AFRL-IF-WP-TR-2002-1566

MorphoSys PROJECT

Dr. Nader Bagherzadeh

University of California, Irvine
Department of Electrical and Computer Engineering
ET 305
Irvine, CA 92697-1875

DECEMBER 2002

Final Report for 07 August 1997 – 31 August 2002

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

[illegible]

KERRY L. HILL
Project Engineer
Embedded Info Sys Engineering Branch
Information Systems Technology Division

Alfred J. Scarpelli

ALFRED J. SCARPELLI
Team Leader
Embedded Info Systems Engineering Branch
Information Systems Technology Division

[illegible]

JAMES S. WILLIAMSON, Chief
Embedded Info Systems Engineering Branch
Information Systems Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE ***(DD-MM-YY)*** December 2002	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 08/07/1997 – 08/31/2002

4. TITLE AND SUBTITLE MorphoSys PROJECT	**5a. CONTRACT NUMBER** F33615-97-C-1126
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62301E
6. AUTHOR(S) Dr. Nader Bagherzadeh	**5d. PROJECT NUMBER** ARPA
	5e. TASK NUMBER AS
	5f. WORK UNIT NUMBER 09
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California, Irvine; Department of Electrical and Computer Engineering; ET 305; Irvine, CA 92697-1875	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate; Air Force Research Laboratory; Air Force Materiel Command; Wright-Patterson Air Force Base, OH 45433-7334	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/IFTA
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2002-1566

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

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT ***(Maximum 200 Words)***
The MorphoSys architecture is a coarse-grain, reconfigurable computing architecture. This program was funded by DARPA in support of the Adaptive Computing Systems (ACS) Technology Initiative. The architecture was successfully functionally prototyped with the fabrication of the M1 chip. This report will review the MorphoSys architecture and the M1 chip. The tools and application mappings are described. The design improvements to the M1 is detailed as the M2 chip design. Additionally, MorphoSys application domains are detailed. Technology transition success through the DARPA Mission Specific Processing Technology Initiative, the MorphoSys related Morpho Technologies venture capital startup, and Motorola licensing agreements are also touched upon. Finally, a program summary is provided.

15. SUBJECT TERMS
adaptive computing systems (ACS), coarse-grained reconfigurable computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	SAR	226	Kerry L. Hill **19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-7698 x3604

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am truly indebted to my colleague and Co-PI, Fadi Kurdahi, for his suggestions and key contributions throughout the first half of this project. The comments, suggestions, and guidance of Kerry Hill were essential for the success of our project. I am also grateful to other collaborators at WPAFB, in particular Al Scarpelli. I appreciate the support of the DARPA management team, Jose Munoz, Bill Phillips, and Bob Reuss that were involved with this project during different phases of the development.

This project would not have been possible without the dedication and contributions of former graduate students and visiting researchers. In particular, I appreciate the work of Eliseu Chaves, Vladimir Castro, Yan Wang, Walid Najjar, Manuel Anido, Alexander Paar, Qing Zhao, Marcos Sanchez-Elez, Ming-hau Lee, Guangming Lu, Hartej Singh, Rafael Maestre, Chengzhi Pan, Haitao Du, Sadik Can, Afshin Niktash, Hooman Parizi, Amir Kamalizad, Arezou Koohi, Chunhui Zhang, Yun Long, Richard Plettner, and Nozar Tabrizi.

Foreword: Technology Transfer

The integrated nature of the MorphoSys architecture provides the potential to satisfy the increasing demand for high performance and low power communication DSP applications. This architecture targets applications with inherent parallelism, high regularity, computation-intensive nature and word-level granularity. In particular it is well suited for source and channel coding applications, which occur in modern digital communication systems. Some examples of these applications are baseband processing for wireless communication, beam forming, antenna diversity, compression, image processing, and other multimedia applications. One of the objectives of this project was to demonstrate these advantages as well as the flexibility of the MorphoSys model for rapid creation of SOC's with ASIC like performance. Moreover, the reconfigurability of the MorphoSys is better suited for adaptive signal processing applications such as the adaptive beam former of a space based radar system.

As far as GOPS/Watt is concerned, the MorphoSys architecture is in par with some of the current ASIC devices, but what sets it apart is ease of integration and reusability of the design as compared with ASIC solutions. The investment made in developing the software kernels for handling computation intensives DSP tasks will be used throughout the future development of the architecture using more aggressive memory and chip design techniques. Also, because of its synthesizable design approach, it is foreseeable to utilize lower power and faster standard cell libraries available from the MSP community. The compact and flexible nature of this architecture is another reason for using the proposed architecture as one of the supercells available from the MSP project for use in future military applications that will benefit from the programmability, performance, size and low power advantages of the MorphoSys processing architecture.

The reconfigurable digital signal processing (DSP) design based on the MorphoSys architecture for stream-based signal processing was the key enabling technology for a startup company called Morpho Technologies that was established in October 2000 in Irvine, California. This company has expanded to more than 30 employees dedicated to commercializing the design concepts that were initiated in this DARPA project for military and civilian applications. In particular, the focus is to reconfigurable DSP to support wireless applications, such as Wideband Code Division Multiple Access (WCDMA) cellular systems, that require high performance and low power DSP capability.

Because of the solid technological achievements in the architecture and tools development for programming, Morpho Technologies was able to secure customers in the competitive reconfigurable DSP design market. In the domain of wireless infrastructure (base station) for 3G cellular phone systems, after months of competitive design evaluations, Motorola selected the Morpho technologies solution for their design. Given the existence of half a dozen companies in this arena, the selection of Morpho's design is a proof of the technology and its impact.

The DARPA community also regarded the design of the MorphoSys appropriate for their applications. The Northrop Grumman as the lead PI for one of the MSP program funded projects has selected Morpho's architecture for their design. This effort has resulted in a flexible and programmable solution for space radar and other applications that require strict specification for power, performance, weight, and area.

1. Background and Introduction

Traditionally, there have been two classes of computing: general-purpose systems and application-specific embedded systems. The former is based on the usage of general-purpose processors wherein a single processor is used to execute a variety of applications. The latter category is based on the use of application specific integrated circuits (ICs) or ASICs, each of which is custom designed for a specific application. However, there is also a third class of computing, reconfigurable computing, which has recently come forth into prominence. Reconfigurable computing refers to computing systems that have reconfigurable architectures or are based on reconfigurable hardware components. The underlying principle of these systems is that the hardware organization, functionality or interconnections may be customized after fabrication of the system on silicon. This customization helps satisfy specific computational requirements of different applications. This feature of postfabrication customization is in contrast to application-specific ICs, which are permanently configured at fabrication time and cannot be reconfigured afterward. The configuration of each reconfigurable system is specified through an additional software component. Reconfigurable systems may be configured repeatedly, in times ranging from milliseconds to a nanosecond or even less. A common example of a reconfigurable device is a field programmable gate array (FPGA). Many reconfigurable systems are based on the use of one or more FPGAs.

A reconfigurable architecture or a system consisting of reconfigurable devices exhibits the following features:

- Configurable functionality: This implies that the processing unit can execute different functions at different times based on different configuration information.
- Configurable interconnect: This refers to the ability of the interconnections to be changed to meet different communication patterns or other application constraints.

The key point behind reconfigurable architectures is that in some special applications, one such system may replace several ASICs corresponding to different configurations of the former. This replacement clearly results in considerable silicon, power, and weight savings, when ASICs and reconfigurable architectures are contrasted with each other. Reconfigurable architectures, in return, suffer from some performance degradation. This, however, is easily tolerated in many cases. The so-called expensive (i.e. complicated) features of general purpose processors, on the other hand, are not usually exploited thoroughly in our intended applications. Therefore, the major asset of these processors is not utilized efficiently, and hence their application is nearly ruled out, as well.

A reconfigurable system may be optimized for multiple applications. Different functions and variable interconnect patterns enable the execution of a range of applications. The combined effect is that applications execute faster (with performance levels close to that of ASICs) and more applications can be executed on the same system.

Reconfigurable systems have gained significance as they represent an intermediate approach between general-purpose processors and ASICs. The major motivation for their usage is that they are free from some of the limitations of both the above categories, while exhibiting beneficial characteristics of each. ASICs are extremely limited in usage since they target only one application. In contrast, general-purpose processors are designed to execute a variety of different applications. However, ASICs can execute applications very fast in order to meet real-time constraints, but general-purpose processors typically have much lower performance levels as compared to ASICs and usually cannot meet real-time performance constraints. Reconfigurable systems exhibit wider applicability than an ASIC and higher performance levels than general-purpose processors and better power consumption characteristics.

In summary, a reconfigurable architecture lends itself to on-board science processing, offering small silicon size and physical weight, low power dissipation, and sufficient performance, for many applications as discussed in Sections 3 and 4 of this report.

These upsides are useful in many real-world scenarios. Consider a system designed for a given class of applications (image processing, data encryption, etc.). Each application may have a complex and heterogeneous nature and comprise several subtasks with varying characteristics. For instance, a multimedia application may include a data-parallel task, a bit-level task, irregular computations, high-precision word operations, and a real-time component. For such complex applications with wide-ranging subtasks, the ASIC approach would lead to an uneconomical die size or a large number of separate chips. Also, most general-purpose processors would very likely not satisfy the performance or power constraints for the entire application. However, a reconfigurable system may be optimally reconfigured for each subtask, thus meeting the application constraints within the same chip. The same reconfigurable system can be reused to perform the other tasks in this application class while still satisfying performance and power constraints.

In this document, first we will discuss the MorphoSys architecture, followed by the first implementation of this architecture, called the M1 chip. The tools and application mappings are discussed after the architectural discussion. The design and improvements to M1 are discussed as the M2 chip design, followed by a wrapup discussion on more recent application domains. This document concludes with a research summary and related future research.

1.1. MorphoSys Architecture

The MorphoSys design model incorporates a reconfigurable component (to handle high-volume data-parallel operations) on the same die with a general-purpose reduced instruction set computer (RISC) processor (to perform sequential processing and control functions), and a high bandwidth memory interface.

The MorphoSys architecture comprises five major components: the reconfigurable cell array (RC Array), control processor (TinyRISC), Context Memory, frame buffer, and a direct memory access (DMA) controller. Figure 1 shows the organization of the integrated MorphoSys reconfigurable computing system.

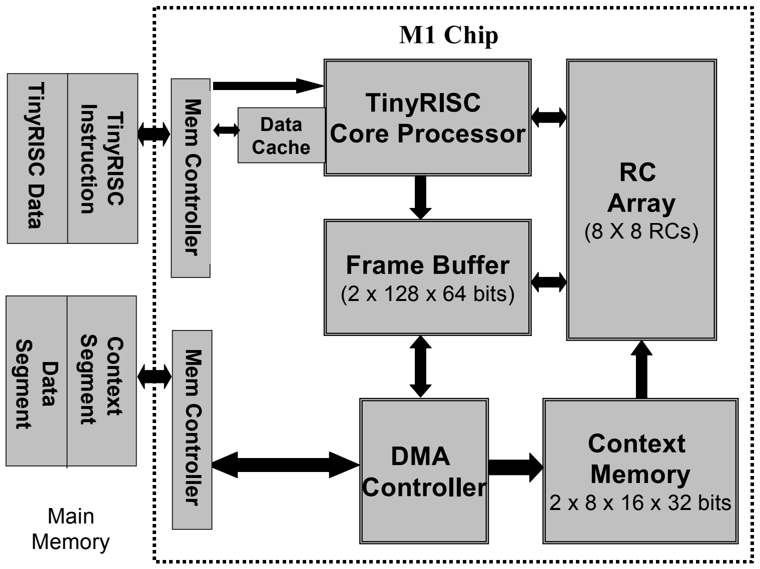


Figure 1. MorphoSys Integrated Architectural Model

1.1.1. RC Array

In the current implementation, the reconfigurable component is an array of RCs or processing elements. Considering that target applications (video compression, etc.) tend to be processed in clusters of 8- by 8 data elements, the RC Array has 64 cells in a two-dimensional matrix, as illustrated in Figure 2. This configuration is chosen to maximally utilize the parallelism inherent in an application, which in turn enhances throughput.

The RC Array follows the single instruction multiple data (SIMD) model of computation. All RCs in the same row/column share the same configuration data (context). However, each RC operates on different data. Sharing the context across a row/column is useful for data-parallel applications. The RC Array has an extensive three-layer interconnection network, designed to enable fast data exchange between the RCs. This results in enhanced performance for application kernels that involve high data movement, for example, the discrete cosine transform (used in video compression).

Each RC incorporates an arithmetic logic unit (ALU)-multiplier, a shift unit, input muxes, and a register file. The multiplier is included since many target applications require integer multiplication. In addition, there is a context register that is used to store the current context and provide control/configuration signals to the RC components (namely, the ALU-multiplier, shift unit, and the input muxes).

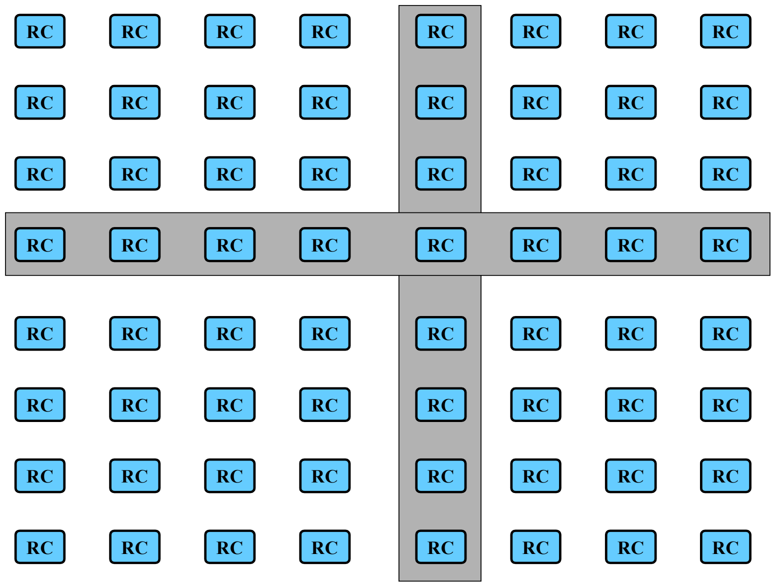


Figure 2. An 8 by 8 RC Array

1.1.2. Frame Buffer and DMA Controller

The potential parallelism of the RC Array would be ineffective if the memory interface is unable to transfer data at an adequate rate. Therefore, a high-speed memory interface consisting of a streaming buffer (frame buffer or FB for short) and a DMA controller is incorporated in the system. The FB has two sets as illustrated in Figure 3. The DMA controller controls the communication between the FB and main memory. Using the two sets of FB alternatively, the computation of the RC Array and the data overlap the load and store of the FB and, therefore, the memory accesses are virtually transparent to the RC Array.

