavioral to physical compilation. Furthermore, instead of using a module generator to generate the layout, Cathedral's compilation is based on "silicon modules" that are are designed by layout designer and are composed of functional building blocks. For example, if the SPUR CPU is to be compiled by a Cathedral type compilers, the Operand Supplier and the Functional Unit (see Figure 2-3-1) are two of the silicon modules. The set of silicon modules available to Cathedral was carefully restricted. According to the authors, this restriction was the key of Cathedral's success. The authors also believed in order to develop a Cathedral type compiler, one must follow a series of five steps:

(1) Define a wide, but concise class of system design applications.

(2) Define a target architecture and its associated layout style.

(3) Define a design strategy.

(4) Define the behavioral language that models the microarchitecture and silicon modules.

(5) Then and only then develop the CAD tools.

In our SPUR CPU example, the result of Step 1 is the specification of a general purpose CPU with LISP support. In Step 2, the target architecture is a RISC-style processor that does not use microcode, and the layout style is the Mead & Conway style. Step 3, Step 4, and Step 5 are the major steps toward a a systematic approach to the microarchitectural design problem. They are discussed in Section 5.3 in more detail.

5.3. Steps Toward a Systematic Approach to Microarchitectural Design

The major steps toward a systematic approach to microarchitectural design were discussed briefly at the end of Section 5.2. I have added some of my own ideas and restated them as:

(1) Propose a general design strategy.

(2) Develop models that can capture the microarchitecture's behavioral, structural, and physical features.

(3) Build or propose CAD tools that can aid the last two steps.

(4) Refine the general design strategy proposed in Step 1 and iterate again.

The first three steps are based on the discussion at the end of Section 5.2. I added the last step to introduce feedback into the approach. These steps are discussed in Section 5.3.1, Section 5.3.2, Section 5.3.3, and Section 5.3.4, respectively. In order to limit the scope of my research, I will focus the discussion on RISC-style processors that do not use microcode. The discussion are based on the the following observations:

(1) For a RISC-style processor, just looking at the instruction set can tell you a great deal about the microarchitecture.

(2) The model or models for the microarchitecture must be abstract enough for making high level design decisions and detailed enough for logic specification and simulation.

(3) We must reduce the time we spent for verification in order to improve the efficiency of the design process.

(4) We must document all the important design decisions and the assumptions or facts on which these decisions are based.

5.3.1. The Design Strategy

In the most general term, the microarchitectural design problem can be divided into two tasks: (1) design the datapath, and (2) design a controller that controls the datapath. This is shown

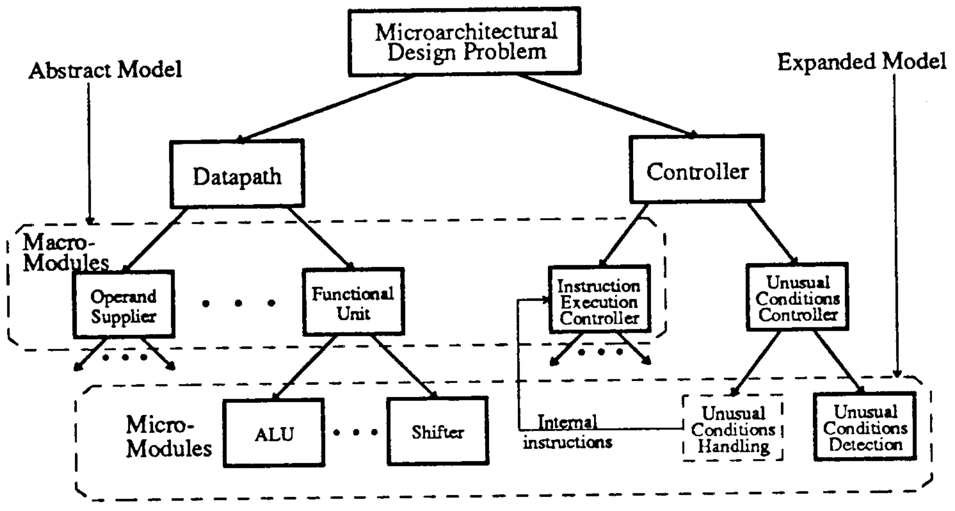

Microarchitectural Design Problem
Abstract Model
Expanded Model
Datapath
Controller
Macro-Modules
Operand Supplier
Functional Unit
Instruction Execution Controller
Unusual Conditions Controller
Micro-Modules
ALU
Shifter
Internal instructions
Unusual Conditions Handling
Unusual Conditions Detection

Figure 5-3-1 Design Strategy

The microarchitectural design problem can be divided into two tasks: datapath design and controller design. The datapath can be further divided into macro-modules and then micro-modules. The controller can be further divided into two parts: one controls instruction execution and another controls unusual conditions. The handling of unusual conditions, however, can be integrated into the part that controls instruction execution via the use of internal instructions (see Section 2.3.3 and Section 2.4.3).

graphically in Figure 5-3-1. The design of the datapath is straightforward. First, the microarchitect must select a set of macro-modules needed to implement the instruction set. Examples of macro-modules in the SPUR CPU are the Operand Supplier and the Functional Unit. After the microarchitect is satisfied with the behavior of these macro-modules, he can expand the macro-modules into micro-modules. Examples of micro-modules in the SPUR CPU are ALU and the Shifter.

The controller of a RISC machine can be divided into two relatively independent components (see Section 2.4.3): the major component that controls instruction execution and a supporting component that controls unusual internal and external conditions. In this design strategy, unusual conditions are considered as secondary effects and they will be examined in terms of how they will affect the primary event–instruction execution. Furthermore, as discussed in Section

2.4.3, unusual condition handling as well as many other control sequences can be reduced to sequences of internal instructions that are similar to the regular instructions in the instruction set. Therefore, the microarchitect can and should concentrate initially only on the primary event–instruction execution, and temporarily ignore the unusual conditions and other complex control sequences.

The microarchitect can build the abstract model of the microarchitecture (see Figure 5-3-1) according to the instruction set by concentrating only on instruction execution. All he has to do is to select a set of macro-modules that are needed by the instruction set and then design the Instruction Execution Controller that controls the macro-modules. After the microarchitect is satisfied with the behavior of this abstract model, he can then turn the abstract model into the expanded model (see Figure 5-3-1) by expanding macro-modules into micro-modules and by taking unusual conditions detection into account. The abstract and expanded models are discussed in Section 5.3.2. Notice that I have changed the definitions of macro and micro-modules slightly in

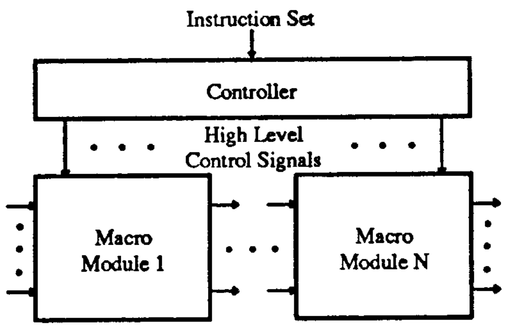


Figure 5-3-2 The Abstract Model

The microarchitect constructs this model to study possible instruction execution schemes. The microarchitect begins the construction by first selecting a set of high level macro-modules according to the execution scheme on his mind. He then design a simple controller to translate the instruction set into a set of high level control signals to control the high level macro-modules. Examples of macro-modules in the SPUR CPU are Cache Controller Interface and Operand Supplier.

this section. Instead of using them to describe components in both the datapath and the controller, I have used them exclusively for components in the datapath. These definitions will be followed for the rest of the discussion.

5.3.2. Different Models for Different Issues

The instruction set, the external interface, and performance requirements are usually fixed at the microarchitectural level. Therefore, the first task of the microarchitect is to develop an instruction execution scheme that can fulfill the external interface and performance requirements. The type of model the microarchitect needs at this stage is the abstract model shown in Figure 5-3-2. He will use this model to verify that the external interface and performance requirements are indeed met. Furthermore, he will also use this model to answer questions concerning the microarchitectural issues: (1) off-chip communications, and (2) pipeline and clocking.

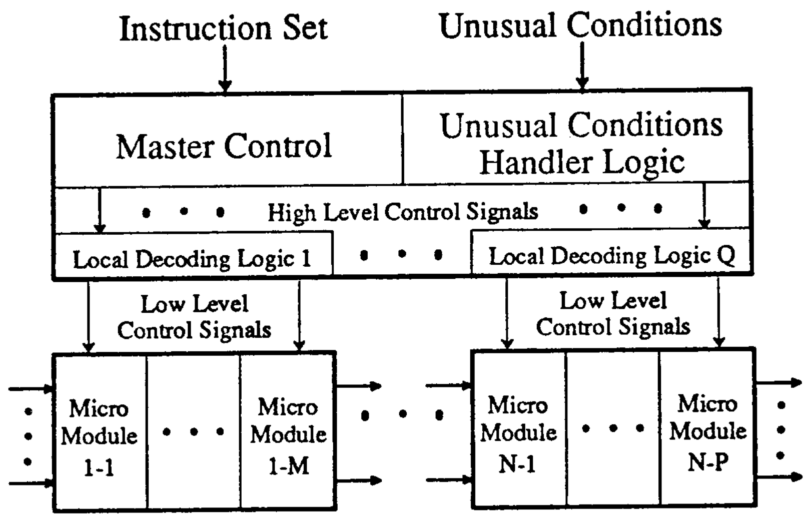


Figure 5-3-3 The Expanded Model

In the expanded model, the macro-modules in abstract model (Figure 5-3-2) are expanded into low level micro-modules. The controller in abstract model is expanded into master control, local decoding logic blocks, and the unusual conditions detection logic. ALU and SHIFTER are examples of micro-modules in the SPUR CPU.

The microarchitect must keep the abstract model as simple as possible so that the effects of his design decisions can be identified more directly. This can be accomplished by ignoring details that have little effects on the microarchitectural issues that are being investigated. Since pipeline and clocking and off-chip communication are the two microarchitectural issues to be investigated by the abstract model, second order effects such as unusual conditions handling can be ignored. In order to simplify the investigation further, the microarchitect may also want to group instructions into types and examine how the abstract model will execute each type of instruction instead of examining individual instructions. In this simple abstract model, where second order effects are being ignored, the set of high level control signals is a good indication of the controller complexity. Similarly, the set of macro-modules is a good indication of datapath complexity.

Once the microarchitect has verified the external interface and performance requirements have been met, he must move onto microarchitectural issues such as on-chip interaction, micro-modules selection, and resource allocation. Since these microarchitectural issues require a more detailed model, he must expand the abstract model into the expanded model shown in Figure 5-3-3. In this model, the macro-modules are expanded into micro-modules and logic is added to detect all the unusual conditions. Furthermore, in order to get a better understanding of the micro-modules, the microarchitect must examine how the expanded model executes each instruction instead of how it executes each type of instruction. Finally, if the instruction is not provided directly by the external world, an instruction supplier must be added to this model. This is not shown in Figure 5-3-3 in order to keep this figure simple.

The microarchitect should be able to learn enough about the microarchitecture from the abstract and expanded models that he can draw out a detailed floor plan. The floor plan can be considered as a physical model. Table 5-3-1 summarize the various models I proposed and the microarchitectural issues each model investigates. The abstract model (Figure 5-3-2) is used mainly to investigate behavioral issues. However, structural information is implied in the abstract model by the macro-modules connection scheme. Similarly, although the expanded

	Behavioral Issues		Structural Issues			Physical Issues
	Off-Chip Communication	Pipeline and Clocking	On-Chip Interaction	Micro-Modules Selection	Resources Allocation	Floor Planning
Abstract Model	MAJOR	MAJOR	minor	–	–	–
Expanded Model	–	–	MAJOR	MAJOR	MAJOR	–
Physical Model	–	–	minor	–	–	MAJOR

Table 5-3-1 The Microarchitectural Models and Issues
"MAJOR" means that issue is a major concern of that model.
"minor" means that issue is a minor concern of that model.
"–" means that issue is a not a concern of that model.

model (Figure 5-3-3) is used mainly to investigate structural issues, the microarchitect is providing behavioral information when he describes the behavior of the micro-modules.

Either the abstract or the expanded model can be used to estimate the average number of cycles per instruction (C) for the performance model (see Section 4.1). This can be accomplished by counting the number of cycles either model takes to execute a set of instructions with the proper mix of instructions. Both models can also be used to estimate the cycle time T for the performance model. This can be accomplished in two different ways:

(1) A simulator for the model should be able to perform simple timing analysis if explicit timing information is provided for each module.

(2) A simulator should be able trace all the sequential events within each clock cycle and the microarchitect can estimate the cycle time based on these lists of events.

In the first approach, there is always the danger of looking at the wrong critical path within a module and subsequently assigning the wrong timing information to the module. In the second approach, the trace information must be interpreted and this can be cumbersome. The best chance of success is to use Approach 1, with Approach 2 acting as a check.

5.3.3. CAD Tools Considerations

The CAD tools needed to make the design process more systematic will be discussed in more details in Section 5.3.4, Section 5.3.5, and Section 5.3.6 when the SPUR CPU is used as an example to illustrate different stages of the systematic approach. In this section, I make some general observations concerning CAD tools.

5.3.3.1. Unifying Different Levels of Verification

In the SPUR CPU design process (Figure 3-2-1), the macroarchitecture, microarchitecture, and the layout were verified sequentially and independently by instruction level, behavioral level, and switch level simulations, respectively. The results of these simulations must be studied independently by the macroarchitect, microarchitect, and the logic designer. One important observation, stated in Section 3.2.4, was that this independent verification strategy required a lot of human interaction time. In order to reduce verification time, the redundancy between different levels of verification must be reduced. In the current SPUR CPU design process, behavioral simulation results were verified by comparing them with the instruction level simulation results. Similarly switch level simulation results were verified by comparing them with the behavioral level simulation results. This approach, however, still requires a lot of human interaction and format conversions (see Figure 3-2-4). Mixed-level simulation is a better approach.

Ideally, we would like to have a mixed-level simulator that will accept modules of different levels of abstraction during various stage of the design process. Initially, only high level macro-modules should be used such that high level design decisions can be made. These high level decisions create a rough specification of each macro-module and enable each of them to be replaced by a set of low level micro-modules. In order to verify that this set of micro-modules can indeed replace the high level macro-module, they must be simulated together with other high level macro-modules.

The macro/micro modules substitution, however, should be a two-way street. During mixed-level simulation, the designer should be able to extract important parameters from the set of low level micro-modules being simulated. These parameters can then be used to update and modify the corresponding high level macro-module. After the important parameters are extracted, the set of low level micro-models can then be replaced by the updated, more accurate, high-level macro-model to reduce simulation time when we simulate other low level micro-modules.

5.3.3.2. Timing Verification

The SPUR CPU timing was verified at the circuit and switch levels. At the circuit level, the circuit designer verified the timing of critical circuits by SPICE even before starting the layout. After the layout of these circuits was completed, the circuit designer measured the parasitic capacitance and resistance such that the SPICE models could be updated for a more accurate timing analysis. In the switch level, the timing of the entire CPU was verified by Crystal [SMH85] which extracted the critical paths from the switch level description. The exact delay of these critical paths are then again verified by SPICE in the circuit level.