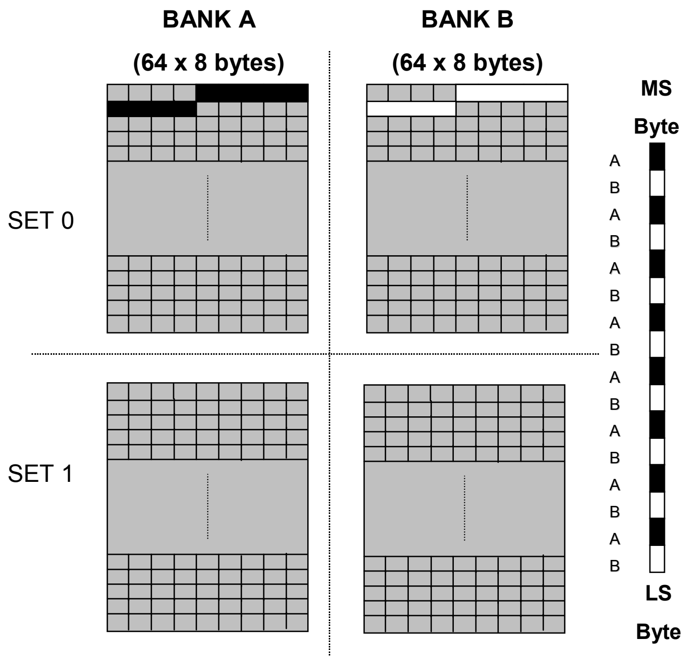


Figure 3. Frame Buffer Block Diagram

1.1.3. Context Memory

The Context Memory stores multiple planes of configuration data (context) for the RC Array, thus providing depth of programmability. As shown in Figure 4, the Context Memory is logically organized into two blocks, column context block and row context block. Each block consists of 8 context sets where each context set has 16 context words. Each row/column context block controls one column/row of the RC Array. This means that Context Memory provides a depth of 32, which has been proven to be adeqauate for most of the DSP and image processing applications we have investigated. This implies that the system spends less time loading fresh configuration data. Fast dynamic reconfiguration is essential for achieving high performance with a reconfigurable system. MorphoSys supports single-cycle dynamic reconfiguration (without interruption of RC Array execution).

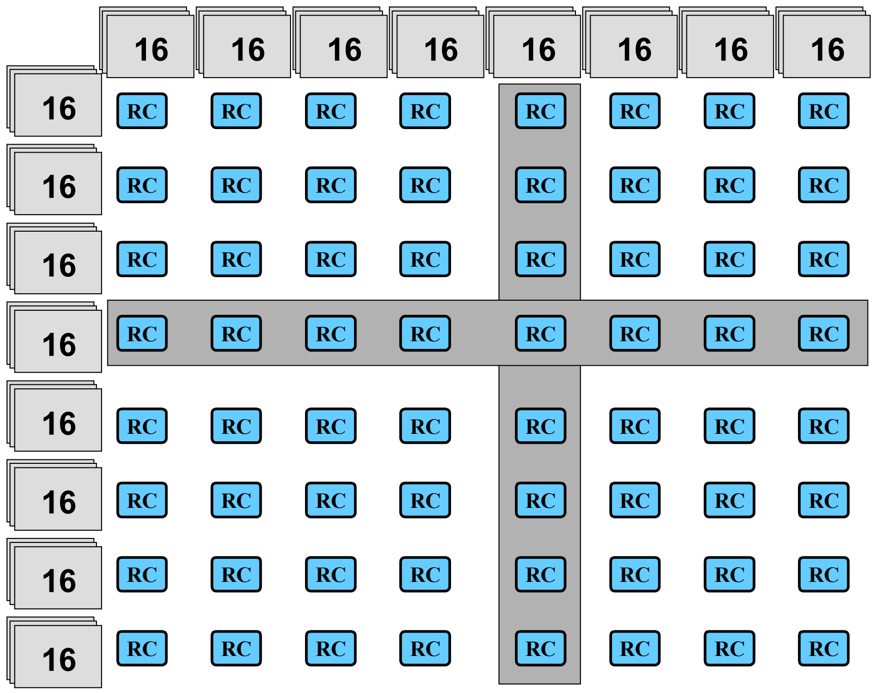


Figure 4. Structure of Context Memory

1.1.4. TinyRISC

Figure 5 shows the block diagram of TinyRISC. Since most target applications involve some sequential processing, a RISC processor, TinyRISC [1], is included in the system. This is a MIPS-like processor with a four-stage scalar pipeline. It has a 32-bit ALU, register file and an on-chip data cache memory. This processor also coordinates system operation and controls its interface with the external world. This is made possible by the addition of specific instructions (besides the standard RISC instructions) to the TinyRISC instruction set architecture (ISA). These instructions initiate data transfers between main memory and MorphoSys components, and control execution of the RC Array.

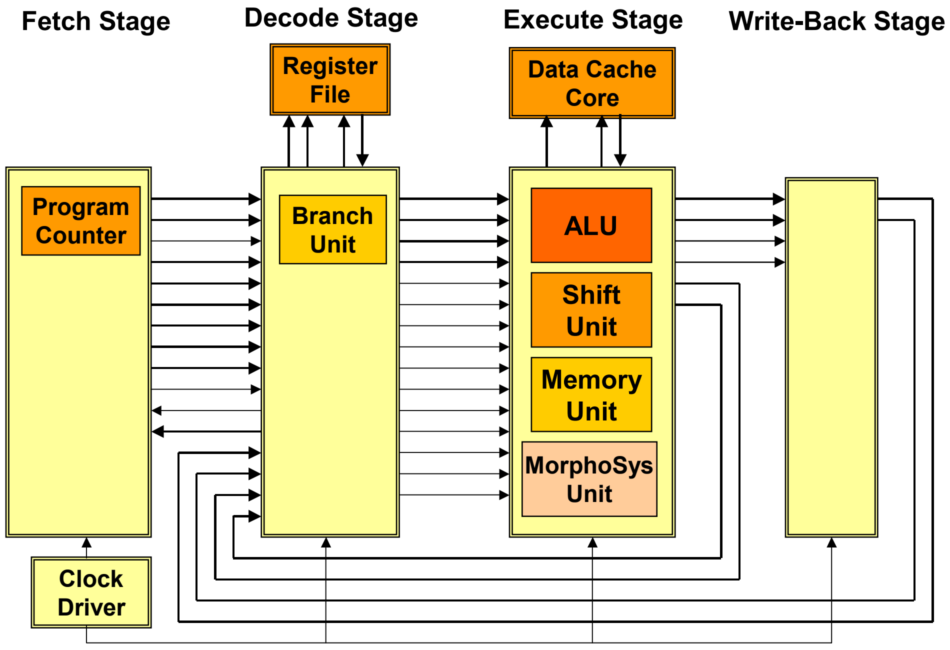


Figure 5. TinyRISC Block Diagram

MorphoSys implements a novel control mechanism for the reconfigurable component through the TinyRISC instructions. The TinyRISC ISA has been modified to include several new instructions (Table 1) that enable control of different components in the system. These instructions contain fields that directly provide different control signals to the RC Array, DMA controller, FB, and Context Memory. There are two major categories of these new instructions: DMA instructions and RC Array instructions.

The DMA instructions contain fields that provide the DMA controller with adequate information regarding the direction, destination and source of every data transfer: starting address in main memory, starting address in the FB and Context Memory, and number of bytes to be transferred.

The RC Array instructions have fields that provide the control signals to the RC Array and the Context Memory. This is essential to enable the execution of computations in the RC Array. This information includes the contexts to be executed, the mode of context broadcast (row or column), location of data to be loaded in from the FB, etc.

1.2. MorphoSys Execution Model

The execution model for MorphoSys is based on partitioning applications into sequential and data-parallel tasks. TinyRISC handles the sequential portion of the application, whereas the data-parallel portions are mapped to the RC Array. TinyRISC initiates all data transfers involving application and configuration data (context). TinyRISC provides various control/address signals for Context Memory, Frame Buffer, and the DMA controller. RC Array execution is enabled through special TinyRISC instructions for context broadcast.

Table 1. New TinyRISC Instructions

Mnemonic	Description of Operation
LDCTXT	Load Context from Main Memory to Context Memory.
LDFB STFB	Load (store) data from (into) Main Memory to (from) Frame Buffer.
CBCAST	Context broadcast, no data from Frame Buffer.
DBCBC DBCBR	Column (or row) context broadcast, get data from both banks of Frame Buffer.
DBCB	Context broadcast, get data from both banks of Frame Buffer.
SBCB	Context broadcast, transfer 128 bit data from Frame Buffer.
WFB WFBI	Write the processed data back to Frame Buffer (with indirect or immediate address).
RCRISC	Write one 16-bit data from RC Array to TinyRISC.

The MorphoSys program flow may be summarized as follows. First, a special TinyRISC instruction, LDCTXT is issued. This initiates loading of context words (configuration data) into the Context Memory through DMA Controller (Figure 1). Next, the LDFB instruction causes the TinyRISC to signal the DMA Controller to load application data, such as image frames, from main memory to the Frame Buffer. When both configuration and application data are ready, a TinyRISC instruction for context broadcast, such as CBCAST, SBCB, etc., is issued. This starts execution of the RC Array.

The context broadcast instructions specify the particular context (from among the multiple contexts in Context Memory) to be executed by the RCs. There are two modes of specifying the context: column broadcast and row broadcast. For column (row) broadcast, all RCs in the same column (row) are configured by the same context word. TinyRISC can also selectively enable a row/column, and can access data from selected RC outputs.

During the execution of a context/data broadcast instruction, the intended context and data are taken from the context memory and the FB, respectively, and loaded into the corresponding registers inside the corresponding RCs, so that in the following clock cycle, the data can be processed as instructed by the context. All different context types are executed in one clock cycle.

MorphoSys supports dynamic reconfiguration. Context data may be loaded into a non-active part of the Context Memory without interrupting RC Array operation. Since the Frame Buffer has two sets, it is possible to overlap computation in the RC Array with data transfers between external memory and the Frame Buffer. While the RC Array performs computations on data in one Frame Buffer set, fresh data may be loaded in the other set or the Context Memory may receive new contexts.

1.3. Extensions to the MorphoSys Model

There are two issues that need to be clarified with reference to the current implementation for MorphoSys. First, the MorphoSys architecture is not limited to using TinyRISC as the main control processor. TinyRISC is used only to validate the design model. Several possible extensions to this model can be envisioned. One would be to use an advanced general-purpose processor in conjunction with TinyRISC (which would then function as an input/output (I/O) processor for the RC Array). Also, an advanced processor with multiple datapath pipeline, such as superscalar or very long instruction word (VLIW) processors, may be used as the main processor. This would enable concurrent processing of the RC Array and the main processor.

The second issue relates to the RC Array. For the current implementation, the RC Array has been finetuned for data-parallel, computation-intensive tasks. However, the design model allows different versions. For example, a suitably designed RC Array may be used as well for stream processing, high-precision signal processing, bit-level operations, control-intensive applications, etc.

1.4. Related Works

There has been considerable research effort for developing coarse-grain reconfigurable processors, such as MATRIX [3], RaPiD[2], RAW[5], and REMARC [4].

MATRIX is a unique architecture in that it is designed as deployable resources that can serve as data memory, datapath ALU, control logic, or instruction memory. The 8-bit basic units (BFUs) are organized as an array. Each BFU has a 256-word memory, an ALU-multiplier unit, and a reduction control logic unit. MATRIX has a similar interconnection network as MorphoSys. The interconnection network has a hierarchy of three levels, but unlike MorphoSys, the levels of interconnect have variable delay (in terms of pipeline stages). The control and array processors are configured out of the same hardware resources. This makes the dynamic system control quite complicated. Also, unlike MorphoSys, this work does not specify the data interface to the external world. MATRIX can deliver up to 10 giga operations per second (GOPS) with 100 BFUs when operated at 100 MHz. Unlike MorphoSys, MATRIX was not implemented as a chip.

RaPiD is organized as a linear array (8 to 32 cells) of functional units configured to form a linear computation pipeline. Therefore, it performs well for systolic applications, but has limited performance for block-oriented application tasks. Each array cell consists of an integer multiplier, three ALUs, and registers and local memory. Segmented buses are used for efficient utilization of interconnection resources. It achieves the performance up to 1.6 GOPS; however, unlike MorphoSys, there is no unified controller and an integrated memory interface is missing.

RAW features a unique approach to implementing a highly parallel architecture by fully exposing low-level details of the hardware architecture to the complier. RAW is a set of replicated tiles, where each tile contains a simple RISC processor, some bit-level reconfigurable logic, and some memory for instructions and data. Each tile also has an associated programmable switch which connects the tiles in a wide-channel point-to-point interconnect. When tested on benchmarks, such as encryption, sorting, FFT, and matrix operations, it provides performance gains from 1X to 100X compared to the Sun SparcStation 20.

REMARC [4] has 64 nanoprocessors. The nanoprocessor does not have a multiplier (even though it targets multimedia applications), but instead has a 16-entry data RAM. The interconnection network has two levels, and the global control unit has to perform the functions of data transfers to the main processor/memory, whereas in MorphoSys, these transfers are carried out by the DMA controller, and are concurrent with the program execution. REMARC does not allow dynamic reconfiguration.

1.5. Summary of MorphoSys Architecture

MorphoSys is a coarse-grain processor designed for data-parallel computation-intensive applications. This integrated architecture has a novel control mechanism and streaming memory interface for the reconfigurable component, and is a complete system-on-a-chip. In summary, the most prominent features incorporated in the MorphoSys architecture are as follows:
Integrated model: MorphoSys has a novel control mechanism for the reconfigurable component that uses a general processor. Except for main memory, MorphoSys is a complete system-on-a-chip.
Coarse-grain reconfigurable computing system: The internal datapath of the RC Array is 16 bits. However, MorphoSys also supports some bit-level operation (e.g., bit-level template matching).
Multiple contexts on-chip: Context Memory can store up to 32 planes of configurations. The users have the option of broadcasting contexts across rows or columns. This feature enables fast single-cycle reconfiguration.
Dynamic reconfiguration: Context data may be loaded into a nonactive part of Context Memory without interrupting RC Array operation. Context loads and reloads are specified through TinyRISC and are actually done by the DMA controller.
On-chip controller: MorphoSys allows efficient execution of applications that have both serial and parallel tasks.
Innovative memory interface: The reading and writing of the Frame Buffer are controlled by the DMA controller. It supports high data throughput by using a two-set data buffer that allows overlap of computation with data transfer.

2. M1 Chip and VLSI Implementation of MorphoSys

In this section, the implementation and verification of the first generation MorphoSys, the M1 chip, are presented. The steps involved in the design and implementation of the major components of MorphoSys, namely the RC, the TinyRISC, the Context Memory, the Frame Buffer, and the DMA Controller, will be described. The chip is designed using 0.35-µm 3.3-V four-metal-layer CMOS technology.

2.1. Design Methodology

MorphoSys components are implemented using both custom design and standard cell design approaches. The components that constitute the critical path (e.g., RC) or have a regular structure (e.g., Context Memory, and Frame Buffer) are custom designed. This enables extensive optimization of these components for the delay and area. The components that are control intensive, consist of random logic, or are not in the critical path, are designed using standard cells. Four metal layers are available for routing; out of these, two layers (metal 3 and metal 4) are reserved for routing between the component blocks. Only two layers are used for routing within a component (such as the RC). We use both IRSIM (switch-level simulator) and Hspice (transistor-level simulator) to verify the design of custom components. For synthesized components, Mentor Graphics' circuit simulator (Lsim and Mac-TA) is used for functional verification and timing analysis. The final integration is performed by Mentor Graphics' automatic placement and routing tool. The implementation and verification methodology is summarized in Figure 6.

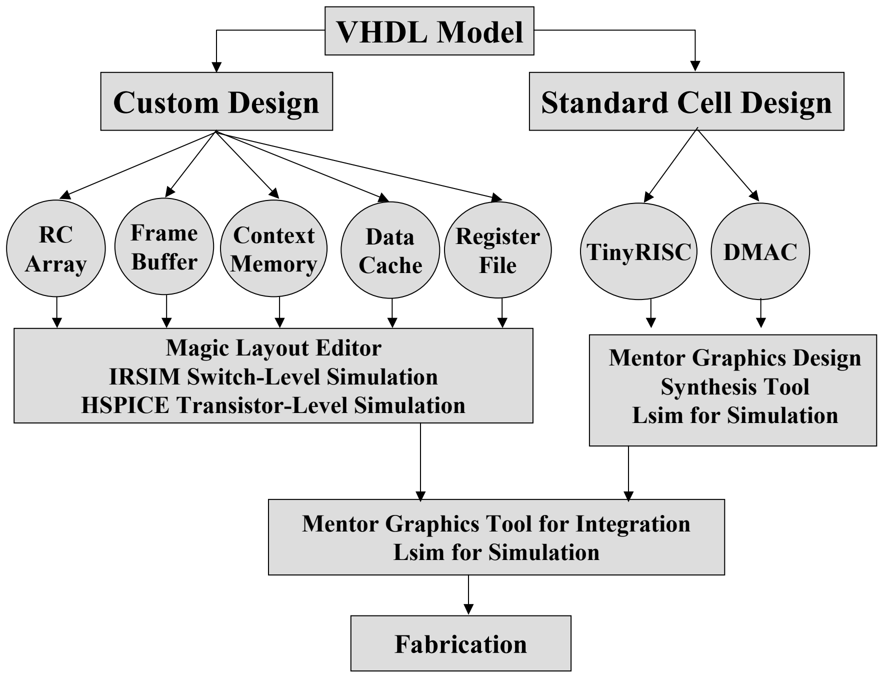


Figure 6. Implementation and Verification Methodology

2.1.1. RC

The RC is the basic element of the RC Array, which is the reconfigurable component of MorphoSys. Each RC (see Figure 7) has a 16 by 12 multiplier. Most multimedia applications require that the second data input to the multiplier be less than or at most equal to 12 bits. Since a 16 x 12 multiplier is significantly smaller, faster, and consumes less power than a 16 by 16 multiplier, and these savings would accrue over 64 RCs, it was decided to use a 16 by 12 multiplier. Corresponding to this input data size, the output of the multiplier cannot be greater than 28 bits. Based on this, we designed a 28-bit ALU for the RC.

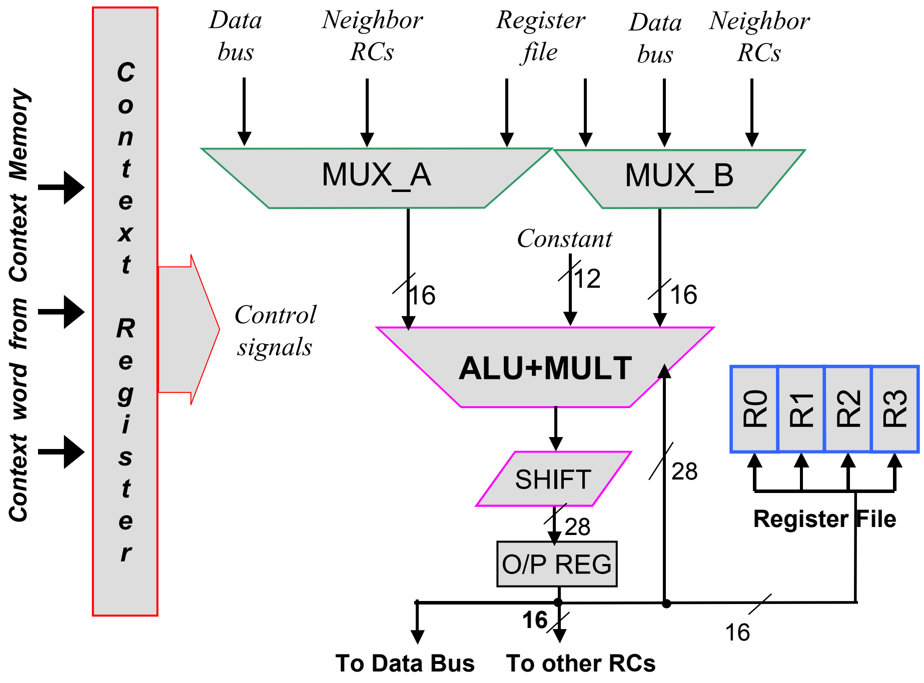


Figure 7. RC Architecture

The data to the ALU-multiplier is provided through two 16-bit input muxes. These muxes allow selection of data operands from different sources. The RC decoder generates control signals for the muxes and the ALU. The critical path of RC consists of the 16-bit input mux, the 16- by 12-bit multiplier, the 28-bit ALU, and a shift unit. Table 2 shows all of the functions implemented in each RC. The special functions, such as absolute value, count ones, and round are implemented as separate units from the ALU to simplify the logic complexity of the ALU and to improve the overall performance. In the following paragraphs, the design and implementation of each component will be presented. The tradeoffs between performance, area, and power consumption will also be discussed.

Table 2. RC Functions

Instruction	Description
A OR B, A AND B, A XOR B, *A OR C, A AND C, A XOR C*	Two-operand logic functions.
A + B, A– B, B – A, A + C, A – C	Two-operand arithmetic functions.
A * C	Multiplication with constant.
A*C + B, A*C + Out(t), A*C – Out(t)	Multiply-accumulate functions.
| A - B | + Out(t)	Absolute difference accumulate.
A *AND* B : Count Ones	AND, then count number of ones in result.
A+B if A>0, A-B if A<0	Conditional add/subtract based on sign bit of A.
Rounding, RESET, BYPASS A, LOAD Constant, No-op	Miscellaneous functions.

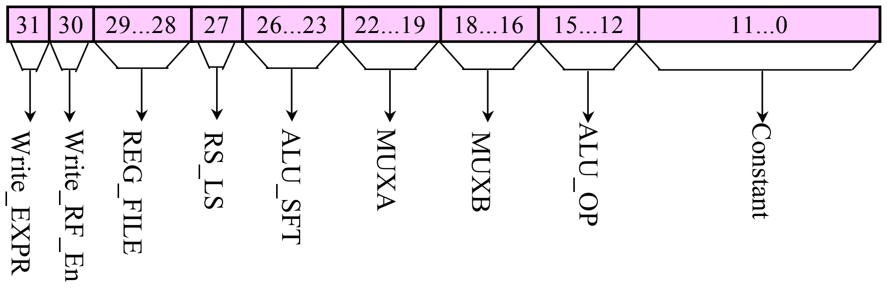
- KEY: A = Mux A operand, B = Mux B operand, C = constant, Out(t) = previous output, Out(t+1) = new output

The constraint of completing the multiply-accumulate (MAC) and shift operations in one cycle (10 ns) is the most challenging part of the design of the RC. The tight delay constraint motivated the use of advanced circuit design techniques. Also, since there are 64 RCs in the RC Array, a small increase in the area or power consumption of an RC would have resulted in a multiplicative effect. Hence, we manually designed the entire RC.

2.1.2. Context Register

There is a 32-bit context register in each RC that receives the configuration data from Context Memory and provides the control signals to all of the components in each RC (e.g., muxes, multiplier, ALU, shifter) through the RC decoder. Different fields of the context word are defined in Figure 8.

Context Field with Constant Operation



Context Field without Constant Operation

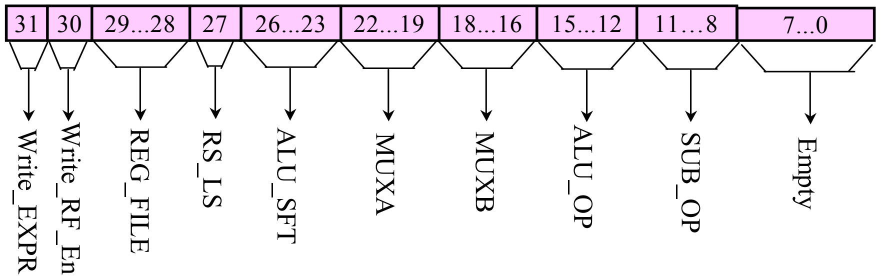


Figure 8. Definition of Context Word

The field ALU_OP specifies the ALU function. The control bits for Mux A and Mux B are specified in the fields MUXA and MUXB. Other fields determine the registers to which the result of an operation is written (REG #), and the direction (RS_LS) and amount of shift (ALU_SFT) applied to output. The 12 least significant bits (LSBs) of the context word represent the constant field. This field is used to provide an operand to a row/column of the RC directly through the context word. It is useful for operations that involve constants, such as multiplication by a constant. However, if such an operation is not needed, some of the extra bits in the constant field may be used to specify an ALU-Multiplier suboperation. These suboperations allow expansion of the functionality of the ALU unit. Table 3 lists all the RC functions and the corresponding control bits in the context word. R_3-R_0 represent the ALU-OP field, and S_3-S_0 represent the most significant four bits of the constant field. As illustrated in Table 3, when $R_3R_2R_1R_0$ = 1111, the constant field is not used, and $S_3S_2S_1S_0$ are used to decode the RC function. When $R_3R_2R_1R_0S_3S_2S_1S_0$ = 11111000, the RC output register keeps the data of the previous clock cycle and the RC is performing a no-operation (NOP) (see the last row of Table 3).

2.1.3. Multiplier

The 16 by 12 multiplier is the component that requires the maximum area and has the longest delay in the RC. Therefore, we use complementary pass-transistor logic (CPL) circuitry for designing the multiplier. CPL allows the realization of complex logic functions with a minimum number of transistors. It also features high-speed operation and low power consumption.

Table 3. RC Functions and Their Opcodes

$R_3R_2R_1R_0S_3$ $S_2S_1S_0$		in	OPA	OPB	**RC Function**
0000			A	C	Bypass C
0001			A	C	A or C
0010			A	C	A and C
0011			A	C	A xor C
0100			A	C	A + C
0110			A	C	A – C
1001					A * C
1100			A*C	Out(t)	A * C + out(t)
1110			A*C	Out(t)	A * C – out(t)
11110000			A	B	Bypass A
11110001			A	B	A or B
11110010			A	B	A and B
11110011			A	B	A xor B
11110100			A	B	A + B
11110110			A	B	A – B
11110111			A	B	B – A
11111010			A	B	BTM (A and B, 1's counter)
11110101 flag=0 flag=1			A	B	A + B A – B
11111110			A	B	| A – B | + out(t)
11111000					Out (t+1) = Out (t) (NOP)

Figure 9(a) (CPL1) shows the basic structure of the CPL circuit. The NMOS pass-transistor network is used to realize the logic function and the two output inverters are used as a level restoration block. This circuit suffers from static power consumption due to the low-swing feature of the NMOS pass-transistor networks. The high level of output inverter inputs are actually lower than the supply voltage, and PMOS transistors are not completely turned off in this situation and, therefore, leakage current will flow in the output inverters. Figure 9(b) (CPL2) shows the modification that solves the static power problem. The two small cross-coupled PMOS transistors are used to restore the outputs of the NMOS pass-transistor network to supply voltage level.