One drawback of the SPUR CPU approach is that timing verification is done at the low level only, and working at the low level is tedious. Working at the high level is possible here because the timing requirements at the low level are direct results of high level decisions. In order to take advantage of this possibility, we need a mixed-level timing verifier. A mixed-level timing verifier will enable the designer to use the high level work to drive the low level timing verification. For example, if at the high level the designer decides that the Functional Unit must have a critical delay less than M during phase N, then any micro-modules that are part of the Functional Unit (Example: ALU, SHIFTER) must all fulfill this same requirement. The "Abstract Timing Verifier" by Dave Wallace [WaS86] is an example of mixed-level timing verifier.

Another drawback of the SPUR CPU approach is that timing verification is done completely independent of logic simulation. Like most timing analyzers, Crystal does not care nor know anything about logic. Consequently, to prevent the timing analyzer from chasing false critical paths, the user must place "flow control" attributes on certain transistors [SMH85]. This takes a lot of time and can also be unreliable. The switch level simulator should be able to help the designer in placing these "flow control" attributes because the switch level simulator knows the direction each signal propagates during switch level simulation.

One point worth noticing is that timing analyzers such as Crystal were designed at a time when computer time was relatively expensive. They were specialized tools designed intentionally to ignore the logic aspect of the circuit such they can run rapidly in a relatively slow computing environment. The price to pay was human preparation time. In current computing environment, computer time is relatively cheap. It is more desirable to have tools that require less human preparation time although it may consume much more computer time.

5.3.3.3. Documenting the Design Decisions

Most practical VLSI projects are so complex that nobody can specify it accurately until some work has been done on it. Informality is a powerful strategy for dealing with complexity because it allows the designer to describe the big picture without worrying about the details. Therefore, an imprecise but brief specification, for the lack of a better word, is *good* at the beginning of an VLSI project. Instead of demanding an complete precise specification from the start, a good design process or system should help the user to specify and refine the specifications continuously. This is accomplished by continuously demanding the designer to answer the following types of questions:

- Given a number of interacting design objectives and goals, how should I prioritize them?
- Given a number of alternative choices, which alternatives should I pick?

Unfortunately, due to the imprecise specification, making these decisions are not always easy. Consequently, a design decision is frequently nothing but an educated guess and the design process is an evolution process. At each stage, the design is a proposal whose correctness and effectiveness must be proved. The proof can be performed either by formal mathematical techniques or experiments. Mathematical techniques, however, are for mathematicians–engineers should always answer their doubts by experiments!

Since the design at any stage is only a proposal that must be evaluated by experiments, design decisions that led to that design must be documented systematically. The above observation is the basis for the development of the theory of plausibility design [HAD88]. Since the underlying philosophy of this thesis is to keep things simple and practical, we will not go into the details of the theory of plausibility design. But I do want to point out the two things the theory of plausible design tries to address:

(1) The sequences of design decisions and the cause-effect relationships between the design decisions.

(2) The assumptions or the evidence used by the designer to justify his design decisions.

In other words, we must find a systematic way to document not just the decisions but also the assumptions and evidences behind all the decisions. However, in order to keep the procedure simple, the designer should only document the important decisions and decisions that are based on questionable assumptions.

5.3.4. Stages of the Systematic Approach

Based on the above discussions, I believe a systematic approach to microarchitectural design should have three stages: the abstract stage, the expansion stage, and the floor planning stage.

The Abstract Stage

Construct the abstract model (Figure 5-3-2) and then use it to conduct primary studies on microarchitectural issues: (1) off-chip communication, and (2) pipeline and clocking.

The Expansion Stage

Construct the expanded model (Figure 5-3-3) by expanding the macro-modules and the controller in the abstract model. The microarchitectural issues to be studied here are: (1)

micro-modules selection, (2) resources allocation, and (3) on-chip interaction.

The Floor Planning Stage

Based on the information we learned from the abstract and expanded model, construct a detailed floor plan for the microarchitecture.

The abstract stage and the expansion stage should be repeated for alternative microarchitectures until satisfactory solutions are found for all microarchitectural issues. Mixed-level simulation that uses a mixture of macro-modules and micro-modules can be used. During simulation, all important and questionable design decisions must be documented systematically. Instead of using toy examples to illustrate this procedure, I will use the SPUR CPU to illustrate the details of this procedure in Section 5.4, Section 5.5, and Section 5.6.

5.4. The Abstract Stage of Microarchitectural Design

In the beginning, the macroarchitect created the instruction set and the interface specifications. The microarchitect must then find a pipeline and clocking scheme that can execute the instruction set and derive an off-chip communication strategy that can satisfy the interface specifications. As I will explain later, the pipeline and clocking and off-chip communication issues are closely related.

5.4.1. Off-Chip Communication

The general off-chip communication problem for a microprocessor is shown in Figure 5-4-1 to be a three-port problem. Most modern microprocessors include the Instruction Supplier on chip. The microprocessor then only has to communicate off chip with the Data Supplier and the Coprocessor(s). No matter what the situation, the microarchitect must design the Data Supplier port, the Coprocessor port, and (if necessary) the Instruction Supplier port such that the performance goal is met and the resources and complexity requirements are still within the constraints.

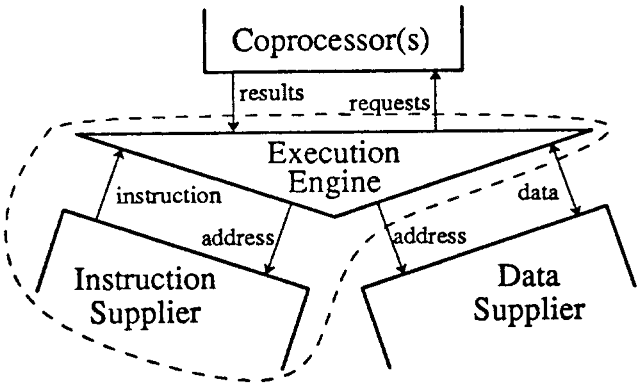


Figure 5-4-1 Off-Chip Communication

The microprocessor off-chip communication is a three-port problem. The instruction execution engine must communicate with the data supplier, the instruction supplier, and the coprocessor(s). Many modern microprocessors have an internal instruction cache which eliminates the instruction supplier interface. Some processor even include complex functions such as floating point operations on-chip to eliminate the coprocessor interface. This option may run into trouble in the future when more complex functions are desired and the only way to provide them is via coprocessors.

The performance specified by the macroarchitect is usually in terms of clock cycles. The microarchitect must decide when to drive or receive the interface signals within a given cycle. The two limiting resources are the number of pins available and the power to drive the output pins. As far as the number of pins is concerned, the sum of input (N_{in}), output (N_{out}), bidirectional (N_{bi}), and power pins ($2 \times N_{Vdd}$) must be smaller than or equal to the total number of pins available ($N_{available}$).

$$N_{in} + N_{out} + N_{bi} + 2 \times N_{Vdd} \leq N_{available} \tag{5.4.1}$$

The number of power pins ($2 \times N_{Vdd}$) is twice the number of V_{dd} pins (N_{Vdd}) because a *GND* pin is needed for each V_{dd} pin. In the old days, one pair of V_{dd} and *GND* was be sufficient. But today, due to high switching frequency and pin inductance, the number of power pins needed is a function of the switching frequency (F_{switch}), the number of output pin (N_{out}), and the number of bidirectional pin (N_{bi}):

$$N_{Vdd} = func(F_{switch}, N_{out}, N_{bi}) \tag{5.4.2}$$

John Keller [Kel85] suggested that for a given switching frequency, a simple solution is to assign a pair of power pins (one V_{dd} and one GND pin) for each group of M output or bidirectional pins. This implies the number of V_{dd} pins can be written as:

$$N_{Vdd} = \frac{N_{out} + N_{bi}}{M} \tag{5.4.3}$$

If one just look at Equation 5.4.3, one may think that it is possible to reduce the number of power pins ($2 \times N_{Vdd}$) by time multiplexing. This is not the case. Although time multiplexing will reduce N_{out} or N_{bi} or both, it also increases the switching frequency (F_{switch}). According to Equation 5.4.2, this increase in switch frequency will negate the effects of the reduction in N_{out} or N_{bi}. In order to rewrite Equation 5.4.3 to take this consideration into account, I define the term logical output L_{out}:

L_{out} = The number of signals the microprocessor must send to the outside world

By definition, the number of logical outputs (L_{out}) will not change by time multiplexing. It is the sum of N_{out} and N_{bi} only if time multiplexing is never used to multiplex more than one output signal onto one output or bidirectional pin. In other words:

If time multiplexing is not used: $L_{out} = N_{out} + N_{bi}$

If time multiplexing is used: $L_{out} > N_{out} + N_{bi}$

Using this term logical output (L_{out}), I can rewrite Equation 5.4.3 as:

$$N_{Vdd} = \frac{L_{out}}{M} \tag{5.4.4}$$

The number of logical outputs L_{out} is used in Equation 5.4.4 to emphasize the fact that the number of power pins needed ($2 \times N_{Vdd}$) will not change by time multiplexing. The SPUR CPU has approximately 120 logical outputs. After careful considerations of the pin inductance, switching frequency, and worst case loading, the SPUR circuit designers [Jeo88] decided that $M = 6$ is sufficient. The SPUR CPU therefore has 20 pairs of V_{dd} and GND pins.

The reason why the number of power pins needed cannot be reduced by time multiplexing also applies when considering the power required to drive the output pins ($Power_{out\_pin}$). Since $Power_{out\_pin}$ cannot be reduced by time multiplexing, $Power_{out\_pin}$ in general is not a function of $(N_{out} + N_{bi})$. The power required to drive the output pins is also a function of the number of logical outputs L_{out}:

$$Power_{out\_pin} = func(F_{switch}, L_{out}) \tag{5.4.5}$$

In conclusion, while time multiplexing can reduce the number of physical pins (N_{in}, N_{out}, N_{bi}), it does not reduce the number of power and ground pins. It also does not reduce the power required to drive the output pins. Furthermore, time multiplexing also increases the complexity of the chip.

Figure 5-4-2 shows the simple SPUR CPU off-chip communication strategy that does not involve time multiplexing. In pursuing this simplest solution, a dedicated set of 32 pins is allocated to the coprocessor interface even though the coprocessor instruction only occurs rarely.

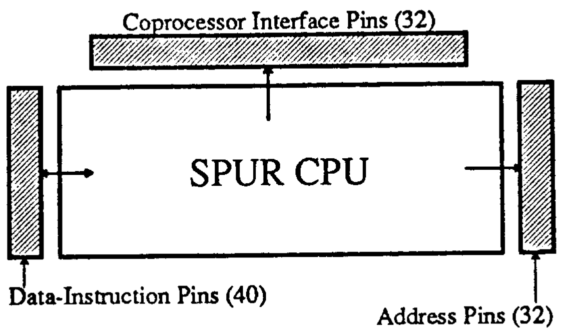


Figure 5-4-2 SPUR CPU Off-Chip Communication

The SPUR CPU designer took the easist way out and pick the simplest solution. This solution allocate a separate set of pins for data, address, and the coprocessor interface. The coprocessor interface broadcasts every instruction the SPUR CPU receives from its internal instruction cache to the coprocessor.

Before accepting any solution, the microarchitect must make sure the solution meets the performance requirements and are within the resources and complexity constraint. Answering questions such as these below helps make the decision:

- Are there enough pins for this solution?
- How much power does it take to drive all the output and bidirectional pins?
- What kind of performance does this solution will give?
- How complex is it to debug and implement this solution?
- Is this the most efficient way to use the limited pin resource?

While you can give quantitative answers to the first two questions, it is hard to give absolute answers to the last three questions. It is much easier give relative answers by comparing different solutions. Furthermore, in order to answer all these questions, you must make some assumptions about clocking.

5.4.2. Pipeline and Clocking

Clocking schemes must be studied together with pipelining because the longer the pipeline–that is more pipe stages for each instruction–the shorter the potential clock cycle. Most useful work is done during the high time of the clock in MOS technology. Therefore the number of clock phases per cycle together with the number of pipeline stages for each instruction determine the number of time slots in which useful work can be done. This is illustrated in Figure 5-4-3, where the SPUR CPU pipeline and clocking scheme are used as an example. In general, the more explicit time slots the easier it is to design dynamic logic [Kon85]. Unfortunately, more explicit time slots also means that more time will be wasted between time slots because explicit non-overlap dead time must be placed between phases to guard against clock skew problems (see Figure 2-3-3).

There is a subtle difference between designing a traditional pipeline and a RISC-style pipeline that handles integer instructions only. In designing a traditional pipeline, the main concern is

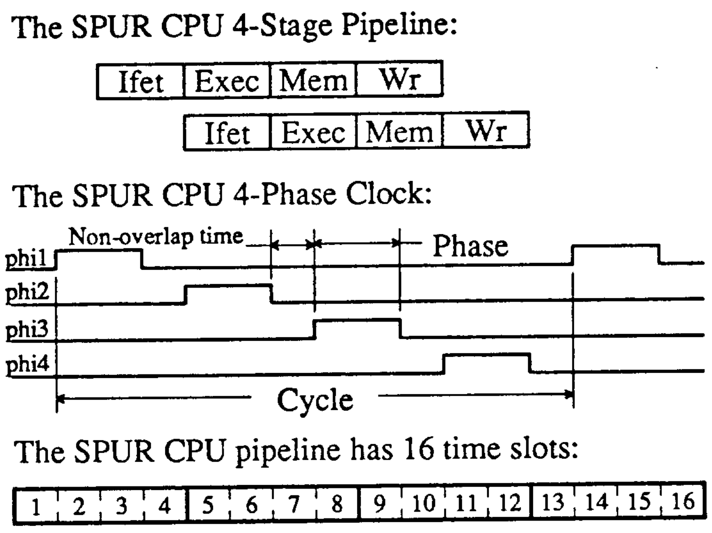


Figure 5-4-3 Pipeline and Clocking

The SPUR CPU uses a 4-stage (Ifet, Exec , Mem, Wr) pipeline–each instruction takes four cycles to finish. The SPUR CPU also uses a 4-phase clock–each cycle is divided into four phases. The 4-stage pipeline together with the 4-phase clock provide 16 time slots to do useful work.

how to schedule the issue of instructions such that there will not be any structural conflict [Kog81]. An example of structural conflict is two instructions trying to use the ALU during the same cycle. On the other hand, a RISC processor that only supports integer operations can execute the simple instructions in a very uniform manner. This makes structural conflicts very easy to detect and eliminate. Therefore, the main concern in designing a RISC-style pipeline is not instruction scheduling–the main concern is what kind of resources are needed to eliminate all structural conflict such that instruction can be issued every cycle. Since most if not all structural conflicts can be eliminated from a RISC-style pipeline, the are only two things left that can degrade a RISC-style pipeline's efficiency: branching and data dependency. Important questions the microarchitectural must keep in mind when he designs a RISC-style pipeline are:

- The cost of branching–how many cycles are wasted?