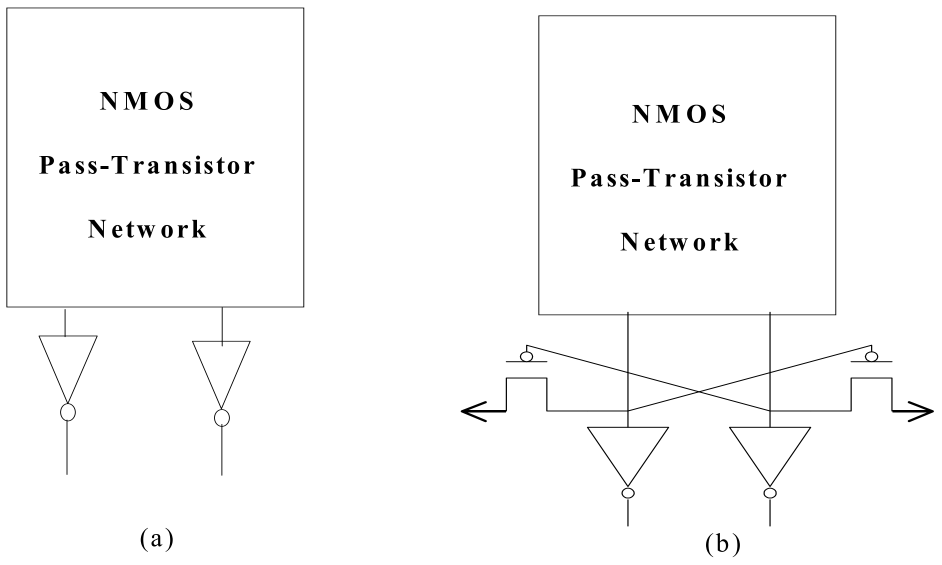


Figure 9. CPL Structure

The multiplier is designed using a carry-save adder (CSA) array structure with a 16-bit carry-skip adder. The CPL implementation of the CSA is shown in Figure 10.

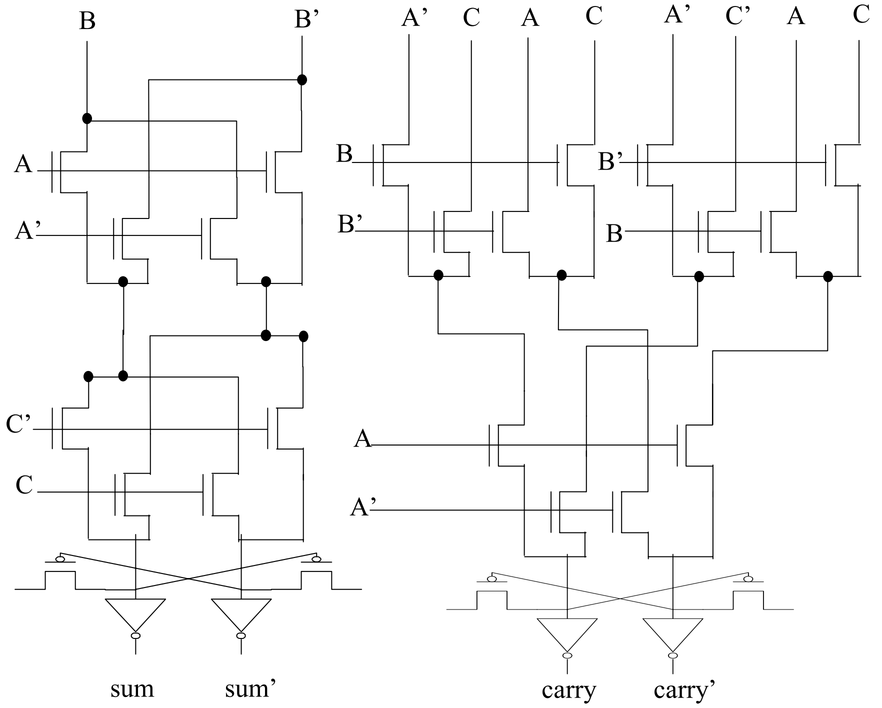


Figure 10. CPL Implementation of CSA

Table 4 shows the comparisons of three CSA designs: standard CMOS, CPL1, and CPL2. SPICE simulation (using HP level 39 0.35-μm device models) was carried out for each of the three CSA designs. All of the circuits were simulated using 3.3-V supply voltage. The clock frequency was set to 100 MHz, and the temperature was set to 25 ^{o}C. From the data in Table 4, the CSA design using CPL2 has the lowest delay-power product; hence, it is used in the current implementation.

Several researchers have shown that both Wallace and Dadda algorithms are efficient for array-type multipliers and can be implemented using the minimum number of CSAs. However, we use a regular array structure instead, which requires more CSAs and has a longer critical path compared to Wallace or Dadda multiplier. The reason behind this decision is that our layout methodology allows only metal 1 and metal 2 for internal routing within components. If we design the multiplier using the Wallace or Dadda design, it would have a much larger area because of irregular structure of these designs. We estimate that both Wallace and Dadda multipliers are about 1.5 times larger than the regular array multiplier when only two layers are used for routing. Thus, for 64 RCs, this increase in area is not tolerable. Hence, we use the regular array structure.

Table 4. Comparison of CSA Designs

	Standard CMOS	**CPL1**	**CPL2**
number of transistors	40	28	30
Delay (ns)	0.54	0.20	0.22
Power (mw)	0.21	0.36	0.25

Another important design consideration is to find an efficient algorithm for 2's complement multiplication. The sign extension in 2's complement multiplication would incur a significant penalty in both area and power consumption for the multiplier. The regular CSA array structure leads us to the decision of sign extension and an array reduction algorithm. Figure 11 shows the steps involved in the array reduction using 5 by 5 bits multiplication as an example.

$$B = -B_4 \cdot 2^4 + \sum_{i=0}^{3} B_i \cdot 2^i$$

$$A \cdot B = A \cdot \sum_{i=0}^{3} B_i \cdot 2^i - A \cdot B_4 \cdot 2^4$$

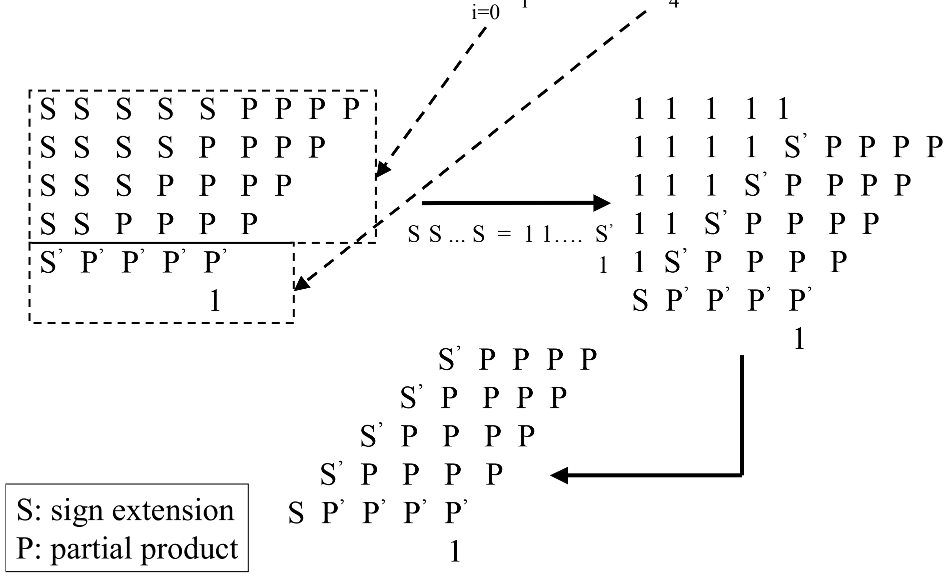


Figure 11. Array Reduction Algorithm for 2's Complement Multiplier

By using this algorithm, the partial product array size of 2's complement multiplication can be reduced to the same as that of integer multiplication, which results in great improvement of area and power consumption without sacrificing the performance. Also, the summation of the partial products can be carried out by a regular structured CSA array without any modification to the CSA, which also greatly simplifies the layout of the multiplier. Figure 12 shows the structure of the 16- by 12-bit multiplier and the result of the partial product array after sign extension and array reduction.

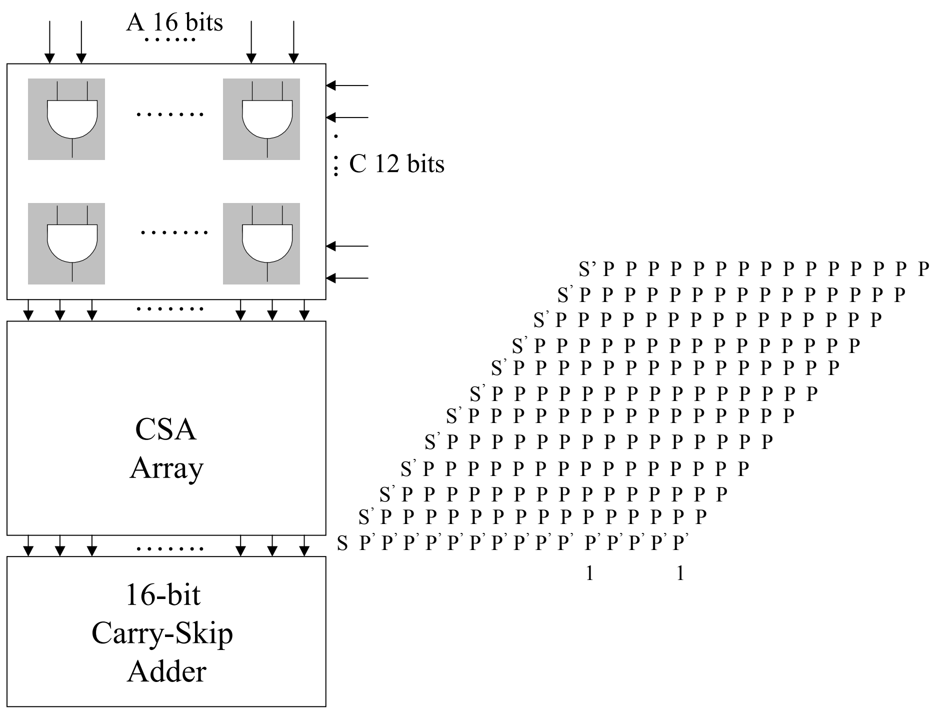

A 16 bits
C 12 bits
CSA Array
16-bit Carry-Skip Adder
S' P P P P P P P P P P P P P P P P
S P' P' P' P' P' P' P' P' P' P' P' P' P' P' P' P'
1
1

Figure 12. Structure and Partial Product Array of 16 by 12 Multiplier

The first part of the multiplier is the partial product generation block, which consists of a 16 by 12 *AND* gates array. It takes 16 bits A input and 12 bits C input to generate a bit array as shown in Figure 12. The CSA array (Figure 13) then performs the partial products summation. The least significant 12 bits of the final results are generated directly by the CSA array, and the 16-bit final adder generates the most significant 16 bits. As we can see from Figure 12, the CSA array has a very regular structure. Each row has 16 full adders, and only connections between two adjacent rows are required.

The critical path delay of the multiplier is analyzed as follows. The partial products generation block has the delay of one *2-input AND* gate. The critical path of the CSA array is 10 full-adders (FAs) since the depth of the array is 10, as shown in Figure 13. The 16-bit final adder is designed using a carry-skip scheme. The detail of the carry-skip adder will be presented in the next section. Figure 14 shows the structure of the 16-bit carry-skip adder. The 16-bit adder is divided into four groups. Within each group, a carry-ripple scheme is used. Its critical path is *4 FAs + 3 4-input AND gates + 3 2-to-1 muxes*.

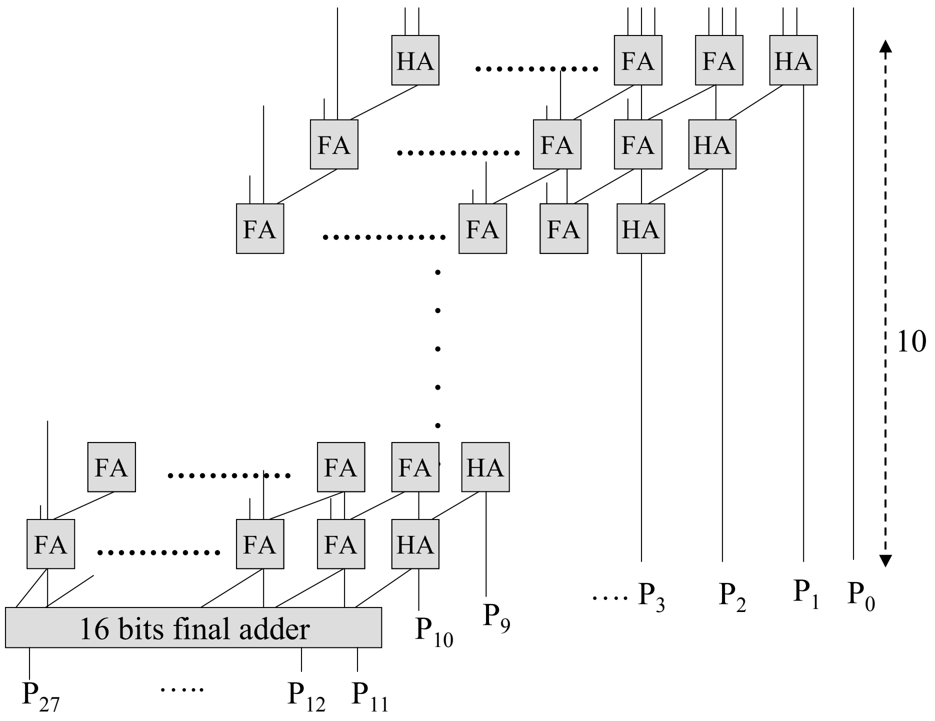


Figure 13. CSA Array Structure

Based on the above analysis, the critical path of the 16- by 12-bit multiplier is *1 2-input AND gate + 14 FAs + 3 4-input AND gates + 3 2-to-1 muxes.* HSPICE was used for delay and power consumption measurement. The 16- by 12-bit multiplier was first extracted using Magic, and a SPICE netlist was generated with all of the parasitic information included. A capacitive load of 100 fF was added to each output in the SPICE netlist. SPICE simulations show that the multiplier delay is 4 ns (0.35-µm, 3.3-V CMOS). The power dissipation is 150 mW at 25 ^{o}C for 100-MHz operation with the input pattern of FFFF*FFF switching to 0000*000. The standby power consumption when inputs to multiplier are kept constant is only 0.6 mW.

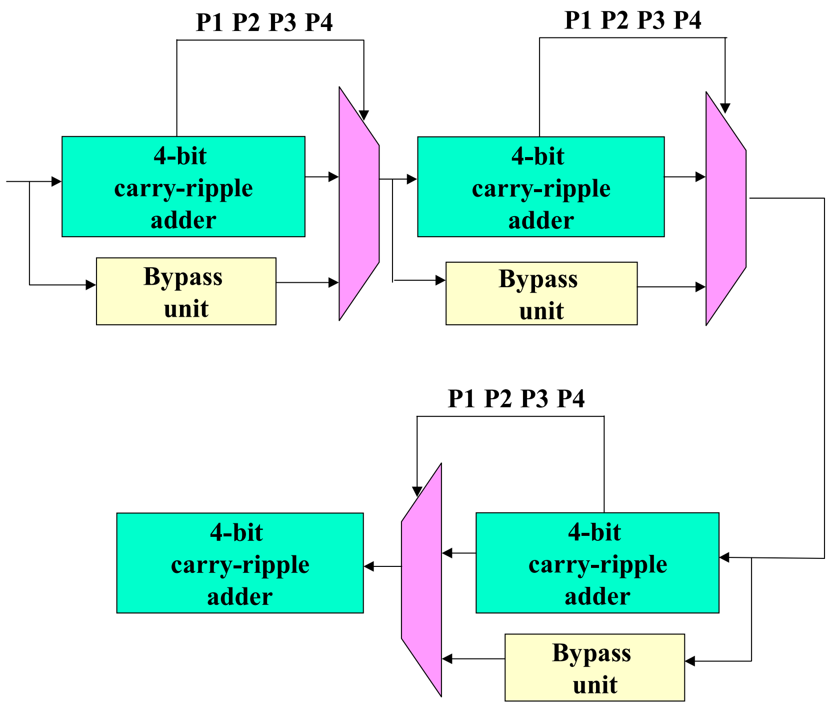


Figure 14. 16-bit Carry-Skip Adder

It is important to note that the multiplier can be disabled when an application does not involve multiplication operations. This feature is realized by bypassing the inputs to the multiplier and having the RC decoder generate control signals to the bypass unit based on the context word. In Moving Picture Experts Group (MPEG), for example, only discrete cosine transform/inverse discrete cosine transform (DCT/IDCT) kernels require multiplication operations, which constitute less than 10 percent of the total operation count. By disabling the multiplier when not in use, a large amount of power can be saved for most of the applications. The bypass unit is implemented using tristate buffers. As we can see in Table 3, for the contexts with constant operation, R_3 is equal only when the multiplier is in use, so R_3 is used as the enable signal to the tristate buffers when the context is in the constant operation mode.

2.1.4. ALU

The ALU of the RC is designed to implement basic logic and arithmetic functions. The logic core of the one bit ALU is shown in Figure 15.

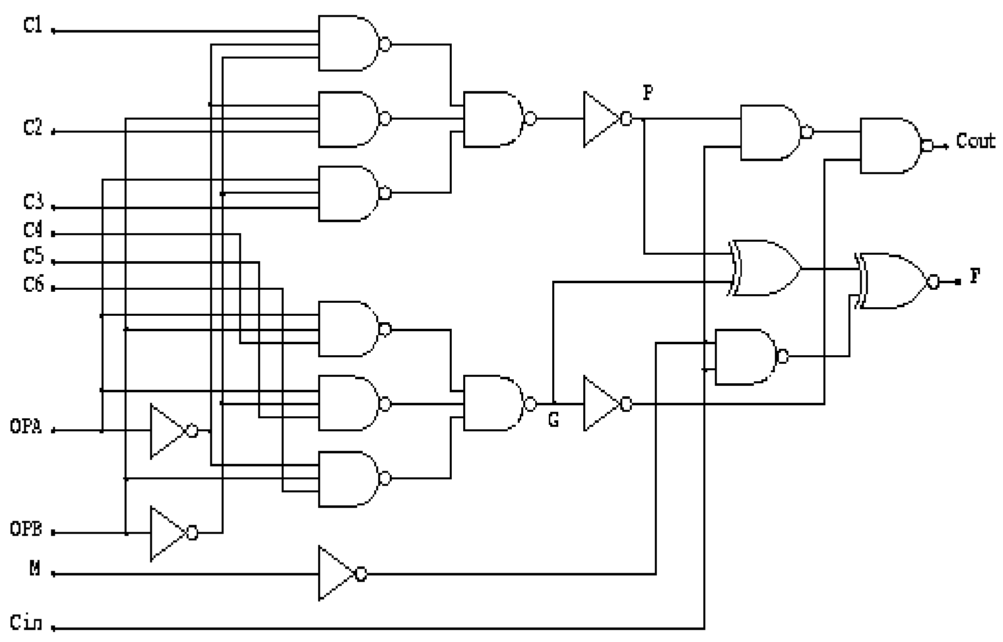


Figure 15. Logic Core of 1-Bit ALU

The functionality of the ALU is configured through the control bits C1 through C6. The RC decoder based on the information in the context word generates these control signals. Table 5 lists the ALU functions and the associated C1 through C6 signals.

Table 5. ALU functions and the associated control signals

C1	C2	C3	C4	C5	C6	M	Cin	ALU Function
0	0	0	0	0	0	0	1	Zero
1	0	0	0	0	0	1	0	A or B
1	1	0	0	1	0	1	0	A and B
1	0	0	1	0	0	1	0	A xor B
1	0	0	1	0	0	0	0	A + B
0	1	0	0	1	0	0	1	A – B
0	0	1	0	0	1	0	1	B – A
1	0	1	0	0	0	1	0	Bypass B
1	1	0	0	0	0	1	0	Bypass A

The important part of ALU implementation is to design the 28-bit addition/subtraction unit for minimum area and delay. The timing budget allows approximately 3 ns for ALU operations. The carry-ripple adder is too slow to accomplish 28-bit addition/subtraction operations in 3 ns. Both carry-lookahead adder and carry-select adder are well-known schemes for high-speed adder designs; however, they require twice as much area as the carry-ripple adder. Consequently, we use a carry-skip scheme (that uses almost the same area as the carry-ripple adder but is much faster) for the ALU design. Figure 16 illustrates the structure of the 28-bit carry-skip adder.

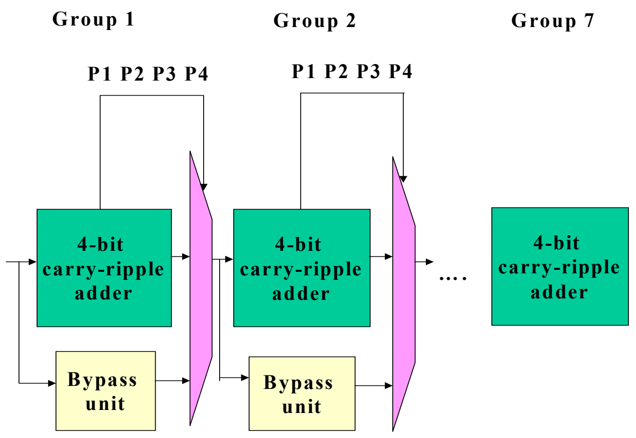


Figure 16. Structure of 28-bit Carry-Skip Adder

The 28-bit adder is divided into seven groups with carry-ripple scheme used in each group. Every group also generates a carry-bypass signal that equals to 1 if all bits internal to the group satisfy $P_i = 1$ ($P_i = x_i \oplus y_i$, where x_i and y_i are the inputs to each bit). This signal can allow the incoming carry to bypass all bits within the group and propagate to the next group. Thus, it reduces the time needed to propagate the carry by skipping over groups of four consecutive adder bits.

The critical path of the 28-bit ALU is equal to *4 1-bit ALU core + 6 4-inputs AND gates + 6 2-to-1 muxes*. The worst-case operation time of the 28-bit ALU from SPICE simulation is within 3 ns. It consumes 15 mw of power at 25 ^{o}C for 100-MHz operation frequency.

2.1.5. Shifter

There is a 28-bit shifter in each RC. The input to this shifter is a 28-bit 2's complement number and the maximum shift width of this shifter is 15 bits. Shifters are usually implemented as an array of pass transistors and can become area intensive when limited layers of metal are allowed. Two different designs were considered for the RC implementation, barrel shifter and logarithmic shifter.

The structure of a barrel shifter is an array of pass transistors where the number of rows equals to the word length of data, and the number of columns equals to the maximum shift width. A major advantage of the barrel shifter is that each signal has to pass through at most one pass transistor, and the propagation delay is theoretically independent of the shift amount or the shifter size. However, this is not true in practice since the capacitance at the input increases linearly with the maximum shift width. Another important property is that the area of a barrel shifter is usually dominated by the number of wires running through the cell and is bounded by the pitch of the metal wires. While the barrel shifter is implemented as an array of pass transistors, the logarithmic shifter uses a staged approach. The total shift amount is decomposed into shifts over power-of-two. A logarithmic shifter with a maximum shift width of N consists of $\log_2 N$ stages, where the ith stage passes its input data to the following stage either unchanged or shifted by 2^i bits. The speed of the logarithmic shifter depends on the shift width in a logarithmic way since a shift of M bits requires $\log_2 M$ stages. In general, it can be concluded that the barrel shifter is appropriate for smaller shifters. For larger shift values, the logarithmic shifter becomes more efficient in terms of both area and speed. Therefore, the logarithm shifter is implemented in RC. Figure 17 shows the structure of the 28-bit logarithm shifter.

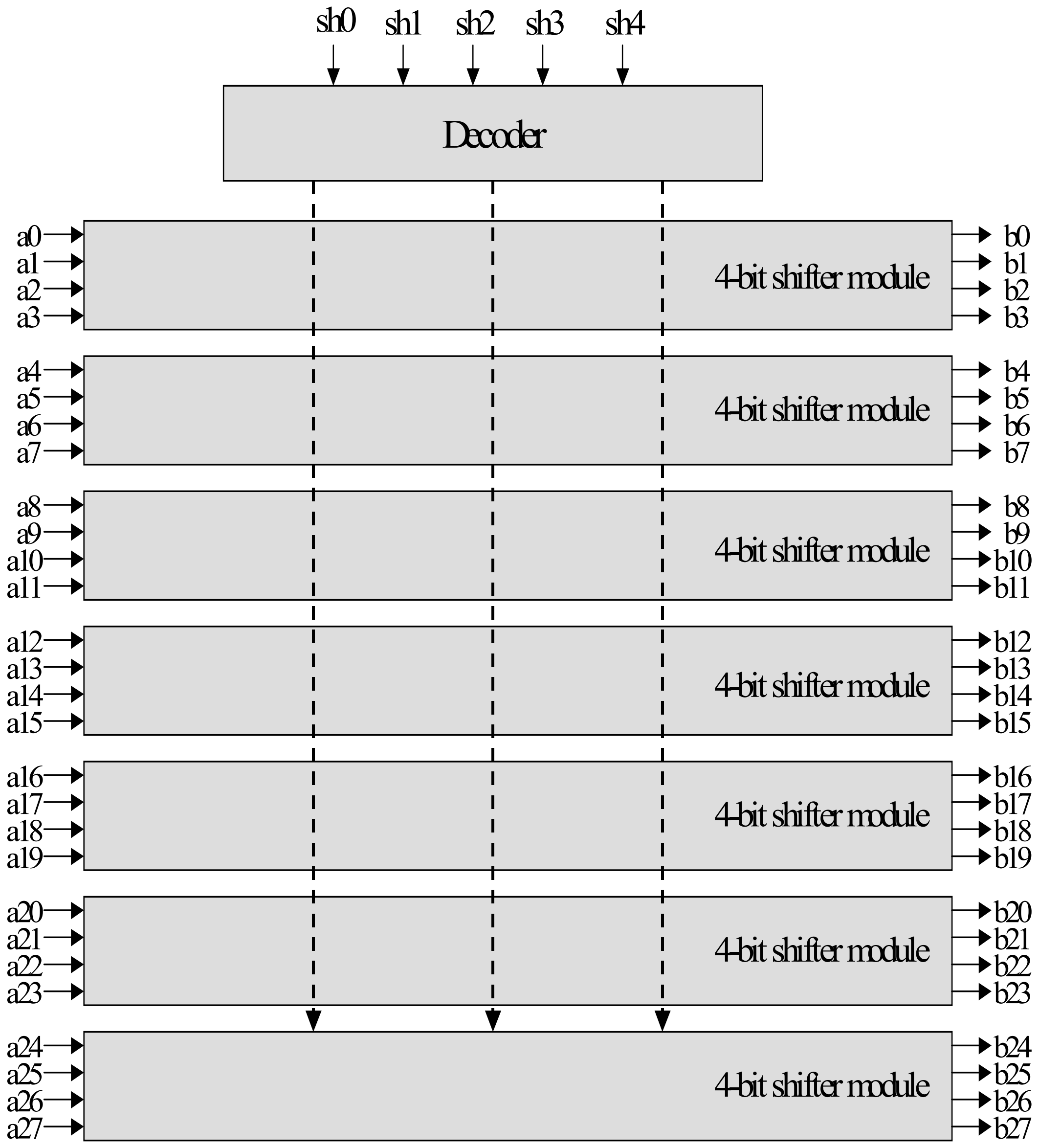


Figure 17. Structure of 28-bit Logarithmic Shifter

The shifter receives a 28-bit input *a27-a0* and generates a 28-bit output *b27-b0* based on the shift value specified by *sh3-sh0* and shift direction (right or left) specified by *sh4*. The shifter is divided into seven cells where each cell receives four bits of the input and generates four bits of output. Figure 18 illustrates the circuit schematic of each cell. Since the maximum shift width is 15 bits, there are four stages is each cell.