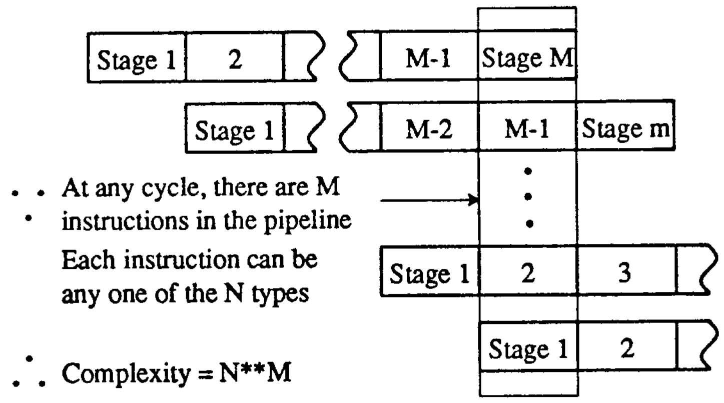

Stage 1
2
M-1
Stage M
Stage 1
M-2
M-1
Stage m
At any cycle, there are M instructions in the pipeline
Each instruction can be any one of the N types
Complexity = N**M
Stage 1
2
3
Stage 1
2

Figure 5-4-4 M-Stage Pipeline

This is an ideal M–stage pipeline where any one of the N types of instructions can be issued at every cycle. At any given cycle, this pipeline can be in any one of the $N_{type}^{M_{stage}}$ possible states. All these states must be controlled properly. Therefore, for the same number of instruction types, longer pipeline also need more complex control which may offset the advantage of longer pipeline.

- The cost of data dependency–how many cycles will an instruction have to wait for data?

- For a given pipeline length, which clocking scheme achieves the best cycle time?

- What are the type and complexity of the macro-modules are needed to implement a pipeline that will allow any instruction to be issued at every cycle?

The first two questions are Computer Science problems and a lot of work has been done on them [Kog81] [McH86]. The last two problems are Electrical Engineering problems and not much work has been done. I think the best way to answer these last two questions is to follow the following procedure:

(1) Divide the instruction set into N types of instructions. Register-register operations (Reg_Reg) and load operations (Load) are two examples of instructions types in the SPUR CPU.

	Ifet	Exec	Mem	Wr
Reg_Reg	I-Unit : Read PC Logic : Inc. I-Unit : Fetch	Operand Supplier : Read Functional Unit : Operate		Operand Supplier : Write
Load	I-Unit : Read PC Logic : Inc. I-Unit : Fetch	Operand Supplier : Read Functional Unit : Eff. Address	CC Interface : Send Address Detect hit/miss	Operand Supplier : Write
Cmp_Branch	I-Unit : Read I-Unit : Fetch	Operand Supplier : Read Functional Unit : Condition PC Logic : Target Address		

Figure 5-4-5 Operations for Several SPUR Instruction Types

The operations for Reg_Reg, Load, and Cmp_Branch type instructions are listed in the (macro-module : task) format. All instructions within a type must require the same macro-module to perform the same task during the same stage of the pipeline. For example, for all Reg_Reg instructions, during the Exec stage, the Operand Supplier must supply the operands and the Functional Unit must operate on the operands.

(2) List the steps it takes to execute each type of instruction. This will give the microarchitect ideas about what pipe stages are needed to execute all types of instructions uniformly.

(3) Construct an uniform M–stage pipeline that has the potential to execute one instruction per cycle (Figure 5-4-4). Examples of pipe stages in the SPUR CPU are: Instruction fetch (Ifet), register read and execution (Exec), memory access (Mem), and register write (Wr).

(4) For each type of instruction, list the operations for each stage of the pipeline. They may be given informally at first but eventually the designer must specify the operation in terms of what macro-module is needed to perform what task in the (macro-module : task) format. Figure 5-4-5 shows several examples.

(5) Based on the results of Step 4, construct the list of necessary macro-modules. For example, by examining Figure 5-4-5, one can construct this list of macro-modules for Reg_Reg, Load, and Cmp_Branch: I-Unit, PC-Logic, CC_Interface, Operand Supplier,

and Functional Unit.

(6) Construct the list of operations each macro-module must provide by using the result of Step 4 and examining the $N_{type}{}^{M}$ possible pipeline states. For example, Figure 5-4-6 shows the Operand Supplier must provide Read and Write operations every cycle.

(7) After careful examination of each macro-module's list of operations, propose a clocking scheme. For example, since the easiest way to implement a large register file in CMOS is to precharge the bit lines before read and write, the Operand Supplier must perform (1) read, (2) precharge for write, (3) write, and (4) precharge for read within a cycle. The 4-phase clock is a natural clocking scheme for these four distinct events.

In the above procedure, all instructions of the same type will end up having the same execution model (Figure 5-4-5) and make the same contributions to the list of macro-modules and their

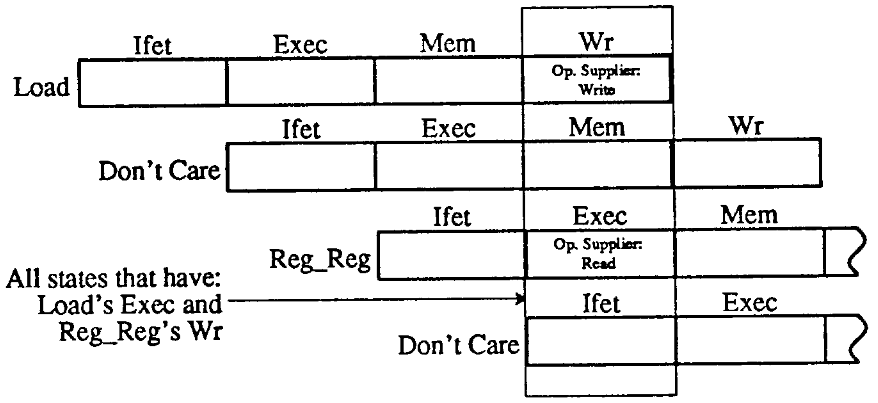


Figure 5-4-6 Potential Structural Conflict

This is a generic diagram for all the pipeline states that has Reg_Reg's Exec stage and Load's Wr stage. Load's Exec Stage requires the (Operand Supplier : Read) operation and Reg_Reg's Wr stage requires the (Operand Supplier : Write). The Operand Supplier must be able to perform Read and Write within a cycle in order to prevent structural conflict. For simplicity, only relevant operations are shown in this figure.

lists of operations. Since the set macro-modules and their lists of operations define the level of abstraction, the division of instructions into types (Step 1) will determine the level of abstraction the microarchitect looks at the proposed microarchitecture. For example, if in Step 1 we divide the instructions into high level types such as Reg_Reg, then we will be listing the operations for macro-modules such as Functional Unit in Step 4. On the other hand, if in Step 1 we divide the instructions into low level types such as Add and Shift, then we will be listing the operations for micro-modules such as ALU and Shifter in Step 4.

At the lowest level, every instruction is a separate type because each instruction must behave differently in some way from the others. This low level of detail is probably not necessary if one only wants to study pipeline and clocking alternatives. The microarchitect should therefore pick a level of abstraction just low enough to show the characteristics of different pipeline and clocking alternatives but not so low that it requires a large number of modules each with a list of very specific operations. The only way to find out the proper level of abstraction by iterating Step 1 through Step 6 of this section. Another reason why iteration may be necessary is that the microarchitect may find out in Step 6 that the list of operations for a macro-module is too long and thus the macro-module is too complex. He may then have to go back to Step 4 and either assign some of its operations to other existing modules or create some new macro-modules. Since this is an iteration process and at each iteration the designer may have to examine as many as $N_{type}^{M_{mr}}$ states, CAD tools must be developed to ease the designer's task.

5.4.3. The Abstract Model of the Microarchitecture

The procedure described in last section will not only create a pipeline and clocking scheme but will also select a set of macro-modules (Step 4). For example, if this procedure is used for the SPUR CPU, the set of macro-modules selected at this point will be: I-Unit, Operand Supplier, Functional Unit, CC_Interface, PC-Logic, and Special Registers. In order to complete the abstract model of the proposed microarchitecture, these macro-modules must be connected. A systematic way to propose a connection scheme is to examine the interaction between the macro-modules

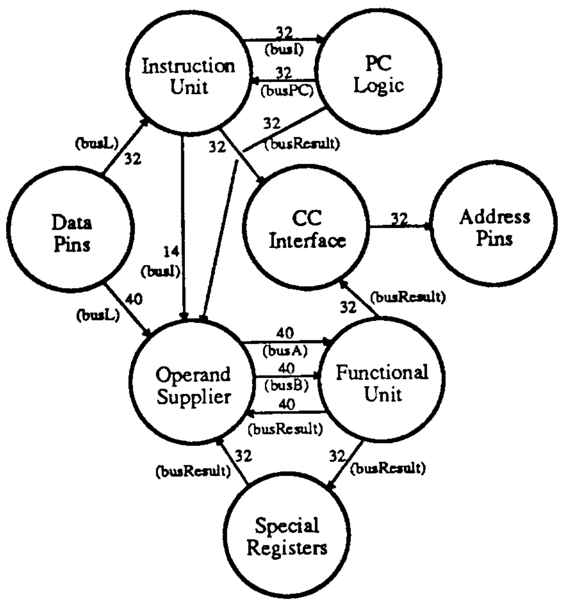


Figure 5-4-7 On-Chip Interaction

Each node in this graph represents a macro-module. Each arc in this graph represents a set of signals that must be sent from one macro-module to the others. The number associated with each arc is the number of signals in the set and the name of the bus assigned to the arc(s) is in parentheses. This graph is constructed by looking at one macro-module at a time and consider to which macro-module must it send its output.

graphically.

Figure 5-4-7 is a directed graph that shows the interaction between the various macro-modules of the SPUR CPU. The simplest connection scheme can be derived from this graph by assigning a signal bus to each arc. Such a scheme, however, will also be very expensive in terms of resources. A better approach is to group some of the arcs together and assign them a single bus. For example, in Figure 5-4-7, *busResult* is assigned to all output arcs of the Functional Unit and some input arcs to the Operand Supplier. Furthermore, *busL* is assigned to both arcs connecting to the Data Pins. Figure 5-4-7 is the logical bus structure. The physical bus structure, which is shown in Figure 5-4-8, is slightly different.

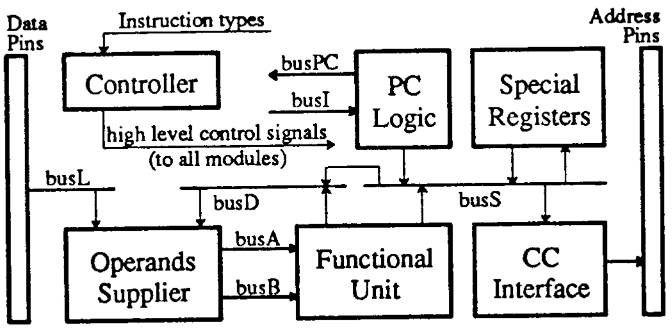


Figure 5-4-8 SPUR CPU Abstract Model

In Figure 5-4-7, *busResult* is assigned to all output arcs of the Functional Unit and some input arcs to the Operand Supplier. When implementation is taken into account, *busResult* is broken into *busS* and *busD* to reduce the bus capacitance that each module has to drive. PC Logic and Special Registers must first send the values onto *busS* which then drives *busD*. In general, asking one bus to drive another can cause a lot of delay. It is acceptable here because PC Logic and Special Registers put values onto *busS* during $\phi 2$ and *busD* does not have to be driven until the following $\phi 4$–not in the same phase.

Figure 5-4-8 is the block diagram of the abstract model of microarchitecture we have proposed thus far. In order to verify and evaluate this proposed microarchitecture, it must be modeled by hardware description language. This is called the abstract model and the major issues this model will be used to study are: (1) off-chip communication, and (2) pipeline and clocking. In modeling, it is utmost important to keep in mind the things one wants the model to examine and keep the model just complex enough to do the job. Therefore, the Instruction Unit is not included in this model and instruction types instead of individual instructions are used. Ideally, for each proposed microarchitecture at the abstract level, the microarchitect should use an abstract model such as Figure 5-4-8 to answer all the unanswered questions before moving on to the detail level of the microarchitecture. Figure 5-4-9 reviews how the abstract model is constructed.

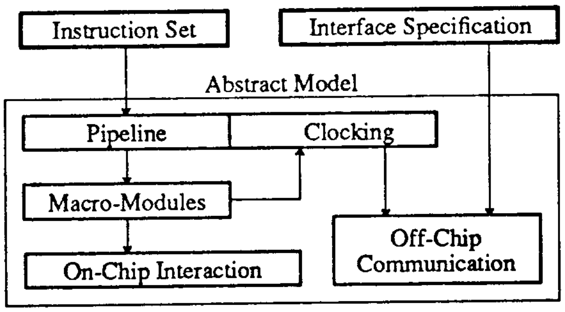


Figure 5-4-9 Building the Abstract Model

This is a flow chart on the construction of the abstract model (Figure 5-4-8). In the beginning, there are the instruction set and the interface specification. A pipeline scheme is proposed by looking at the instruction set. The execution model for each type of instruction with respect to the pipeline gives a list of the macro-modules required and their operations. The list of operations for each module can then be used to determine the clocking scheme. The clocking scheme then together with the interface specification will determine the off-chip communication strategy. Finally, on-chip interaction must be taken into account in order to connect the set of macro-modules together.

5.5. The Expansion Stage of Microarchitectural Design

The previous section shows how to create, specify, and examine a microarchitecture at the abstract level. The resulting abstract model consists of a set of macro-modules controlled by a high level controller. The goal of the expansion stage is to obtain a detailed specification of the microarchitecture by expanding the macro-modules and the high level controller.

5.5.1. Micro-Modules Selection and Resources Allocation

The functionality of the macro-modules are defined during the abstract stage. The microarchitect has many options to achieve this functionality by selecting different sets of micro-modules. For example, Figure 5-5-1 is one possible option the microarchitect may use to expand the macro-module Functional Unit. Another option is to use a 32-bit barrel shifter instead of the EXT_INS and SHIFTER. The microarchitect must evaluate the performance, resources, and complexity tradeoffs quantitatively when he considers his options. This is similar to the problem studied in Chapter 4 where the microarchitect has many options in what features to include in the CPU. The systematic approach shown in Section 4.7.3, which was based on the performance resources and complexity tradeoffs, can be applied here.

Figure 5-5-2 illustrates graphically the performance, resources, complexity tradeoffs. This is similar to Figure 4-7-2 except that in Figure 4-7-2, the options on the vertical axis corresponds to different features that can be included in the CPU. Here in Figure 5-5-2, the options on the vertical axis corresponds to different ways a macro-module can be expanded. For example, suppose we are considering the tradeoffs in building the macro-module Instruction Unit. Option 1 could be a simple (low in complexity), low cost in resource, and low performance direct-mapped cache. Option 2 is similar but with bigger cache size (more resources). Option 3 can be a cache with prefetching (more complex) such that the cache size can be smaller (fewer resources) and still achieve the same performance as Option 2. Option 4 can be considered as a set associative cache with prefetch which gives higher performance than Option 3 but also require more resources and higher degree of complexity.