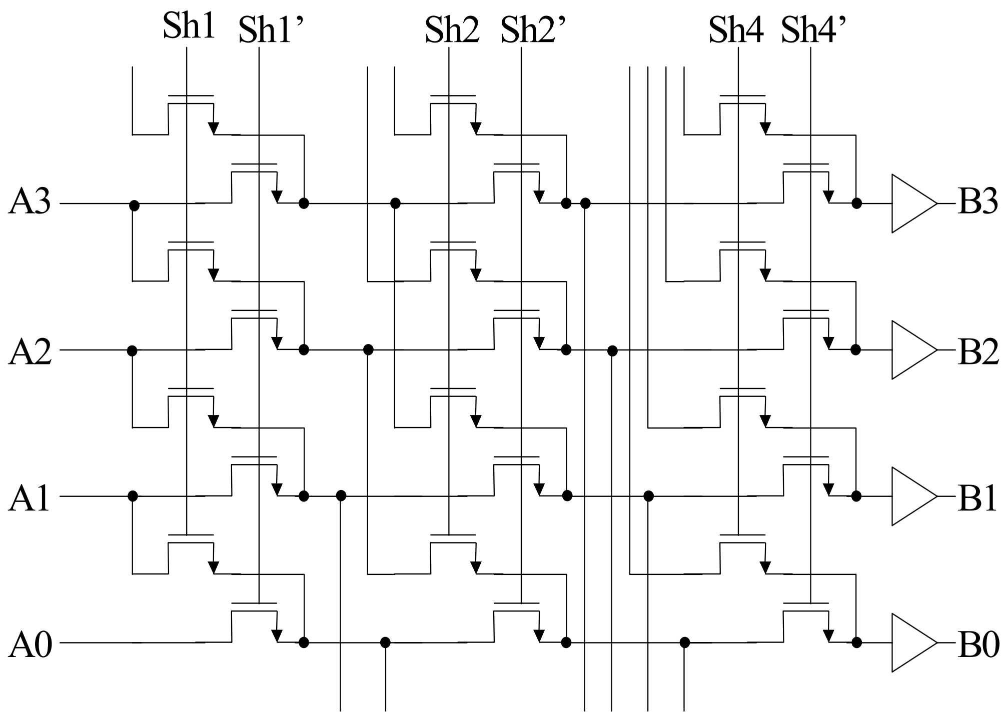


Figure 18. Schematic of 4-bit Cell for Logarithm Shifter

The SPICE simulation shows that the worst-case (when the shift value is 15 bits) delay is 1.2 ns for the 28-bit shifter. The average power consumption is 10 mW at 25 ^{o}C and 100-MHz clock frequency.

2.1.6. One's Counter

The one's counter is custom hardware designed for applications that require processing of binary image data, such as automatic target recognition (ATR). In binary image template matching, the basic operation involved is summing the correlation of the binary template against each plane within the image. To map this operation on MorphoSys, we designed a binary template matching (BTM) function in each RC. It takes two 8-bit inputs, performs bit-wise *AND* on these two numbers, and uses the one's counter to count the number of '1' in the *AND*ed output. The one's counter is designed as a full-adder tree. Figure 19 illustrates its structure.

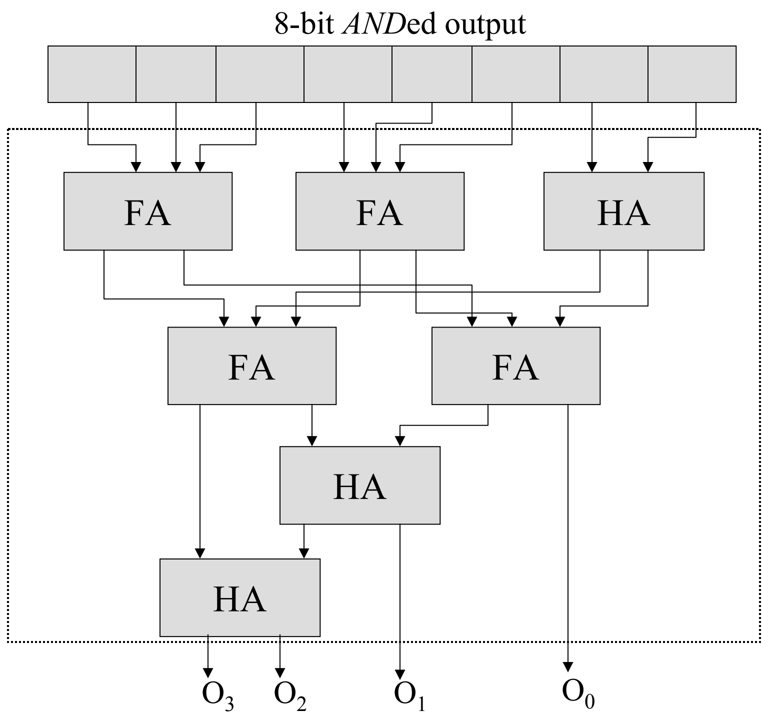


Figure 19. Structure of One's Counter

2.1.7. Muxes

There are five multiplexers in each RC, which are two input multiplexers (inA_mux and inB_mux), two ALU operand selection multiplexers (opA_mux and opB_mux), and one output multiplexer (out_mux). Figure 20 shows the schematic of these multiplexers.

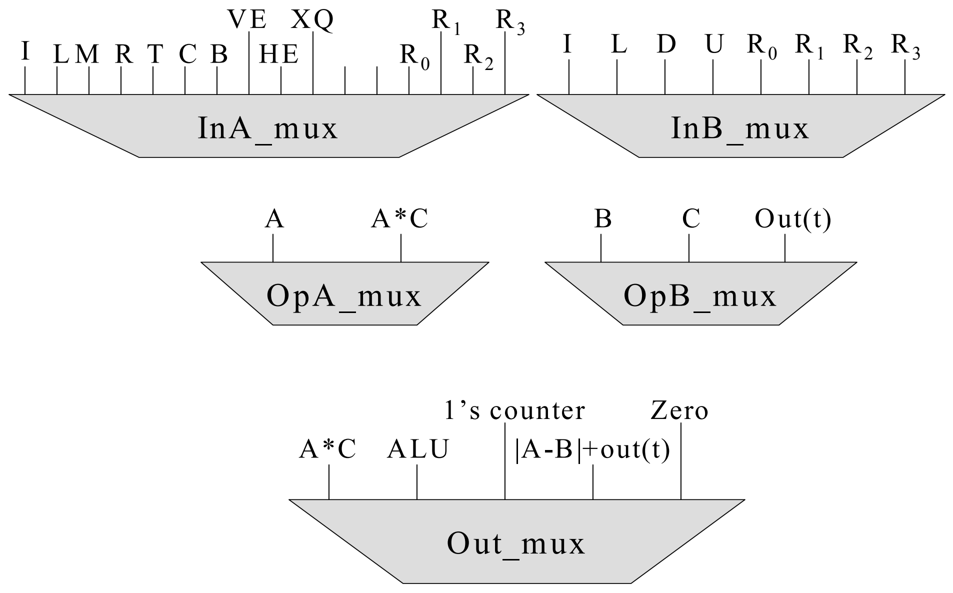


Figure 20. Schematic of the Multiplexers

InA_mux (16-to-1 mux) and InB_mux (8-to-1 mux) select the data from the output of Frame Buffer (I), the outputs of other RCs through the RC Array interconnection network, and the outputs of the four RC local registers.

Other inputs to InA_mux and InB_mux in Figure 20 are as follows:
L (left-A), M (middle), R (right), T (top), C (center), B (bottom), U (up), D (down), L (left-B), VE (vertical express lane), HE (horizontal express lane), and XQ (cross quadrant RC). The first nine inputs are directly driven by nine specific RCs in the array, and the last three inputs come from three buses which facilitate some global inter-RC connections. For more details refer to [6].

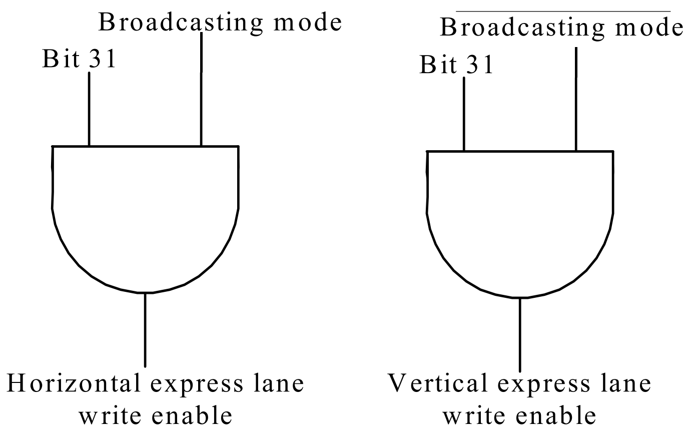
OpA_mux and OpB_mux provide the two operands for ALU. OpA_mux (2-to-1 mux) selects *A*C* when the operation involved is multiply-accumulate (MAC), or *A* directly from the output of inA_mux for the basic two-operands function. OpB_mux (4-to-1 mux) selects *B* or *C* for basic two-operands operation based on the context mode (with or without constant), or selects *Out(t)* for the MAC operation.

Out_mux (8-to-1 mux) selects the outputs from all of the functional units in RC and sends the results to the output register. A zero input to the Out_mux is used to reset the output register.

2.1.8. RC Decoder

The 32-bit context register contains the context word for configuring each RC. This 32-bit configuration data goes through the RC decoder to generate control signals for all the components in the RC (e.g., multiplexers, multiplier, ALU, shifter, etc.). In this section, each field of the context word and its associated decoder design will be discussed.

Bit 31 of the context word controls the writing of the express lane. There are two sets of express lanes in the RC Array interconnection network—the horizontal express lane and the vertical express lane. When this bit is set to 1, the output of the RC will be written to the horizontal/vertical if the column/row context broadcasting mode is used in that clock cycle. This process is implemented by using tristate buffers to connect the output register to the express lane, and having bit 31 of the context word served as the enable signal of the tristate buffers. Figure 21 shows how the enable signal is generated.


Broadcasting mode
Bit 31
$\overline{\text{Broadcasting mode}}$
Bit 31
Horizontal express lane write enable
Vertical express lane write enable

Broadcasting mode = 1 when column context broadcasting
Broadcasting mode = 0 when row context broadcasting

Figure 21. Control of the Express Lane

Bit 30 to bit 28 are used to control the write function of the local registers. There are four local registers in each RC. When bit 30 is high, the output of the RC will be sent to one of its local registers based on the address specified on bit 29 and bit 28. The address decoder is illustrated is Figure 22.

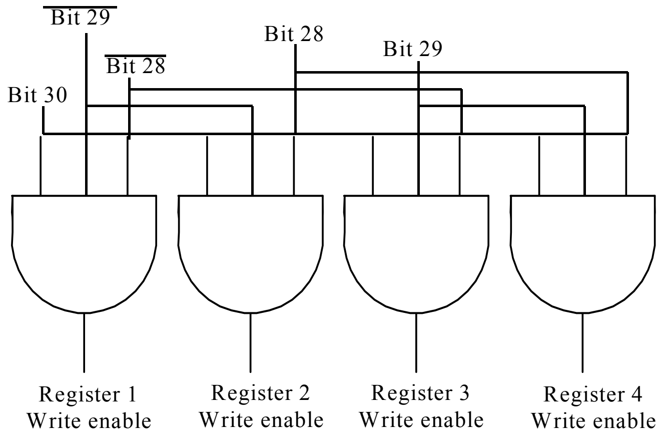


Figure 22. Local Registers Address Decoder

The shift direction and the shift value of the 28-bit logarithmic shifter are specified in bits 27 through 23. Bit 27 controls the shift direction. The shifter will shift left/right if bit 27 equals to 1/0. The shift value is specified in bits 26 through 23. As discussed before, the maximum shift width of the shifter is 15 bits; therefore, we use 4 bits to specify the shift value.

The next two fields, bits 22 through 19 and bits 18 through 16, are used to control the two input multiplexers, inA_mux and inB_mux as discussed previously. Since the 16-to-1 mux and 8-to-1 mux are used in inA_mux and inB_mux, we use 4 bits to control inA_mux and 3 bits to control inB_mux. The decoders for these two multiplexers are similar to what is shown in Figure 22. For inA_mux, 4-input *AND* gates are used to decode the four bits address, while for inB_mux, 3-input *AND* gates are used.