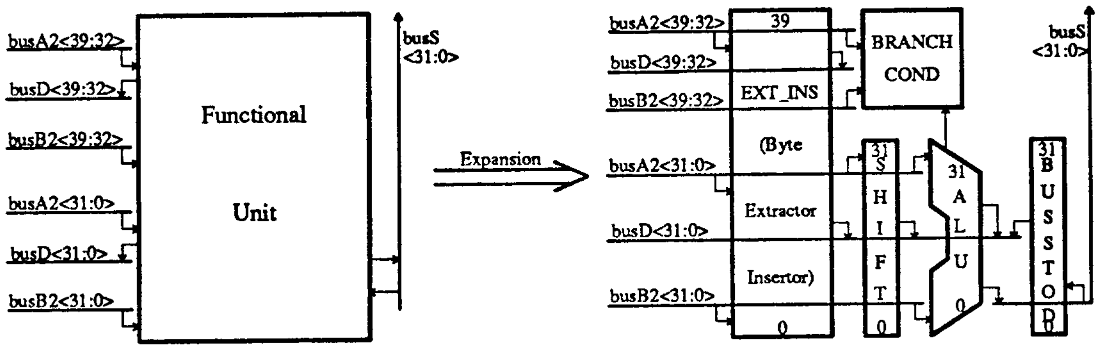


Figure 5-5-1 Micro-Modules Selection for the SPUR CPU Functional Unit

In this example, the macro-modules Functional Unit is expanded into micro-modules EXT_INS (byte extractor insertor), SHIFTER, ALU, BRANCH COND, and BUSSTOD.

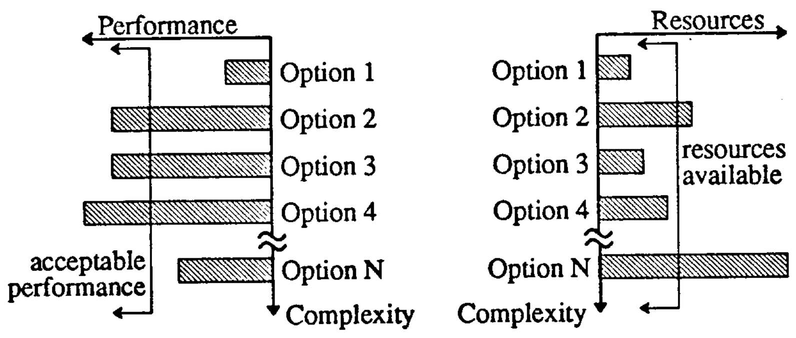


Figure 5-5-2 Performance Resources and Complexity Tradeoffs

The options, which correspond to different ways the macro-modules can be expanded, are placed in increasing complexity on the vertical axis. The performance and resources needed for these options are plotted on the horizontal axes. The performance requirement and resources available for each macro-module place the "acceptable performance" bound on the performance axis and the "resource available" bound on the resources axis.

Each macro-module has is own minimum performance requirement. This requirement is a direct result of the overall performance goal and can be obtained during simulation at the abstract stage. The performance requirement for each macro-module places an "acceptable performance" bound on the performance axis in Figure 5-5-2. Given this performance requirement, the microarchitect must allocate enough resources such that an option that has acceptable performance and complexity can be built. Below is an example on how we can apply the systematic approach discussed in Section 4.7.3 to select an option to expand the macro-modules. This is very similar to the example shown in Section 4.7.3 except there the options are what features to be included in the SPUR CPU.

(1) Make an educated guess on how many resources you are willing to spend on this macro-module. This places a "resource available" bound on resources axis in Figure 5-5-2.

(2) Within this bound, pick the simplest option available.

(3) If this option's performance is within the acceptable range, then mission accomplished. Otherwise, go to Step 4.

(4) If there are any other options within the resource bound, pick the next more complex option and go back to Step 3. Otherwise go to Step 1 and increase the resources available bound.

Using Figure 5-5-2 as an example, Step 2 of this procedure will pick Option 1. However, in Step 3, we will find out Option 1's performance is below the acceptable range. In Step 4, Option 2 is not chosen in Step 4 because it uses more resources than available. Option 3 will be chosen because it is less complex than Option 4. Finally, when we get back to Step 3, we will find out Option 3's performance is acceptable.

5.5.2. On-Chip Interaction and Second Order Effects

At the abstract stage of the design, on-chip interaction concerns with the interaction among various macro-modules. Section 5.4 showed how this problem can be solved by connecting the macro-modules via signal busses embedded in the datapath. At the expansion stage, on-chip

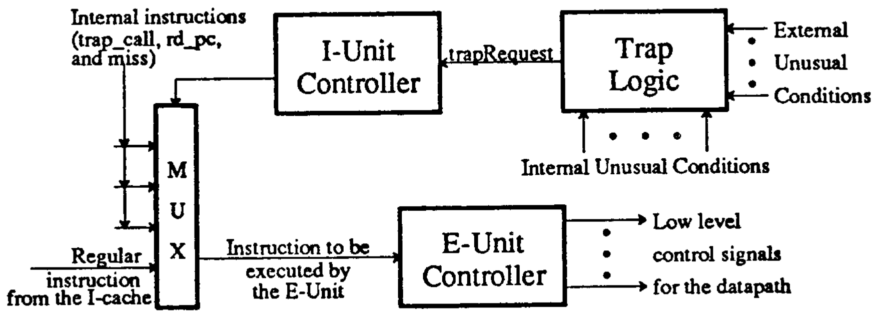

Internal instructions (trap_call, rd_pc, and miss)
I-Unit Controller
trapRequest
Trap Logic
External Unusual Conditions
Internal Unusual Conditions
M U X
Regular instruction from the I-cache
Instruction to be executed by the E-Unit
E-Unit Controller
Low level control signals for the datapath

Figure 5-5-3 The SPUR CPU Control Strategy

The SPUR CPU is controlled by three modules: Trap Logic detects all internal and external unusual conditions, I-Unit Controller controls the instruction unit that delivers instruction, and E-Unit Controller decodes every instruction it receives into control signals. Whenever the Trap Logic detects an unusual condition, all it has to do is to send a signal (*trapRequest*) to the I-Unit Controller. The I-Unit Controller then delivers the proper internal instructions that can handle the trap to the E-Unit Controller.

interaction concerns the control of the micro-modules. We must also take into account second order effects–such as trap detection–that are ignored during the abstract. Second order effects are very important to the controller design during the expansion stage. A systematic approach to this problem can be summarized in three words: isolation, specialization, and optimization.

Isolation

Isolate different aspects of the control function into sub-functions that have minimum interaction among them.

Specialization

Design specialized modules for the sub-functions.

Optimization

Finally, optimize the specialized modules.

In order to illustrate this three-step approach, I will use the SPUR CPU as an example. The SPUR CPU control strategy is shown in Figure 5-5-3. The control function of the SPUR CPU can be isolated into three sub-functions. These three sub-functions and their respective specialized modules are:

- The control of the Instruction Unit (I-Unit) that delivers the instruction. This is handled by the I-Unit Controller.
- The control of the Execution Unit (E-Unit) that executes the instruction. This is handled by the E-Unit Controller.
- The detection of internal and external unusual conditions. This is handled by the Trap Logic.

One major optimization we performed in the SPUR CPU is the use of internal instructions to further reduce the on-chip interaction. As illustrated in Figure 5-5-3, the Trap Logic asserts the *trapRequest* signal whenever it detects any unusual condition. Upon receiving the *trapRequest* signal, the I-Unit Controller will deliver the internal instructions that handle trap to the E-Unit controller. This optimization therefore reduces the interaction between the Trap Logic and the I-Unit Controller to one signal–*trapRequest*. This optimization also limited the interaction between the I-Unit Controller and the E-Unit Controller to normal and internal instructions only.

The I-Unit Controller consists of two finite state machines (Figure 2-2-1) and the Trap Logic consists of five random logic blocks (Figure 2-4-1). Their designs are simple finite state machines and random logic design problems and were discussed earlier in Section 2.2 and Section 2.4, respectively. Neither the I-Unit Controller nor the Trap logic involve further on-chip interaction consideration. The E-Unit Controller, on the other hand, must distribute the control information it generates to the micro-modules in the datapath. The E-Unit Controller does not have to distinguish internal instructions from normal instructions. It simply generates a set of control signals for each instruction it receives. The E-Unit Controller is therefore just a combinational logic block. The rest of this section will discuss strategies that can be used to reduce the interaction between the E-Unit Controller and the datapath.

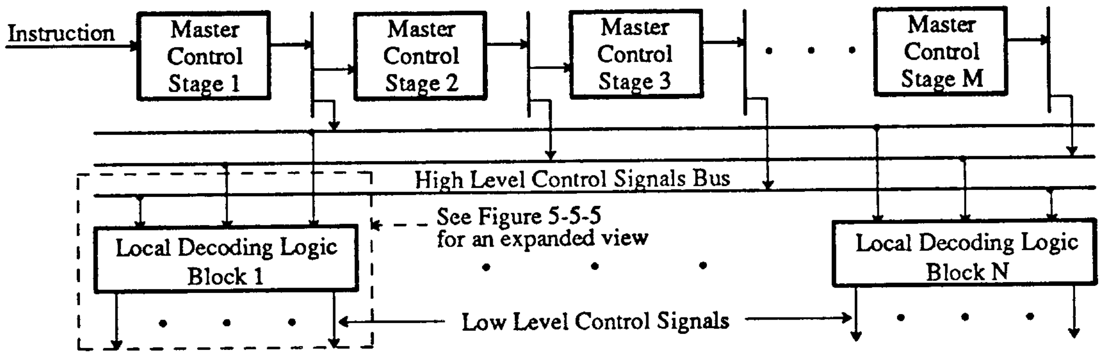

Instruction
Master Control Stage 1
Master Control Stage 2
Master Control Stage 3
Master Control Stage M
High Level Control Signals Bus
See Figure 5-5-5 for an expanded view
Local Decoding Logic Block 1
Local Decoding Logic Block N
Low Level Control Signals

Figure 5-5-4 The E-Unit Controller Bus Structure

The Master Control is divided into M stages where M is the length of the pipeline. Each stage generates one set of high level control signals that controls one stage of the pipeline. High level control signals are distributed via the high level control signal bus. The local decoding logic blocks then generates the low level control signals that control the datapath.

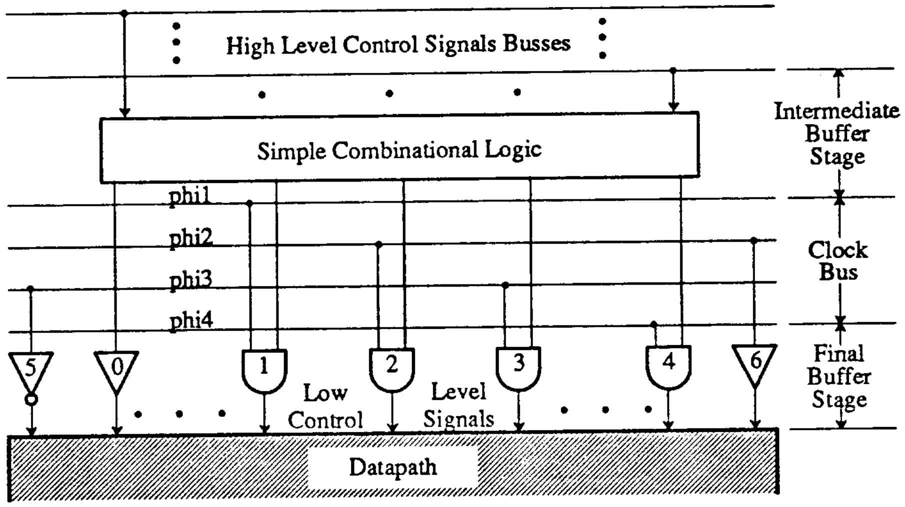

High Level Control Signals Busses
Simple Combinational Logic
phi1
phi2
phi3
phi4
5
0
1
2
3
4
6
Low Control
Level Signals
Datapath
Intermediate Buffer Stage
Clock Bus
Final Buffer Stage

Figure 5-5-5 Distribution of Control Information

The high level control signals are decoded into low level control signals by very simple combinational logic. Most outputs of the combinational logic must be ANDed with one of the clock phases (1, 2, 3, or 4) before being used by the datapath. For those outputs that do not have to be ANDed with any clock phase, they are buffered (0). Furthermore, in order to reduce the load on the clock generator, the clock signals are also buffered (5 and 6) before they are fed into the datapath.

Signal busses are used during the abstract stage to reduce the interactions among various macro-modules. Similarly, signal busses can also be used here to reduce the interaction between E-Unit Controller and the datapath. This is illustrated in Figure 5-5-4. The E-Unit Controller is divided into two parts: the Master Control and Local Decoding Logic. The master control is located far away from the datapath but the local decoding logic blocks are placed right next to the datapath. A signal bus is used to distribute the high level control signals generated by the master control to the local decoding logic blocks. This simplifies the on-chip interaction because the number of high level control signals are relatively small compare to the number of low level control signals. This reduction in number is due to sharing–each high level signal is used by more than one local decoding logic block.

The master control consists of M stages, where M is the length of the pipeline (see Figure 5-5-4). The first stage decodes the instruction (mainly the opcode) into the first set of high level control signals. The other stages uses the outputs of their previous stage as inputs. The major function of these latter stages is to delay the control signals by one cycle. One may also put some simple logic in the latter stages to combine some of their inputs. This is especially useful when instructions require different control signals at early stages of the pipeline but require the same control signal at latter stages.

The local decoding logic block consists of a block of simple combinational logic, a clock signals bus, and some AND, NOT, and BUF gates between the clock bus and the datapath. This is illustrated in Figure 5-5-5. Since CMOS logic gates can also serve as buffers, the simple combinational logic block also serves as the intermediate stage of the multi-stage control information distribution network between the E-Unit Controller and the datapath. Similarly, the AND, NOT, and BUF gates between the clock bus and the datapath serves as the final stage. The sizes of the buffers at the final stage should be selected according to the RC loading of the low level control signals and clock wires such that all the buffers will drive their outputs within approximately the same delay.

The control strategy shown in Figure 5-5-3, Figure 5-5-4, and Figure 5-5-5, was used in the SPUR CPU. This strategy resulted in a much more compact controller for the SPUR CPU (see Figure 1-4-3) than the controller for SOAR (see Figure 1-4-2). One important question concerning this strategy is that how the designer should divide the high and low level decoding. There must be a good balance between the number of high level control signals and the complexity of the local decoding logic. I do not have a satisfactory answer, but the SPUR CPU implementation do provide us some insights to this question. In the SPUR CPU, the combinational logic in the local decoding logic are single level logic and are mostly OR gates. This indicates that this is a generalized PLA problem with the the local decoding logic as the OR plane and the high level control signals as the product terms. In a more general view, one may also consider this as a multiple level logic optimization problem with the local decoding logic as the last logic level. This, however, is a more challenging problem for the CAD tools designer than the normal multiple level logic optimization problem for the following reasons:

- The area available for the combinational logic within the local decoding logic block are restricted by the spacing between the low level control signals.
- The optimum size of the simple combinational logic gates and the final buffers (see Figure 5-5-5) depends on the RC loading of the low level control signals.

5.5.3. The Expanded Model of the Microarchitecture

The expansion stage of the microarchitectural design process takes into account the micro-modules selection, resource allocation, and on-chip interaction. The result is the expanded model of the proposed microarchitecture. This is illustrated in Figure 5-5-6. The SPUR CPU behavioral

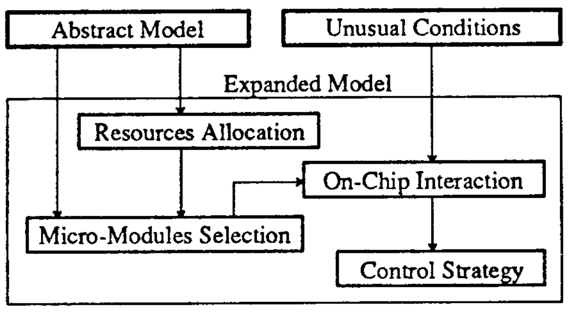


Figure 5-5-6 Building the Expanded Model

This is a flow chart on the construction of the expanded model. At this stage of the design, the microarchitect has an abstract model of the microarchitecture. He must also consider all the unusual conditions that can affect the normal operation of the microarchitecture. Macro-modules in the abstract model are then expanded into micro-modules according to the resource available and the functionality of the macro-modules. The microarchitect must also consider on-chip interaction carefully in order to derive a control strategy that controls the micro-modules and handles all the unusual conditions.

model (Figure 3-2-2) can be considered as the expanded model of the SPUR CPU. A complete expanded model is time consuming to build. Therefore, the microarchitect should not start building the expanded model until he has done enough primary investigation using the abstract model. The expanded model, however, can provide much better insights into the performance, resources, and complexity tradeoffs of the proposed microarchitecture than the abstract model.

5.5.3.1. Using the Expanded Model for Performance Estimation

In Chapter 4, I have shown that the performance of a microarchitecture should be measured in terms of the $T \times I \times C$ product. The expanded model of the CPU can be used to estimate the C–the average number of cycles per instruction, and the T–the cycle time more accurately than using the abstract model.

For example, in the N.2 environment where the expanded model of the SPUR CPU is simulated, test programs can be compiled and loaded into simulated memory. The SPUR CPU expanded model can then be simulated using the test programs that reside in the simulated memory. Since the expanded model models the details of the microarchitecture, it can be used to measure the number of cycles the CPU takes to execute certain test programs ($I \times C$) accurately. Unfortunately, the simulator for the expanded model can be relatively slow. For example, the SPUR CPU expanded model takes an average of 2 SUN3/160 CPU seconds to simulate each instruction! This will severely limit the size of test programs that can be simulated. However, if the test programs have the proper mix of instructions, the microarchitect can still calculate the average number of cycles per instruction (C) accurately by dividing the cycle count ($I \times C$) he measured by the number of instructions (I) in the test program.

In the N.2 environment, the SPUR CPU expanded model can be used to estimate the microarchitecture's cycle time in two different ways:

- The N.2 simulator can perform simple timing analysis if explicit timing information is provided for each micro-module, and

- The N.2 simulator can trace the significant sequential events within each clock phase. This list of events will give the microarchitect some insights on the duration of each clock phase.

Once the microarchitect has estimated the cycle time (T), and has calculated the average number of cycles per instruction (C), the last thing he has to do before he can predict the CPU's performance accurately is to get an estimate of the number of instructions the CPU takes to execute certain large benchmarks (I). This, of course, can be measured from the instruction level simulator.

5.5.3.2. Using the Expanded Model for Resources Estimation

As discussed in Section 3.2.1, the SPUR CPU behavioral model (the expanded model) is a composite module that consists of many modules connected by a top level topology file. The dimensions of each module can be estimated either based on previous experience or better yet based on the results of resource allocation analysis illustrated in Figure 5-5-2. The dimensions of each module can then be added to the topology file as comments. These dimensions together with the connection information in the topology file can be used by a designer assisted by CAD tools to create a tentative floor plan that gives a rough estimate of chip area. Floor planning is discussed in more details in Section 5.5.6.

The expanded model can also be used for power estimation. In CMOS where static power consumption is low, most of the power will be consumed in driving busses and off-chip output pads. All busses and output pads are simply internal and external connections in the top level topology file that describes the expanded model. Furthermore, if power consumption of certain modules is significant compared to busses and pads, this information can also be added to the topology file as comments. Therefore, power consumption can also be extracted easily from the expanded model.

5.5.3.3. Using the Expanded Model for Complexity Estimation

In Chapter 4, I have shown that the complexity of a microarchitecture can be measured in terms of the number of cycles of diagnostics and the human effort it takes to verify the microarchitecture. The expanded model is tested by test programs. The size of these test programs and the human effort involve in preparing and running these tests are therefore good measures of the microarchitecture's complexity. These, however, only measure one aspect of complexity which I called the functional complexity. Functional complexity measures the degree of difficulty in analysis, design, and testing of the microprocessor.

Implementation complexity is another aspect of complexity. Implementation complexity measures the degree of difficulty in implementing the microprocessor. The total number of micro-modules in the expanded model, the size of the micro-modules' description, the number of low level control signals that control the micro-modules, and the number of high level control signals generated by the master control are all important metrics for the implementation complexity.

5.6. The Floor Planning Stage of Microarchitectural Design

As the name implies, the goal of the floor planning stage is to produce a floor plan for the proposed microarchitecture. There are two important considerations in designing the floor plan:

- The interaction between the macro-modules.
- The relative dimensions of the macro-modules.

The macro-modules are the products of the abstract stage. The interaction between the macro-modules are also studied during the abstract stage. The relative dimensions of the macro-modules, however, depends on how they are expanded into micro-modules during the expansion stage. The floor planning stage of microarchitectural design can therefore be considered as the stage that summarizes the results from the abstract and expansion stage.

The interaction between the macro-modules and their relative dimensions can both be summarized in a graph. This is done in Figure 5-6-1 for the SPUR CPU as an example. Notice that Figure 5-6-1 is similar to Figure 5-4-7 except that I have added the relative dimensions ($height \times width$) for each macro-module. The goal here is to place the macro-modules that have a

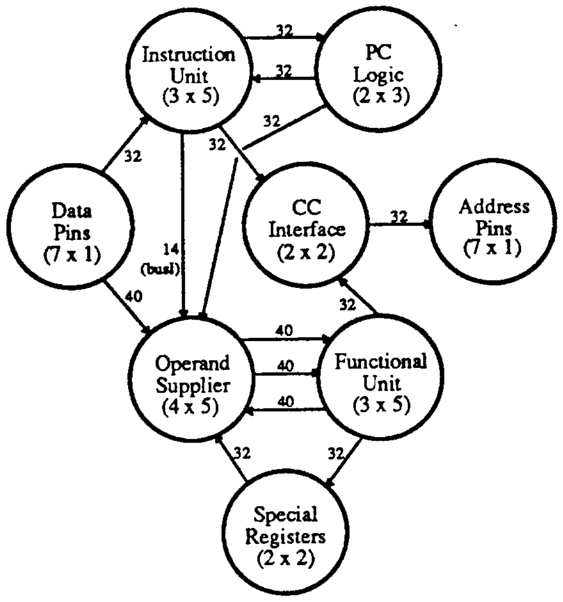


Figure 5-6-1 Important Information for Floor Planning

Each node in this graph represents a macro-module. The number within each node is the macro-module's relative dimensions (*height* × *width*). Each arc in this graph represents a set of signals that must be sent from one macro-module to the others. The number associated with each arc is the number of signals in the set.

large number of of connections between them close to each other while at the same time try to maintain the overall shape as rectangular as possible.

For example, based on the dimensions shown in Figure 5-6-1, we decided to place the Instruction Unit on top of the Operand Supplier because they are the biggest and longest. On the other hand, based on the interactions shown in Figure 5-6-1, we decided to place the PC Logic adjacent to the Instruction Unit and the Functional Unit adjacent to the Operand Supplier. After careful consideration of the rest of Figure 5-6-1, we selected the floor plan shown in Figure 5-6-2 for the SPUR CPU.

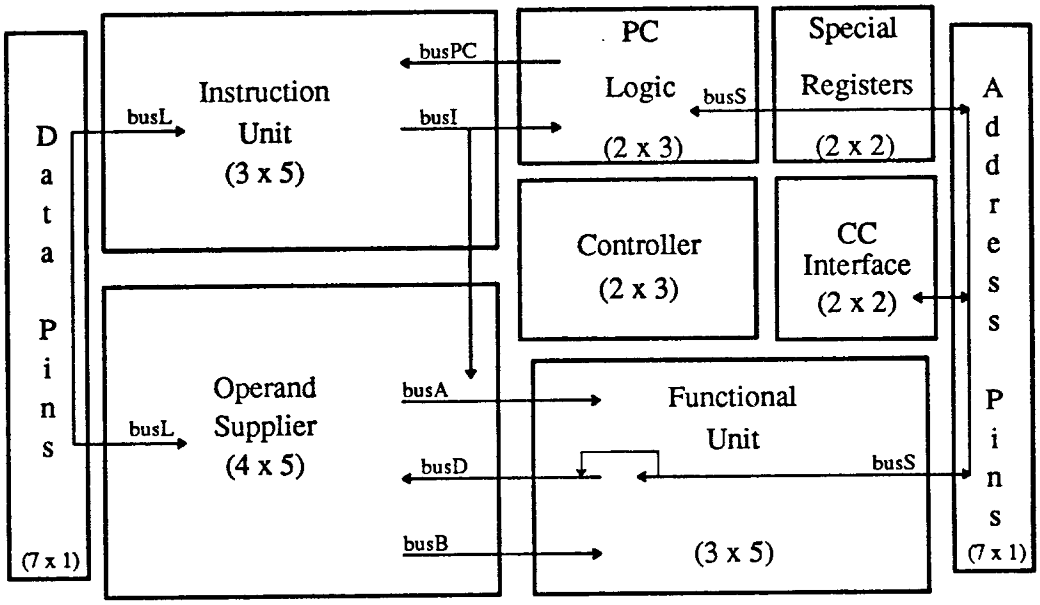


Figure 5-6-2 The SPUR CPU Floor Plan

This is the floor plan of the SPUR CPU. The relative dimensions of each macro-module are shown in parentheses. Relative dimensions are used instead of absolute dimensions such that a tentative floor plan can be produced even before the exact technology is known.

5.7. Conclusion

In this chapter, I first defined the term microarchitecture and the phrase "microarchitectural design." Based on my experiences in SPUR, I believed that the important issues concerning microarchitectural design are: (1) off-chip communication, (2) pipeline and clocking, (3) micro-modules selection, (4) resources allocation, (5) on-chip interaction, and (6) floor planning. Off-chip communication, and pipeline and clocking are behavioral issues. Micro-modules selection, on-chip interaction, and resources allocation are structural issues. Finally, floor planning is a physical issue.

A systematic approach to microarchitectural design must begin with a systematic approach to these microarchitectural issues. More specificly, a systematic approach to microarchitectural design consists of three stages:

(1) The Abstract Stage

Construct the abstract model (Figure 5-3-2) and then use it to conduct studies on microarchitectural issues: (a) off-chip communication, and (b) pipeline and clocking.

(2) The Expansion Stage

Construct the expanded model (Figure 5-3-3) by expanding the macro-modules and the controller in the abstract model. The microarchitectural issues to be studied here are: (a) micro-modules selection, (b) resources allocation, and (c) on-chip interaction.

(3) The Floor Planning Stage

Based on the information we learned from the abstract and expanded model, construct a detail floor plan for the microarchitecture.

These three stages are illustrated in Section 5.7.4, Section 5.7.5, and Section 5.7.6 using the SPUR CPU as an example.

5.8. REFERENCES

[Cam87] R. Camposano, ''Structural Synthesis in the Yorktown Silicon Compiler'', *VLSI 87*, 1987.

[Chu74] Y. Chu, ''Why Do We Need Computer Hardare Description Language'', *Computer 7*, 12 (December 1974).

[DRS86] H. DeMan, J. Rabaey, P. Six and L. Claesen, ''Cathedral-II: A Silicon Compiler for Digital Signal Processing'', *IEEE Design & Test of Computers*, December 1986.

[Gaj85] D. D. Gajski, ''Silicon Compilation'', *VLSI Systems Design, VI*, 11 (November 1985).

[GoH86] J. R. Goodman and W. C. Hsu, ''On the Use of Registers vs. Cache to Minimize Memory Traffic'', *The 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 2-5, 1986.

[HAD88] A. Hooton, U. Aguero and S. Dasgupta, ''An Exercise in Plausibility-Driven Design'', *Computer 21*, 7 (July 1988).

[Jeo88] D. K. Jeong, Private Communication EECS Department, University of California, Berkeley, CA 94720, July, 1988.

[Joh81] D. L. Johannsen, *Silicon Compilation*, Doctoral Dissertation, Department of Computer Science, California Institute of Technology, Pasadena, California, 1981.

[Kel85] J. Keller, *Power and Ground Requirements for a High-speed 32 Bit Computer Chip Set*, Master Report, EECS Department, University of California, Berkeley, CA 94720, August, 1985.

[Kog81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981.

[Kon85] S. Kong, *Some Design Techniques for High-Performance MOS Circuits*, Master Report, EECS Department, University of California, Berkeley, CA 94720, January 1985.

[Kow85] T. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, 1985.

[Lip75] G. J. Lipovski, ''On Gray Box Description of Microprocessors'', *Proceedings International Symposium on CHDL's and Their Applications*, 1975, 184-186.

[McH86] S. McFarling and J. Hennessy, ''Reducing the Cost of Branches'', *The 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 2-5, 1986.

[NeS86] A. R. Newton and A. L. Sangiovanni-Vincentelli, ''Computer-Aided Design for VLSI Circuits'', *Computer 19*, 4 (April 1986).

[PaS80] D. A. Patterson and C. H. Sequin, ''Design Considerations for Single-Chip Computers of the Future'', *IEEE Journal of Solid-State Circuits SC-15*, 1 (February 1980).

[SMH85] W. Scott, R. Mayo, G. Hamachi and J. Ousterhout, editors. ''1986 VLSI Tools: Still More Works by the Original Artists'', Report No. UCB/Computer Science Dpt. 86/272, Computer Science Division, EECS Department University of California, Berkeley, CA 94720, December 1985.

[Su77] S. Su, ''Hardware Description Language Applications: An Introduction and Prognosis'', *Computer 10*, 6 (June 1977).

[WaS86] D. E. Wallace and C. H. Sequin, ''Plug-In Timing Models for An Abstract Timing Verifier'', *23rd Design Automation Conference*, Las Vegas, Nevada, 1986.

Chapter 6

SUMMARY AND FUTURE TRENDS

I never think of the future. It comes soon enough.

Albert Einstein, 1930

This chapter first summarizes this thesis and the SPUR project in Section 6.1. In Section 6.2, I discuss what I think will be the future trends based on the lessons I learned.

6.1. Summary

Section 6.1.1 summarizes this thesis. Section 6.1.2 reviews the history of the SPUR project in the SPUR CPU's perspective. Section 6.1.3 discusses the organization of the SPUR project.

6.1.1. Thesis Summary

In Chapter 1 and Chapter 2, I gave a brief history of VLSI projects at U.C. Berkeley, an overview of the SPUR project, and an overview of the SPUR CPU microarchitecture. One of the most important difference between the SPUR CPU and the previous two generations of Berkeley RISC projects is that the goal of the SPUR project is not just to build a CPU. The SPUR project's goal is to build a system in which the SPUR CPU is just one of the three custom VLSI chips.