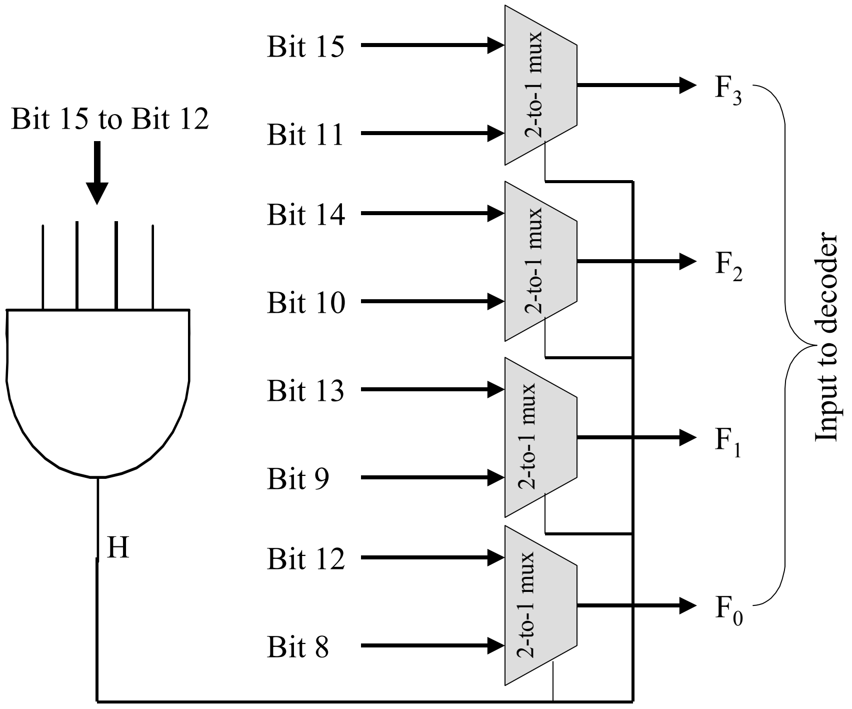


Figure 23. Control Bits Selection

The remaining 16 bits of the context word are used to generate the control signals for the multiplier and the ALU, and to provide the operand (if the constant is used). The decoder uses 4 bits to generate all of the control signals, however, depending on context word definition (with or without constant), the decoder may select bits 15−12 (R_3−R_1 in Table 3) or bits 11−8 (S_3−S_1 in Table 3) as the input. The selection process is implemented as Figure 23.

The multiplier in each RC can be disabled when it is not in use. This feature is realized by inserting tristate buffers at the inputs of the multiplier. As we can see from Table 3, the multiplier is activated only when the context is in constant operation mode and R_3 (bit 15) equals to 1, therefore, $H'F_3$ is employed as the enable signal to the tristate buffers of the multiplier. $H'F_3$ is also used as the control signal for opA_mux, as shown in Figure 20, since the multiplexer will select the input of $A*C$ only when the multiplication is involved (see the OPA column of Table 3).

Another multiplexer, opB_mux, provides another operand to the ALU from one of the three inputs, *B, C,* and *Out(t)* (see the OPB column in Table 3). The implementation of opB_mux with its control can be illustrated in Figure 24.

The RC decoder also provides the control signals for the ALU. As shown in Figure 15, eight signals need to be generated, which are M, C_{in}, $C1$, $C2$, $C3$, $C4$, $C5$, and $C6$. Table 3 also summarizes the relationship between the ALU_OP field of the context word and the control signals M and C_{in}. From Table 3, it can be easily derived that $M = F_2'$.

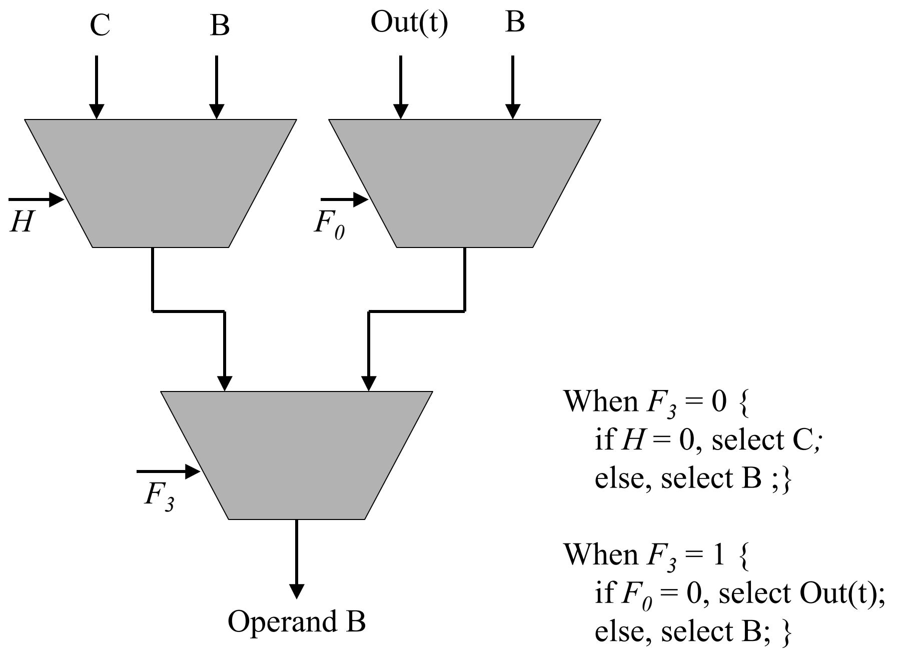


Figure 24. Implementation and Control of opB_mux

As depicted in Table 3, C_{in} depends not only on F_3, F_2, F_1, and F_0, but also on the flag. The flag signal in each RC is set to the value of the most significant bit of operand A (output of inA_mux) when it is required. Figure 25 shows the implementation for generating C_{in}.

Table 6. Truth Table of the ALU Control Signals Decoder

H	F_2	F_1	F_0	F	C1	C2	C3	C4	C5	C6
X	0	0	1	X	1	0	0	0	0	0
X	0	1	0	X	1	1	0	0	1	0
X	0	1	1	X	1	0	0	1	0	0
X	1	0	0	X	1	0	0	1	0	0
X	1	0	1	0	1	0	0	1	0	0
X	1	0	1	1	0	1	0	0	1	0
X	1	1	0	X	0	1	0	0	1	0
X	1	1	1	X	0	0	1	0	0	1
0	0	0	0	X	1	0	1	0	0	0
1	0	0	0	X	1	1	0	0	0	0

$$C1 = F_2' + F_1'F_0' + F_1'F \qquad (1)$$

$$C2 = F_1F_0' + F_2F_1'F_0F + HF_2'F_0' \qquad (2)$$

$$C3 = F_2F_1F_0 + H'F_2'F_1'F_0' \qquad (3)$$

$$C4 = F_2'F_1F_0 + F_2F_1'F_0' + F_2F_1'F' \qquad (4)$$

$$C5 = F_1F_0' + F_2F_1'F_0F \qquad (5)$$

$$C6 = F_2F_1F_0 \qquad (6)$$

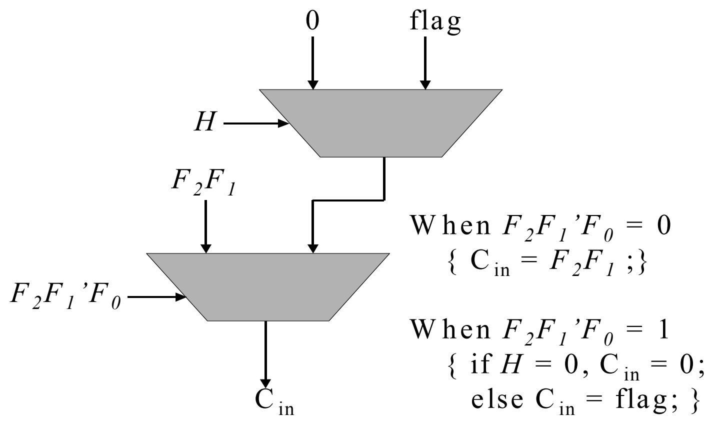

0
flag
H
F_2F_1
$F_2F_1'F_0$
C_{in}
When $F_2F_1'F_0$ = 0 { C_{in} = F_2F_1 ;}
When $F_2F_1'F_0$ = 1 { if H = 0, C_{in} = 0; else C_{in} = flag; }

Figure 25. Decoder for C_{in}

The other six control signals, *C1, C2, C3, C4, C5,* and *C6*, determine the function of the ALU. These six signals are generated by F_2, F_1, F_0, along with *H* and flag *(F)*. Table 6 shows the truth table of the decoder logic. The results derived from Table 6 are depicted in Equations 1−6.

The control signals for the output multiplexer (out_mux) are also provided by ALU_OP context field through RC decoder. An 8-to-1 multiplexer is used for the 7-input out_mux, and therefore, three control bits need to be generated. Table 7 shows the truth table of out_mux decoder logic and the associated out_mux output. The inputs to the decoder are H, F_3, F_2, F_1, and F_0, and the outputs are three control bits, m_2, m_1, and m_0. Since there are only seven inputs, both $m_2m_1m_0 = 000$ and $m_2m_1m_0 = 001$ are reserved to select ALU output.

Table 7. Truth Table of out_mux Decoder Logic

H	F_3	F_2	F_1	F_0	m_2	m_1	m_0	**Output**
0	0	X	X	X	0	0	X	ALU
0	1	0	X	X	0	1	0	A*C
0	1	1	X	X	0	0	X	ALU
1	0	X	X	X	0	0	X	ALU
1	1	0	0	X	0	1	1	Out(t)
1	1	0	1	X	1	0	0	1's counter
1	1	1	0	X	1	0	1	Rounding
1	1	1	1	0	1	1	0	| A−B | + Out(t)
1	1	1	1	1	1	1	1	Zero

2.1.8.1 RC Simulation and Critical Path Analysis

The critical path of the RC, which is also the critical path of the MorphoSys M1 chip, includes a 16-to-1 mux, a 16 by 12 bits multiplier, a 4-to-1 mux, a 28-bit ALU, an 8-to-1 mux, and a 16-bit shifter (Figure 26). We have performed SPICE simulations for the entire RC. In order to consider the effects of long wires and large fanout, the maximum possible load of each RC is computed and replaced by the equivalent capacitance in the SPICE input file. The critical path delay of the RC is 9.5 ns. Each RC consumes 200 mW of power at 100 MHz (25 ^{o}C) when the multiplier is activated.

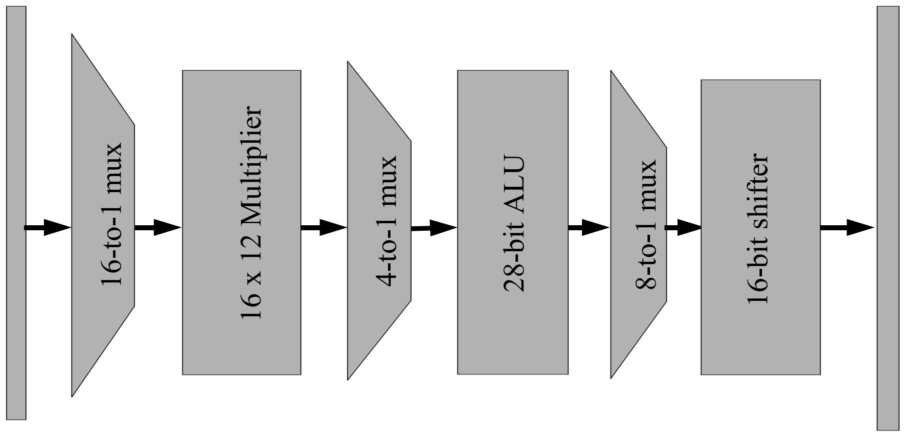


Figure 26. Critical path of RC

2.1.9. TinyRISC

TinyRISC [1] is a simplified 32-bit RISC processor, which has four pipeline stages: fetch, decode, execute, and write-back stages, as shown in Figure 27. It supports data forwarding to eliminate the data hazards between pipeline stages. It also uses delayed branch to avoid branch penalties. The detailed description of each stage is presented next.

Figure 28 shows the schematic of the fetch stage. Its major task is to read one instruction from instruction memory using the address in the program counter (PC) and place it in the pipeline register. The PC address is incremented by four and loaded back to the PC for the next clock cycle. The incremented address is also placed in the pipeline register in case it is needed later for a branch instruction.

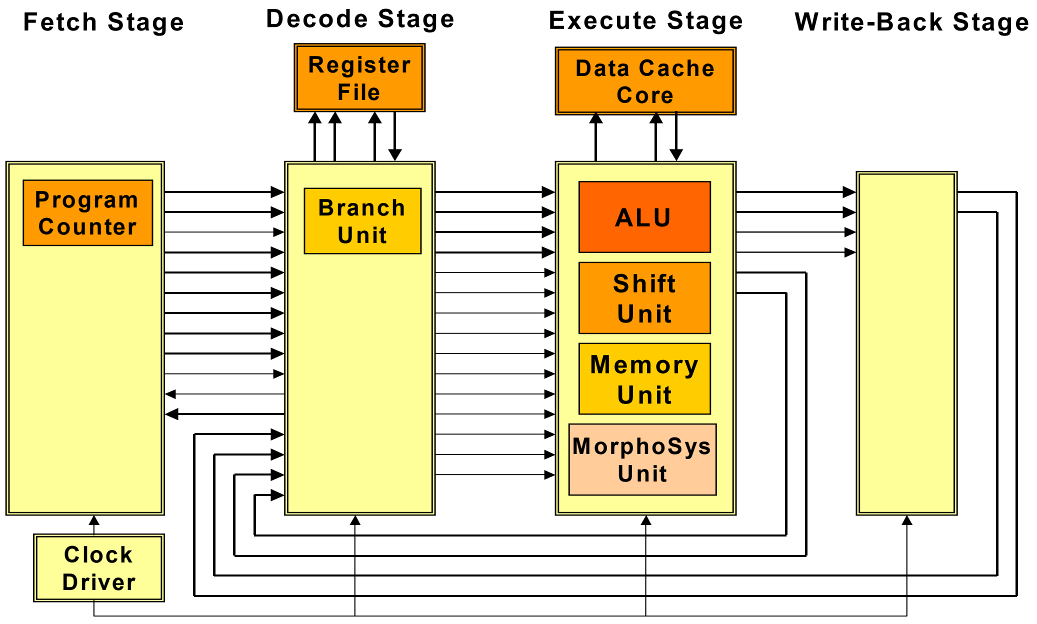


Figure 27. TinyRISC architecture

Figure 27 is the schematic view of the decode stage. It reads two 32-bit numbers from the register file based on the address specified in the *SRC1* and *SRC2* fields of the instruction and supplies the 16-bit immediate field that is extended to a 32-bit number. All three numbers are placed in the pipeline register and will be processed later in the execute stage. The branch instruction is processed by the branch unit and the correct PC address is selected and stored back to the PC in the fetch stage to fetch the next instruction.