In Chapter 3, I first talked about how we used proven ideas from the two previous generations of Berkeley RISC machines to design the SPUR CPU microarchitecture. Just because we used proven ideas does not mean our job of building a chip for a system is easy. We still have to deal with many problems that are not as important when one just wants to build a CPU. Two specific examples are:

(1) We must implement some system features that require a lot of work, and

(2) We must do many extra simulations.

One important lesson I learned in dealing with all these problems is that we must make the process of designing microprocessor more like a science than an art.

In Chapter 4, I stated that the designer can make the process of designing microprocessor more like a science than an art by putting more emphasis on quantitative evaluation of the performance, resources, and complexity tradeoffs. I also showed a simple performance model and then performed tradeoffs evaluation for LISP support, floating point support, 4-stage pipeline, on-chip instruction cache, and multiprocessing support. One major conclusion from Chapter 4 is that the designer must keep the cycle time and the average number of cycles per instruction as low as possible.

Finally in Chapter 5, I introduced a systematic approach to microarchitectural design that consists of three stages: (1) the abstract stage, (2) the expansion stage, and (3) the floor planning stage. During the abstract stage, we build the abstract model of the microarchitecture to study the off-chip communication, and pipeline and clocking issues. During the expansion stage, the abstract model is expanded. The major issues to be studied during the expansion stage are micro-modules selection, resources allocation, and on-chip interaction. Finally, during the floor planning stage, we applied what we learned from the abstract and expansion stage and design a floor plan for the microarchitecture.

6.1.2. The History of the SPUR Project

The SPUR project is probably one of the most ambitious computer projects ever accomplished in the university environment. The history of the SPUR project is described below with respect to the SPUR CPU development.

Spring 1985

The SPUR CPU basic microarchitecture was conceived in the CS292i class taught by Professor Randy Katz [Kat85].

Summer 1985

The CPU's external interfaces to the cache controller (CC), the coprocessor (FPU), and the processor board were defined.

Fall 1985

We began the datapath layout and started writing the behavioral description that models the microarchitecture.

Spring 1986

The layout of the datapath was completed.

Summer 1986

We began the layout of the control unit.

Fall 1986

The layout of instruction unit was completed. At the same time, we also completed the behavioral description.

Spring 1987

Logic simulation and timing analysis of individual modules was carried out.

Summer 1987

Global simulation was completed. The SPUR CPU was submitted for fabrication on August 25. It came back in December 1987.

Spring 1988

We tested and debugged the SPUR CPU. The second version of the CPU was submitted for fabrication in April. It came back in June 1988.

Summer 1988

We tested and debugged the SPUR processor board. The Spirte [OCD88] operating system was running on SPUR hardware by the end of the summer.

Fall 1988

We concentrated our effort in multiprocessor testing and debugging.

Spring 1989

On January 9, 1989, a SPUR multiprocessor running Spirte operating system and LISP software was presented at U.C. Berkeley.

This ambitious four-year project involve a large group of professors and graduate students. Section 6.1.3 shows the organization of the SPUR project.

6.1.3. The Organization of the SPUR Project

Professor Dave Patterson was the principal investigator of the SPUR project. The SPUR project was organized into three groups: (1) the hardware group, (2) the operating system group, and (3) the programming language group.

The Hardware Group. Professor Randy Katz and Professor David Hodges were in charge of the hardware group with Professor Randy Katz on architectural design and Professor David Hodges on circuit design. The hardware group was furthered divided into four groups:

The CPU Group

Mark Hill and George Taylor were responsible for the macroarchitecture of the CPU. I was responsible for the microarchitecture. Dave Lee, with the help of Rich Duncombe (initial implementation of the Instruction Unit) and Wook Koh (initial implementation of the Upper Datapath) were responsible for the circuit design and layout.

The Cache Controller Group

David Wood, Garth Gibson, and Susan Eggers were responsible for the macroarchitecture and microarchitecture of the Cache Controller. D.K. Jeong was responsible for the circuit design and layout.

The Floating Point Unit Group

B.K. Bose, Paul Hansen, and Corina Lee were were responsible for all aspects of the Floating Point Unit design.

The Processor Board Group

Our staff engineer Ken Lutz with the help of Kathy Armstrong were responsible for the design and implementation of the SPUR processor board.

The hardware group also got very valuable help form Joan Pendleton during initial stage of the SPUR project and Doug Johnson–an engineer from Texas Instruments–during the final stage of the project. The U.C. Berkeley CAD research community also gave us constant support.

The Operating System Group. Professor John Ousterhout was in charge of developing the operating system Spirte for the SPUR multiprocessor. The graduate students who worked in the operating system group were Michael Nelson, Brent Welch, Fred Douglis, Andrew Cherenson, and Mendel Resenblum.

The Programming Language Group. Professor Paul Hillfinger was in charge of developing the LISP system [Tay86] [ZHH88] for the SPUR multiprocessor. The graduate students who worked in the programming language group were Jim Larus and Ben Zorn.

6.2. Future Trends

In this section, I want to say a few words about what I think the future trends are in the architectural, technology, and CAD support areas. In my performance evaluation, I have shown quantitatively in Section 4.7.2 that a simple architecture can surpass the performance of a more complex architecture by keeping the average number of cycles per instruction and the cycle time low. Therefore, I think the architectural trend is to reduce the average number of cycles per instruction and the technology trend is to lower cycle time. Both of these can be accomplished more easily if CAD support is readily available.

6.2.1. Architectural Trends

The architectural trend is to reduce the average number of cycles per instruction. The average number of cycles can be reduced by the following methods:

(1) Improve the hit rate of the on-chip instruction cache. Mark Hill [Hil87] has studied better instruction cache design and implementation ideas.

(2) When more on-chip transistors are available, I think on-chip data cache is more desirable than complex functions. In Chapter 4, I have shown that it is more cost effective to support complex functions via a coprocessor.

(3) Reduce the branch penalty. Branch folding in CRISP [BDM87] is one example where in the best scenario, a branch is executed implicitly with other instructions. This reduces branch penalty to zero.

(4) Finally one may try to put multiple functional units on chip such that multiple instructions can be executed per cycle.

The first two methods, even in the best scenario, can only reduce the average number of cycles per instruction to one. The third and fouth methods, on the other hand, can reduce the average number of cycles per instruction to less than one.

6.2.2. Technology Trends

The technology trend is to reduce the cycle time. The cycle time can be reduced by:

(1) Scaling down the CMOS technology. Common belief is that device width of 0.25μm is the practical limit and it will be reached in the 1990s [MYH86].

(2) ECL is faster than CMOS but it also has lower density and uses more power.

(3) BICOMS takes advantages of both bipolar and CMOS circuits on the same chip and looks very promising.

(4) GaAs is very fast, uses less power than ECL, but it is very expensive in terms of wafer cost, yield, and density.

One thing I want to point out is that although ECL logic gates use more power than GaAs and CMOS logic gates, this may become less important in the future. I believe in the future most of the power will not be consumed in the logic gates. Most of the power will be consumed in driving the internal busses and off-chip pads. For example in the SPUR CPU, we estimated that 60% of the total power consumption is spent in driving the off-chip pads and 20% is spent in driving the on-chip busses already. These numbers are going to get worst when the feature size gets smaller and when the CPU runs at higher clock rate because:

- As feature sizes gets smaller, the capacitance of the on-chip busses gets relatively bigger–more power is needed to drive them at high speed.
- As the CPU runs faster, the off-chip communication channel must also run faster–more power is needed to drive the off-chip pads.

GaAs uses field effect transistors that are similar to MOS transistors use in CMOS. Although Seymour Cray suggested that GaAs discrete component has low capacitance [Cra88], this unfortunately may not apply to VLSI application. In VLSI, capacitance of the multi-level interconnect network is the dominant factor. This multi-layer interconnect capacitance depends on the materials that separates the interconnect layers and does not depend on the substrate material. Therefore GaAs is likely to have similar power consumption problems as CMOS. ECL, on the other hand, uses bipolar transistors that have much bigger current driving capability and smaller voltage swings than field effect transistors. Consequently, ECL will have less problem driving highly capacitive internal busses and off chip pads. Finally, one should realize that a fast CPU must run in a fast environment. The power consumption of this fast environment can make the power consumption of the CPU negligible.

6.2.3. CAD Support Trends

I believe CAD researchers will continue their emphasis in building automatic layout generators–silicon compilers. Although silicon compliers are useful, I do not think they will solve all the problems because the designer still have to define the microarchitecture as inputs to the silicon compiler. My experience in SPUR has convinced me the following:

- Defining a microarchitecture is not a simple task. Therefore, we need more high level design tools that can help designer make quantitative tradeoffs decisions.
- Moving to the future means back to the basic! Mead and Conway design style is not enough. Building a high performance VLSI chip is still an electrical engineer's job. We need electrical rules checkers that understand resistance, capacitance, and inductance.
- I think VLSI designers can do CAD tool designers a big favor by building simple tools. No matter how simple the tool is, I think it is still the best way to define the problem formally.

As I stated in Section 3.5, I do not believe CAD tools are there to replace VLSI designer. However, I do believe the future of VLSI design depends strongly on CAD support. One contrasting view is from Nick Tredennick [Tre87]:

> This book is partly in response to the growing presumption that computers are an essential part of logic design. They are not. ... I think of computers as an expensive and awkward alternative to pencil and paper.
>
> Nick Tredennick, Microprocessor Logic Design, Page 4

This may be true for a talented VLSI artist such as Nick. However, for an average VLSI engineer like myself, CAD tools are essential. As a matter of fact, I and my colleagues in SPUR have promised ourselves not to design another VLSI chip unless we have more CAD support for all aspects of the design. I think the introduction of CAD support to VLSI design is analogous to the introduction of jet engine to aviation at the end of World War II.

CAD support enable VLSI designer moves much faster but it also takes some of the art out of VLSI design. Similarly, a jet engine enables aircraft to fly faster but it also takes some of the art out of aviation. In many ways, a jet aircraft is easier to fly than a propeller driven aircraft because a spinning propeller creates many mysterious effects that are handled by pilots more like an art than a science. Furthermore, jet engine also enables aircraft to fly much higher to avoid most bad weather. Consequently, when the jet engine was first introduced, many "real aviators" insisted flying propeller driven aircraft was still the only true art of aviation. They may be right. After all, how can you argue with an artist? However, most people probably prefer getting to their destination in one hour–in a jet aircraft piloted by just an average pilot–than getting to their destination in four hours in a propeller driven aircraft, piloted by a "real aviator."

6.3. REFERENCES

[BDM87] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan, ''Architectural Innovations in the CRISP Microprocessor '', *COMPCON 87*, San Francisco, California, February 23-27, 1987.

[Cra88] S. Cray, ''What's All These Fuss About Gallium Arsenide'', *Keynote Address, Super Computing*, Orlando, Florida, November, 1988.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Dissertation, Computer Science Division, EECS Department University of California, Berkeley, Fall 1987.

[Kat85] in *Proceedings of CS 292i: Implementation of VLSI Systems*, R. Katz (editor), Computer Science Division, EECS Department, University of California, Berkeley, September 1985.

[MYH86] G. J. Myers, A. Y. C. Yu and D. L. House, ''Microprocessor Technology Trends'', *Proceedings of the IEEE 74*, 12 (December 1986).

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, ''The Sprite Network Operating System'', *Computer 21*, 2 (February 1988).

[Tay86] G. Taylor et al., *Evaluation of the SPUR Lisp Architecture*, The 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 2-5, 1986.

[Tre87] N. Tredennick, *Microprocessor Logic Design*, Digital Press, 1987.

[ZHH88] B. Zorn, P. Hillfinger, K. Ho, J. Larus and L. Semenzato, ''Features for Multiprocessing in SPUR LISP'', Report No. UCB/Computer Science Dpt. 88/406, Computer Science Division, EECS Department, University of California, Berkeley, March 1988.

Appendix A

DETAILED DESCRIPTION OF THE SPUR CPU MICROARCHITECTURE

Architecture is the art of how to waste space.

Philip Johnson, 1964

A.1. The SPUR CPU Block Diagram

Figure A-1-1 is the detailed SPUR CPU block diagram. This block diagram shows the relative position of each block in the layout. The following naming conventions are used in this block diagram:

- Register names start with an upper case letter and the rest are lower case except to improve readability. Examples: Dst1 and IfetPC.
- Functional block names are in upper case letters only. Example: ALU and EXT_INS.
- Signal names start with a lower case letter and the rest are lower case except to improve readability. Examples: busA and trapType.

The CPU can be divided into two units: (1) the Instruction Unit (I-Unit) at the upper left corner and (2) the Execution Unit (E-Unit) at the rest of the area. The Instruction Unit (see Section 2.2) is a 512-byte direct-mapped instruction cache. The Execution Unit (see Section 2.3) consists of 4 parts:

(1) The lower datapath performs all the register-to-register operations.

(2) The upper datapath performs all the program control operations.

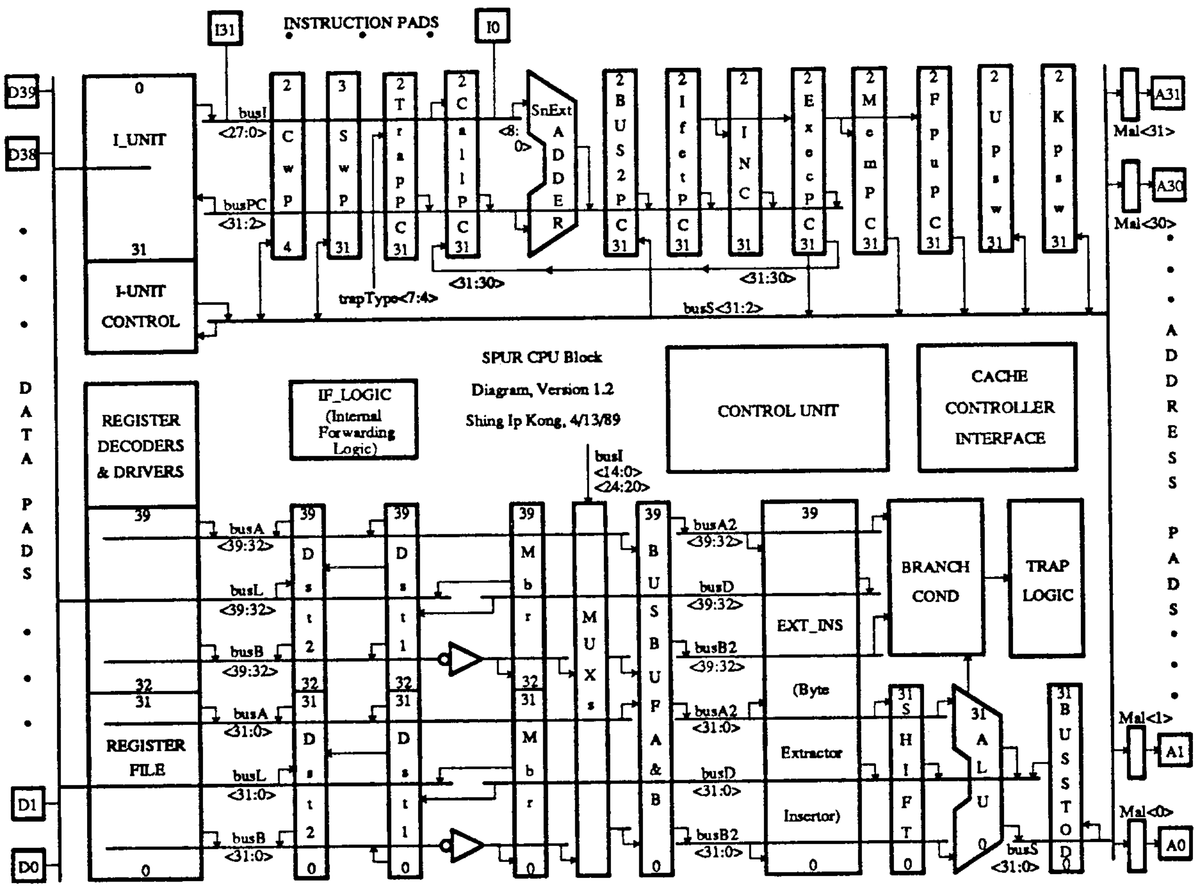


Figure A-1-1 The SPUR CPU Block Diagram

The datapath is split into two parts (see Section 2.3): The upper datapath at the top and the lower datapath at the bottom. The CONTROL UNIT and the CACHE CONTROLLER INTERFACE are in the middle. The upper and lower datapaths are connected by busS. There are three major sets of IO pads: The DATA PADS on the left, the ADDRESS PADS on the right, and the INSTRUCTION PADS on the top. The INSTRUCTION PADS are part of the coprocessor interface. The coprocessor (FPU) must monitor these pads continuously to detect any instruction it has to execute.

(3) The cache controller interface communicates with the cache controller chip.

(4) And the control unit controls the Execution Unit.

The lower datapath contains a 138 word-register file, some temporary registers, and several functional units. It is 40 bits wide because 8 of the bits are used for tags. The upper datapath contains some special registers and the program counters logic. It is 30 bits wide because all instructions are word addressed. The CPU chip resides inside a 208-pad pad frame. The CPU only needs about 180 pads but the same pad-frame is used by all three SPUR custom chips.

A.2. The SPUR CPU Register Set

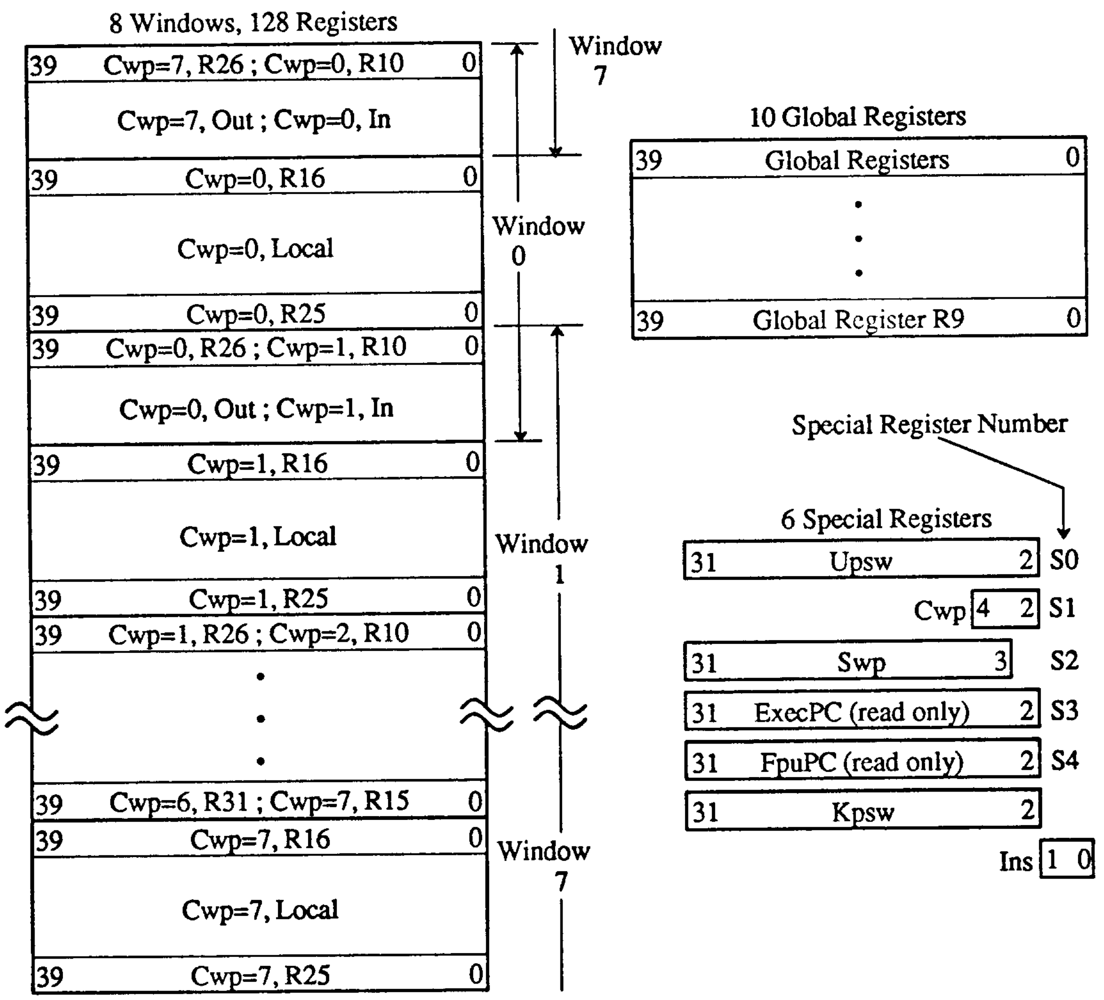


Figure A-2-1 The SPUR CPU Registers Set

Each register window has ten local registers, six input registers, and six output registers. The input and output registers of adjacent windows overlap and are used for parameters passing. Special register numbers are used by instructions RD_SPECIAL and WR_SPECIAL to specify the source (Ss1) and destination (Sd) special registers, respectively. Kpsw and Ins do not have any special register number because they are not manipulated by the RD_SPECIAL nor the WR_SPECIAL instruction.

The SPUR CPU register set (Figure A-2-1) consists of 138 general purpose registers and seven special registers. The 138 general purpose registers are organized into 10 global registers and eight overlapping register windows. The seven special registers are:

Cwp<4:2>	Current register window pointer. Points to the register window that is currently in use.
Swp<31:3>	Save register window pointer. Points to the memory location where the last overflow register window (points to by Swp<9:7>) is saved.
Kpsw<31:2>	Kernel processor status word.
Upsw<31:2>	User processor status word.
Ins<1:0>	Insert byte count register.
ExecPC<31:2>	Program counter contains the address of the instruction currently being executed. This is a read only register.
FpuPC<31:2>	Program counter contains the address of the last floating point instruction send to the FPU coprocessor. This is a read only register.

Whenever a CALL instruction is executed, Cwp is incremented by one and a new window is opened. Conversely, whenever a RETURN instruction is executed, Cwp is decremented by one and the window is closed. When window overflow occurs, register windows are saved to memory. Swp contains the memory address at which the last register window is saved. The status of the SPUR CPU is stored in the two processor status words: Kpsw and Upsw. Both the Kpsw and the Upsw are 30 bits wide. However, as shown in Figure A-2-2, only a small number of these 60 bits are used by the hardware. The rest of the bits are used by software to store relevant process information. The Ins register is used by the INSERT instruction to decide where within a word should the byte be inserted. Special registers ExecPC and FpuPC are read only and writing to them are the same as NOOP.

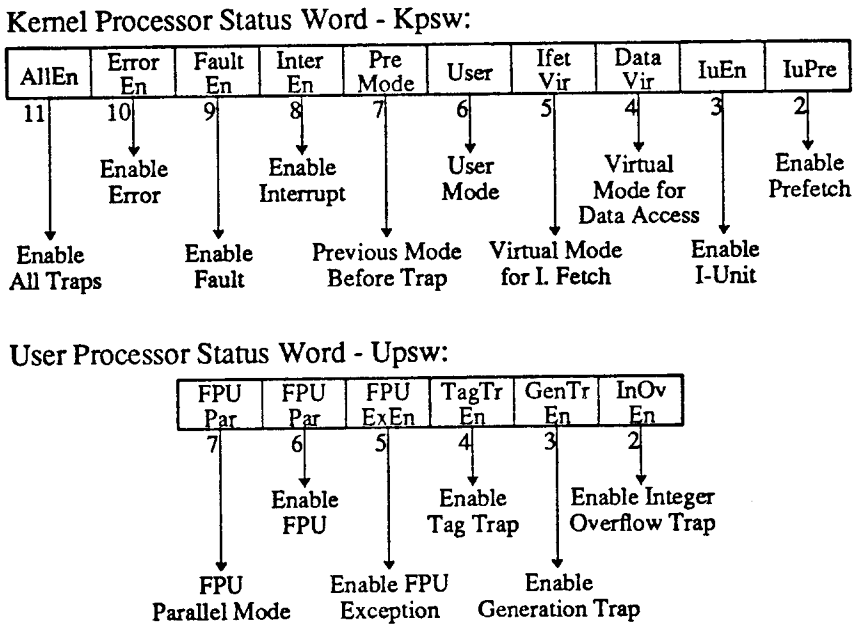


Figure A-2-2 Upsw and Kpsw Bit Assignments

Each bit's definition specified in this figure is the meaning of the bit when it is equal to 1. As shown in Figure A-2-1, Upsw<31:2> and Kpsw<31:2> are 30 bits wide. This figure only shows the bits that are used by the hardware. Bits that are not used by the hardware can be read and written by software.

A.3. The SPUR CPU Instruction Set

The SPUR CPU instruction set [Tay85] can be divided into seven types of instructions: (1) Load, (2) Register–Register, (3) Jump–Register and Return, (4) Read and Write Special Registers (5) Store, (6) Compare–Branch, and (7) Call–Jump. These instructions are summarized in Tables A-3-1 through A-3-7. Floating point instructions are not discussed in this section and can be found in [Bos88]. As mentioned in Section 2.4.1, unusual conditions can arise during instruction execution and may cause a trap. These unusual conditions are listed in Table A-3-8. Finally, Table A-3-9 shows all the branch conditions for all the Compare–Branch instructions.

Load, Jump–Register and Return, and Read and Write Special Registers instructions all have the same format as Register–Register instructions. The Ri field of Register–Register

Load Instructions				
Instruction	**Operands**	**Cache Operations**	**Action**	**Unusual Conditions (Table A-3-8)**
LD_40 LD_40_RO	Rd, Rs1, Ri	RD64 RFO64	Rd<39:0> <– Mem[Rs1+Ri]	None
CXR CXR_RO	Rd, Rs1, Ri	RD64 RFO64	Rd<39:0> <– Mem[Rs1+Ri] LISP pointer check	D
LD_32 LD_32_RO LD_32_RI	Rd, Rs1, Ri	RD32 PR32 RO32 RA32	Rd<31:0> <– Mem[Rs1+Ri] Rd<39:32> <– 0x00	None
TEST_&_SET	Rd, Rs1, Ri	TS32	Rd<31:0> <– Mem[Rs1+Ri] Rd<39:32> <– 0x00	None
LD_ EXTERNAL	Rd, Rs1, Ri	RD_ CACHE	Rd<31:0> <– CC Reg[Rs1+Ri] Rd<39:32> <– 0x00	E

Table A-3-1 Load Instructions

instructions can either be a register specified by the Rs2 field or the sign extension of the 14-bit immediate field (see Figure 2-1-1). In order to support the Berkeley Ownership cache consistency protocol [KEW85], most Load instructions have two favors–simple read or read for ownership (opcode_RO). LD_32 has one more favor–the RI favor. LD_32_RI tells the Cache Controller to ignore any page fault it may cause and provide the data to the CPU anyway. LD_32 is also the only Load that can access data in physical mode. Therefore LD_32's Cache Operations can either be RD32 (virtual mode) or PR32 (physical mode). TEST_&_SET and LD_EXTERNAL are similar to LD_32 as far as the CPU is concerned. The only difference is their Cache Operations which will be handled differently by the Cache Controller. A complete explanation of all the Cache Operations is given in [WEG87].

The Store instructions do not have the same format as the Register–Register instructions. Its immediate field (Imm) is the sign extension of the 14-bit immediate filed formed by concatenating the High Imm and Low Imm fields of the instructions (see Figure 2-1-1). Since ST_32 is the

Register–Register Instructions			
Instruction	**Operands**	**Action**	**Unusual Conditions (Table A-3-8)**
ADD_NT	Rd, Rs1, Ri	Rd<31:0> <– Rs1 + Ri Rd<39:32> <– Rs1<39:32>	None
ADD	Rd, Rs1, Ri	Rd<31:0> <– Rs1 + Ri Rd<39:32> <– Rs1<39:32>	F & I
SUB	Rd, Rs1, Ri	Rd<31:0> <– Rs1 - Ri Rd<39:32> <– Rs1<39:32>	F & I
AND	Rd, Rs1, Ri	Rd<31:0> <– Rs1 and Ri Rd<39:32> <– Rs1<39:32>	F
OR	Rd, Rs1, Ri	Rd<31:0> <– Rs1 or Ri Rd<39:32> <– Rs1<39:32>	F
XOR	Rd, Rs1, Ri	Rd<31:0> <– Rs1 xor Ri Rd<39:32> <– Rs1<39:32>	F
SLL	Rd, Rs1, Ri	Rd<31:0> <– Rs1<31:0> shift left by Ri<1:0> bits Rd<39:32> <– Rs1<39:32>	F
SRA	Rd, Rs1, Ri	Rd<31:0> <– Rs1<31:0> arithmetic shift right by Ri<0> bit Rd<39:32> <– Rs1<39:32>	F
SRL	Rd, Rs1, Ri	Rd<31:0> <– Rs1<31:0> logic shift right by Ri<0> bit Rd<39:32> <– Rs1<39:32>	F
RD_TAG	Rd, Rs1	Rd<31:8> <– 0 Rd<7:0> <– Rs1<39:32>	None
EXTRACT	Rd, Rs1, Ri	Rd<31:8> <– 0 Rd<7:0> <– Rs1[byte Ri<1:0>]	None
WR_TAG	Rd, Rs1, Ri	Rd<31:0> <– Rs1<31:0> Rd<39:32> <– Ri<7:0>	None
INSERT	Rd, Rs1, Ri	Rd[byte Ins<1:0>] <–Ri<7:0>	None

Table A-3-2 Register–Register Instructions

only instruction that can perform store in physical mode, ST_32's Cache Operations can either be WR32 (virtual mode) or PW32 (physical mode).

Compare–Trap is a special case of Compare–Branch in which the "branch" is taken as a trap. Compare Branch or Compare Trap's 3rd operand can either be Rc, Rs2, or Tag Imm depending on the Cond filed. The Rc option means the operand can either be a register specified

Jump–Register and Return Instructions			
Instruction	**Operands**	**Action**	**Unusual Conditions (Table A-3-8)**
JUMP_REG	Rs1, Ri	PC <– Rs1 + RC	None
RETURN	Rs1, Ri	PC <– Rs1 + Ri Pop to previous window: Cwp <– Cwp - 1	B
RETURN_ TRAP	Rs1, Ri	PC <– Rs1 + Ri Pop to previous window: Cwp <– Cwp - 1 Enable all traps: Kpsw<AllEn> <– 1	B

Table A-3-3 Jump–Register and Return Instructions

by the Rs2 field or zero extension of the 5-bit Short Immediate field (see Figure 2-1-1).

Read and Write Special Registers Instructions			
Instruction	**Operands**	**Action**	**Unusual Conditions (Table A-3-8)**
RD_SPECIAL	Rd, Ss1	Rd<31:0> <– Ss1 Rd<39:32> <– 0x00	None
RD_INSERT	Rd	Rd<31:0> <– Ins Rd<39:32> <– 0x00	None
RD_KPSW	Rd	Rd<31:0> <– Kpsw Rd<39:32> <– 0x00	None
WR_SPECIAL	Sd, Rs1, Ri	Sd <– Rs1 + Ri	None
WR_INSERT	Ri	Ins <– Ri<1:0>	None
WR_KPSW	Rs1 Ri	Kpsw <– Rs1 + Ri	E
INVALID_IB		Invalidate all entries in the on-chip instruction cache	None

Table A-3-4 Read and Write Special Registers Instructions

Ss1 and Sd are specifiers that specify the special registers to be read or written.

Store Instructions				
Instruction	Operands	Cache Operations	Action	Unusual Conditions (Table A-3-8)
ST_40	Rs2, Rs1, Imm	WR64	Mem[Rs1+Imm] <– Rs2<39:0> Generation Check	H
ST_32	Rs2, Rs1, Imm	WR32 PW32	Mem[Rs1+Imm] <– Rs2<31:0>	None
ST_ EXTERNAL	Rs2, Rs1, Imm	WR_ CACHE	CC Reg[Rs1+Imm] <– Rs2<31:0>	E

Table A-3-5 Store Instructions

Compare–Branch Instructions				
Instruction	Cond Field (Table A-3-9)	Operands	Action	Unusual Conditions (Table A-3-8)
CMP_BRANCH	always, ge, ne, gt, never, lt, eq, le, uge, ugt, ult, ule	Cond, Rs1, Rc, Offset	if (Cond=TRUE) PC <– PC + signExt(Offset)	G
CMP_BRANCH	eq_tag, eq_38, ne_tag, ne_38	Cond, Rs1, Rs2, Offset	if (Cond=TRUE) PC <– PC + signExt(Offset)	None
CMP_BRANCH	eq_tag_imm, ne_tag_imm,	Cond, Rs1, Tag Imm, Offset	if (Cond=TRUE) PC <– PC + signExt(Offset)	None
CMP_TRAP	always, ge, ne, gt, never, lt, eq, le, uge, ugt, ult, ule	Cond, Rs1, Rc, Offset	if (Cond=TRUE) Take a trap!	G, J
CMP_TRAP	eq_tag, eq_38, ne_tag, ne_38	Cond, Rs1, Rs2, Offset	if (Cond=TRUE) Take a trap!	J
CMP_TRAP	eq_tag_imm, ne_tag_imm,	Cond, Rs1, Tag Imm, Offset	if (Cond=TRUE) Take a trap!	J

Table A-3-6 Compare–Branch Instructions

Call–Jump Instructions			
Instruction	Operands	Action	Unusual Conditions (Table A-3-8)
JUMP	Word Address	PC <– PC<31:30> cat Word Address	None
CALL	Word Address	PC <– PC<31:30> cat Word Address Open new window: Cwp <– Cwp + 1 Save PC: R10 (new window) <– PC	A

Table A-3-7 Call–Jump Instructions

Unusual Conditions	Definition and Condition
A	**Window Overflow:** Attempt to execute CALL when Cwp+1 = Swp<9:7>
B	**Window Underflow:** Attempt to execute RETURN or RETURN_TRAP when Cwp-1 = Swp<9:7>
D	**LISP Pointer Type Violation:** Rs1<37:32> != CONS or NIL
E	**Kernel Mode Violation:** Attempt to execute a privilege instruction when Kpsw<UserBit> = 1
F	**LISP Data Type Violation:** Rs1<37:32> != FIXNUM or Rs2<39:32> != FIXNUM
G	**LISP Data Type Violation:** (Rs1<37:32> != FIXNUM or Rs2<39:32> != FIXNUM) and (Rs1<37:32> != CHARACTER or Rs2<39:32> != CHARACTER)
H	**Generation Violation:** Rs2<39:38> > Rs1<39:38>
I	**Integer Overflow**
J	**Compare_trap with valid condition**

Table A-3-8 The SPUR CPU Unusual Conditions

Mnemonic	Binary (Hex.)	Branch Conditions			Notes
ALWAYS	00 000 (00)	Always		Branch	
GE	00 001 (01)	Rs1<31:0>	≥	Rc<31:0>	1,2
NE	00 010 (02)	Rs1<31:0>	≠	Rc<31:0>	1,2
GT	00 011 (03)	Rs1<31:0>	>	Rc<31:0>	1,2
NEVER	00 100 (04)	Never		Branch	
LT	00 101 (05)	Rs1<31:0>	<	Rc<31:0>	1,2
EQ	00 110 (06)	Rs1<31:0>	=	Rc<31:0>	1,2
LE	00 111 (07)	Rs1<31:0>	≤	Rc<31:0>	1,2
UGE	01 001 (09)	Rs1<31:0>	≥	Rc<31:0>	3
UGT	01 011 (0B)	Rs1<31:0>	>	Rc<31:0>	3
ULT	01 101 (0D)	Rs1<31:0>	<	Rc<31:0>	3
ULE	01 111 (0F)	Rs1<31:0>	≤	Rc<31:0>	3
FPU_TRUE	10 000 (10)	fpuBrT_F_C4	=	1	4
EQ_TAG	10 001 (11)	Rs1<37:32>	=	Rs2<37:32>	5
EQ_38	10 011 (13)	Rs1<37:0>	=	Rs2<37:0>	5
FPU_FALSE	10 100 (14)	fpuBrT_F_C4	=	0	4
NE_TAG	10 101 (15)	Rs1<37:32>	≠	Rs2<37:32>	5
NE_38	10 111 (17)	Rs1<37:0>	≠	Rs2<37:0>	5
EQ_TC	11 001 (19)	Rs1<37:32>	=	Tag_Imm	5
NE_TC	11 101 (1D)	Rs1<37:32>	≠	Tag_Imm	5

Table A-3-9 The SPUR CPU Branch Conditions

Notes:

1. If busI<14> = 0, Rc = Rs2. Otherwise, Rc = Zero Ext (Short Imm).
2. Rs1 and Rc are treated as 2's complement signed integers.
3. Rs1 and Rc are treated as unsigned integers.
4. fpuBrT_F_C4 is an external input coming from the FPU.
5. Only the type tag are checked. Generation numbers are ignored.

A.4. Special Cases of Register–Register Instructions

Load, Jump–Register and Return, Read Special Registers, and Write Special Registers instructions can be considered as special cases of the Register–Register instructions. The timing of Load operation is shown in Table A-4-1. It is similar to Register–Register operation except the ALU output is sent out as effective address and the data from memory is written into the destination register. The timing of Jump–Register and Return operations is shown in Table A-4-2. In this case, the ALU output is sent to the upper datapath and then the Instruction Unit as the target address. The timing of Read and Write Special Registers operations are shown in Table A-4-3 and Table A-4-4 respectively. They are similar to Register–Register operation except special registers are involved instead of general purpose registers.

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;
Exec Stage: Phase 1	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ; BUSBUFA <– busA, if (busI<14>=0) BUSBUFB <– (not busB) else BUSBUFB <– Sign Extend (busI<13:0>) ;
Phase 2	busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of ALU <– busA2, Port B of ALU <– busB2 ;
Phase 4	busS <– ALU ; Address Pads <– busS ;
Mem Stage: Phase 1	busPC <– INC ; IfetPC <– busPC, I-Unit <– busPC ;
Phase 3	busL <– Data Pads ; Dst2 <– busL ;
Wr Stage: Phase 3	busA <– Dst1, busB <– (not Dst2) ; REG_FILE[rd] <– (busA & (not busB)) ;

Table A-4-1 Load Operation

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;
Exec Stage: Phase 1	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ; BUSBUFA <– busA, if (busI<14>=0) BUSBUFB <– (not busB) else BUSBUFB <– Sign Extend (busI<13:0>) ;
Phase 2	if (opcode = RETURN) Cwp <– Cwp - 1, busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of ALU <– busA2, Port B of ALU <– busB2 ;
Phase 4	busS <– ALU ; BUSS2PC <– busS ;
Mem Stage: Phase 1	busPC <– BUSS2PC ; IfetPC <– busPC, I-Unit <– busPC ;
Wr Stage: Phase 3	if (opcode = RETURN) Update Backup Copy of Cwp ;

Table A-4-2 Jump–Register and Return Operations

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;
Exec Stage: Phase 2	busS <– (Cwp, Swp, Upsw, or Kpsw) ; BUSSTOD <– busS ;
Phase 4	busD <– BUSSTOD ; Dst1 <– busD ;
Mem Stage: Phase 1	busPC <– INC ; IfetPC <– busPC, I-Unit <– busPC ;
Phase 3	Dst2 <– Dst1 ;
Wr Stage: Phase 3	busA <– Dst1, busB <– (not Dst2) ; REG_FILE[rd] <– (busA & (not busB)) ;

Table A-4-3 Read Special Registers Operation

Stage/Phase	Actions
Ifet Stage: Phase 3	busI <– I-Unit[busPC] ;

Exec Stage:
Phase 1
	busA <– REG_FILE[Rs1], busB <– (not REG_FILE[Rs2]) ;
	BUSBUFA <– busA,
	if (busI<14>=0) BUSBUFB <– (not busB)
	else BUSBUFB <– Sign Extend (busI<13:0>) ;

Phase 2	busA2 <– BUSBUFA, busB2 <– BUSBUFB ; Port A of ALU <– busA2, Port B of ALU <– busB2 ;
Phase 4	busS <– ALU ; (Cwp, Swp, Upsw, or Kpsw) <– busS ;
Mem Stage: Phase 1	busPC <– INC ; IfetPC <– busPC, I-Unit <– busPC
Wr Stage: Phase 3	Update Backup Copy of (Cwp, Swp, Upsw, or Kpsw) ;

Table A-4-4 Write Special Registers Operation

A.5. REFERENCES

[Bos88] B. K. Bose, *VLSI Design Techniques for Floating-Point Computation*, Doctoral Dissertation, Computer Science Division, EECS Department, University of California, Berkeley, November 1988.

[KEW85] R. Katz, S. Eggers, D. Wood, C. Perkins and R. Sheldon, ''Implementing A Cache Consistency Protocol'', *The 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, June 17-19, 1985.

[Tay85] G. S. Taylor, ''SPUR Instruction Set Architecture'', in *Proceedings of CS292i: Implementation of VLSI Systems*, R. Katz (editor), Computer Science Division, EECS Department, University of California, Berkeley, September 1985.

[WEG87] D. Wood, S. Eggers and G. Gibson, ''SPUR Memory System Architecture'', Report No. UCB/Computer Science Dpt. 87/394, Computer Science Division, EECS Department, University of California, Berkeley, December 1987.

Appendix B

THE SPUR CPU PROBLEMS REPORT

The great liability of the engineer compared to men of other professions is that his works are out in the open where all can see them. ... If his works do not work, he is damned.

Hebert Hoover, 1916

This appendix lists all the known SPUR CPU problems. Unless otherwise specified, all these problems can still be found in the second version of the SPUR CPU. The solutions to these problems are also listed. The SPUR CPU problems can be classified into three groups: (1) microarchitectural problems, (2) electrical problems, and (3) implementation problems.

B.1. Microarchitectural Problems

The CPU chip is doing exactly what the microarchitect designed it to do although it is not doing what the microarchitect wanted it to do. The microarchitect has designed it wrong! These problems can be simulated in behavioral and switch level simulation. They were not detected during simulation because we did not cover all possible cases or we did not realize they were problems. The SPUR CPU microarchitectural problems are:

(1) The SPUR CPU does not allow two consecutive instructions to modify the the same special register (special registers are listed in Appendix A.2). The problem is that the SPUR CPU cannot recover the special register if the second instruction is trapped. The software solution is to avoid writing code that will modify the same special register in two consecutive instructions. The hardware solution is to add one more temporary latch between the special register and its backup copy (see Figure 3-3-1).

(2) The SPUR CPU cannot recover from an interrupt if the second instruction that is being killed is a CALL or RETURN instruction. This problem and its software solution are discussed in Section 3.3.1. The same hardware solution that can solve Microarchitectural Problem 1 above can also solve this problem.

(3) The SPUR CPU treats the internal instructions just like any other normal instructions. The user can, therefore, use the internal instructions in his program to crash the system. This is a security hole and we did not have any software solution for it. The hardware solution is to change the SPUR CPU instruction decoder such that it treats all internal instructions as privilege instructions. Whenever a user program attempts to execute an internal instruction, the SPUR CPU should take a mode violation trap.

(4) The CPU and the Cache Controller assume different meanings for the *cacheBusy* signal. The CPU assumes it will stay asserted during the entire cache operation. The Cache Controller, on the other hand, assumes it can disassert it in the middle of a cache operation as long as the *dataValid* signal remains disasserted. Consequently, when the Cache Controller disasserts *cacheBusy* momentarily in the middle the TEST_AND_SET operation, the SPUR CPU is confused and starts prefetching instruction erroneously if the I-Unit is enabled. We did not have any software solution for this problem. The easiest hardware solution is to put some "glue" logic on the processor board.

B.2. Electrical Problems

The CPU chip is not doing what the microarchitect nor the logic designer designed it to do due to unexpected electrical problems. These problems cannot be simulated in behavioral nor switch level simulation. Careful and in-depth circuit simulation is the only way to detect these problems. These problems exist because the switch level simulation is not low level enough and it is not practical to run circuit simulation for the entire chip. The SPUR CPU electrical problems are:

(1) A hazardous circuit caused the CALL instruction not able to save the return address properly in the first version of the SPUR CPU. This problem and its solutions are discussed in Section 3.3.2.1.

(2) Misplaced well and substrate contacts resulted in stuck at "0" and stuck at "1" problems in the first version of the SPUR CPU. This problem and its solutions are discussed in Section 3.3.2.2.

(3) There is small gap in one of the wires in instruction unit. We solved this problem by performing micro-surgery on the chip to connect the broken wire using the "Focused Ion Beam IC Development System" available from Seiko Instrument Inc. This problem was introduced when we were fixing problems from the first version.

(4) The SPUR CPU is not ignoring interrupts during global pipeline suspension causes by external cache miss. This problem is caused by a race condition in the trap logic. The easiest hardware solution is to put some "glue" logic on the processor board.

B.3. Implementation Problems

Implementation problems occur when the CPU chip is doing exactly what the logic or circuit designer designed it to do although it is not doing what the microarchitect want it to do. The logic or circuit designer implement something differently than what the microarchitect has in mind! These problems may be detected by comparing the switch level simulation results against behavioral level simulation results if both the switch level and behavioral level descriptions have the proper level of details. These problems exist because of miscommunication between the microarchitect and the logic or circuit designer. The SPUR CPU only has one implementation problem:

(1) The backup copy of all special registers were implemented as dynamic registers. They should be static or pseudo static registers. This problem and its solution is discussed in Section 3.3.3.