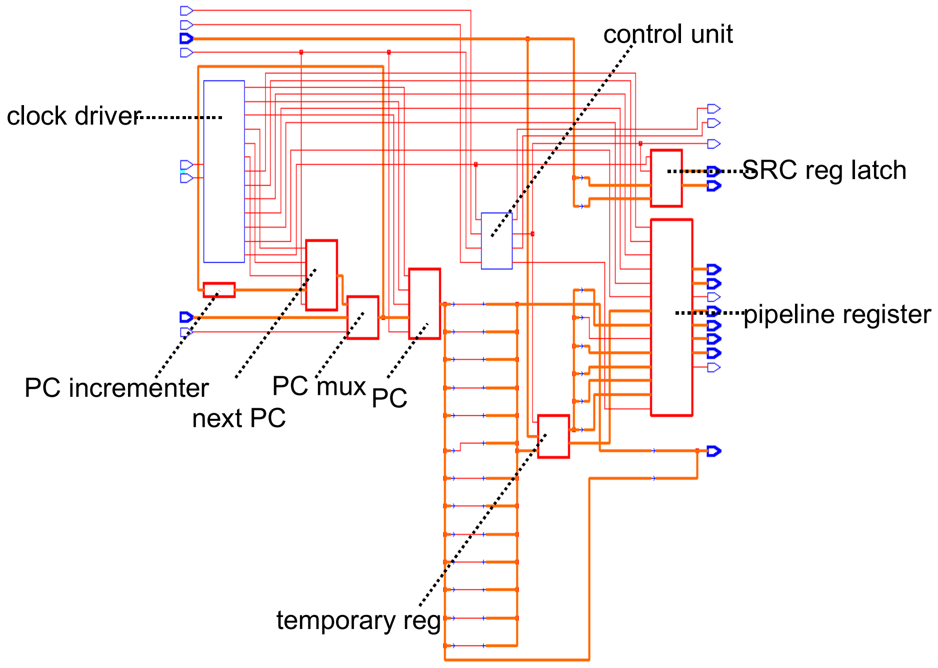


Figure 28. Schematic View of Fetch Stage

The register file consists of 16 registers. It is a standard SRAM-based design with precharge and sense amplifier. Each SRAM cell has one write port and two read ports. The register file is written in the first half of the clock cycle and read in the second half, which supports data forwarding.

The execute stage (Figure 30) consists of an ALU, a shift unit, a memory unit, and a MorphoSys unit. The major task of the execute stage is to perform arithmetic, logic, or memory access operations based on the control information specified in the opcode field of the instruction. The opcode also generates control signals that select the two register file outputs or the immediate field in the instruction as the operands.

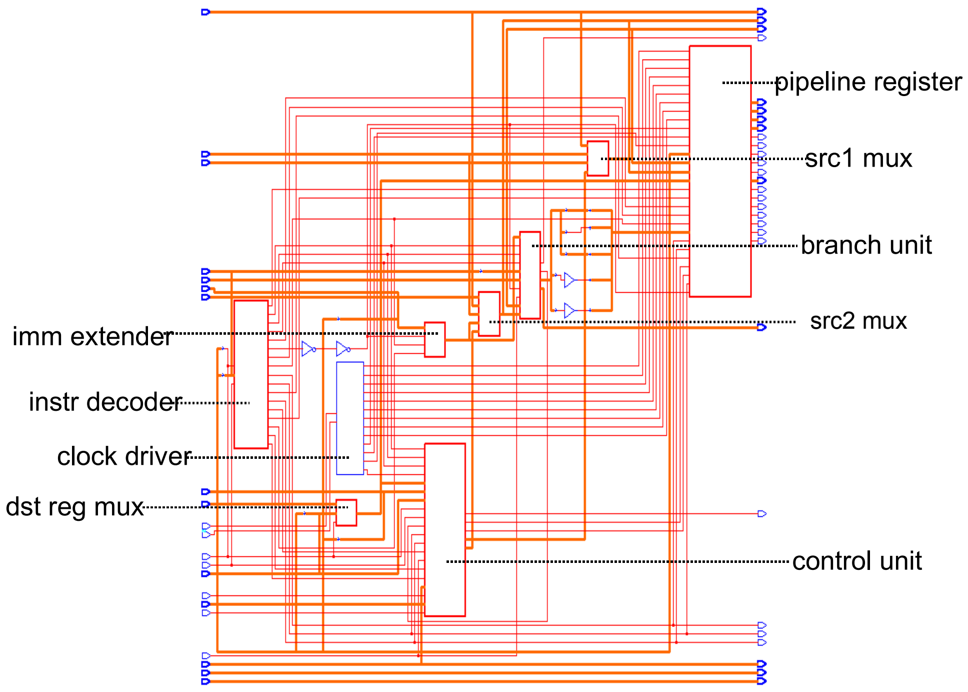


Figure 29. Schematic View of Decode Stage

The MorphoSys decoder, a subcomponent in the execute stage, decodes the TinyRISC instructions that are specifically for MorphoSys. As illustrated in Figure 31, the MorphoSys decoder activates the DMA Controller to transfer data, provides control signals to the RC Array to execute operations defined by the configuration context. It also establishes communication among TinyRISC and DMA Controller, Frame Buffer, Context Memory, and RC Array.

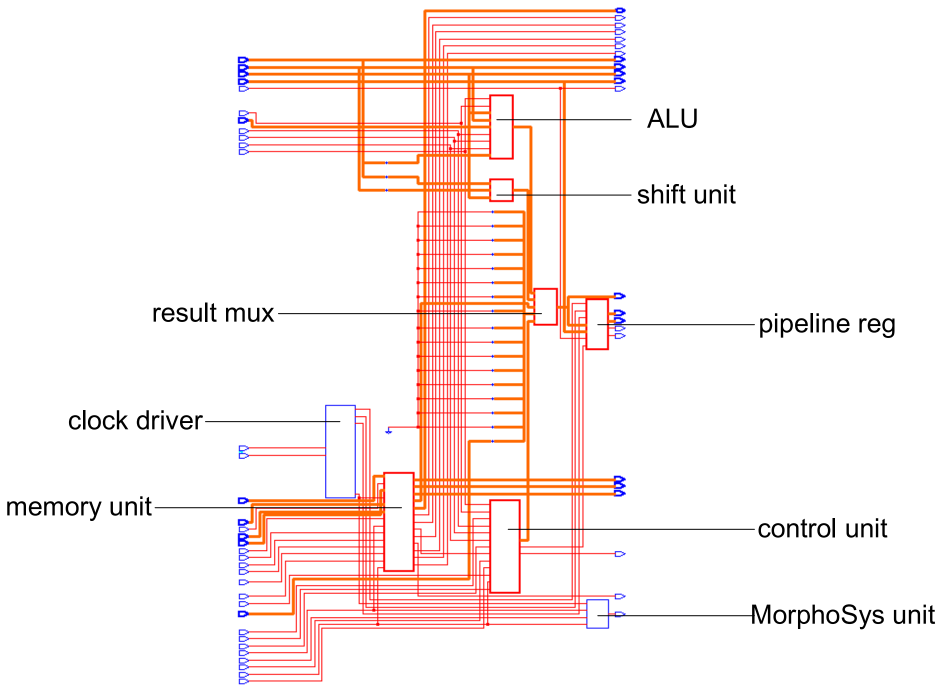


Figure 30. Schematic View of Execute Stage

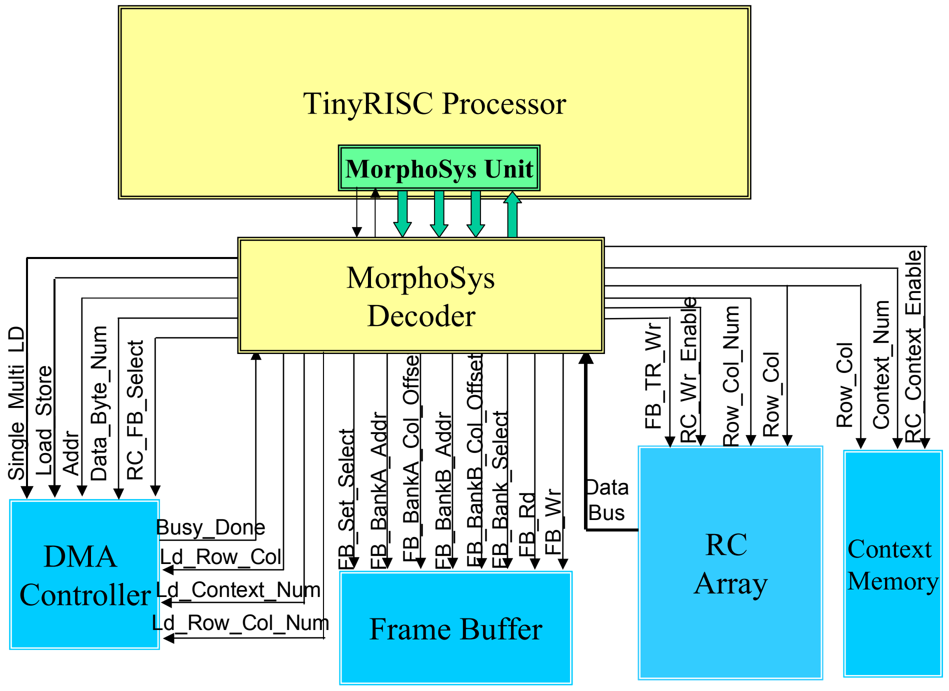


Figure 31. MorphoSys Decoder Control Mechanism

The memory unit consists of a data cache and a cache controller that controls the access of the data cache and the communication between the main memory and the data cache. The size of the data cache is 512 bytes and the cache line size is 4 words. It is a direct mapped cache with write back scheme. Figure 32 shows the block diagram of the data cache.

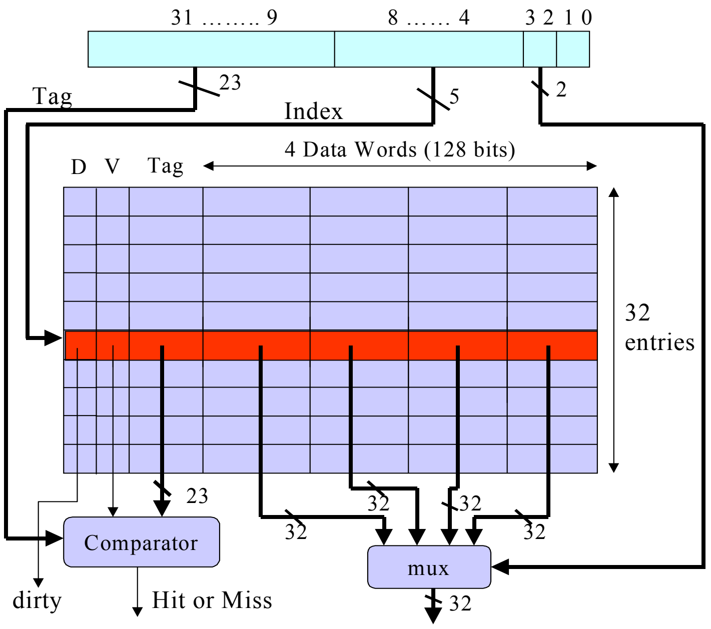


Figure 32. Data Cache Block Diagram

The dirty bit and valid bit of each entry are set to "0" in the initial phase. This is to make sure that the first access to each cache line is always a miss and incorrect write back (to main memory) does not happen.

All the TinyRISC components except for the register file and data cache are synthesized using Synopsys and Mentor Graphics tools and the HP 0.35-µm standard cell library developed by the Air Force Research Laboratory at Wright Research Site.

2.1.10. Context Memory

The Context Memory stores the configuration program (context) for the RC Array. The Context Memory is logically organized into two *context blocks*, each block containing eight *context sets*. Each context set has 16 *context words*.

The major focus of the RC Array is on data-parallel applications, which exhibit a definite regularity. Following this principle of regularity and parallelism, the context is broadcast on a row/column basis. The context words from one context memory block are broadcast along the rows, while context words from the other block are broadcast along the columns. Each block has eight context sets and each context set is associated with a specific row (or column) of the RC Array. The context word from the context set is broadcast to all eight RCs in the corresponding row (or column). Thus, all RCs in a row (or column) share a context word and perform the same operations, and each row (column) of the RC Array receives a context word every clock cycle, from the Context Memory.

The Context Memory is implemented using a standard CMOS SRAM cell with one read port and one write port. The block diagram of the Context Memory is shown in Figure 4. Corresponding to either row/column broadcast of the context word, a set of eight context words can specify the complete configuration (*context plane*) for the RC Array. As there are 16 context words in a context set, up to 16 context planes may be simultaneously resident in each of the 2 blocks of Context Memory.

Context Memory supports dynamic reconfiguration. When the Context Memory needs to be changed in order to perform some different part of an application, the context update can be performed concurrently with RC Array execution. This dynamic reconfiguration enables the reduction of effective reconfiguration time to zero.

Another important feature of Context Memory is selective context enabling. This implies that only one specific row or column may be enabled for operation in the RC Array. This feature is primarily useful in loading data into the RC Array. Since context can be used selectively, and because data bus design allows loading of one column at a time, one set of context words can be used repeatedly to load data into all eight columns of the RC Array. Without this feature, 8 context planes (out of 32 available) would be required just to read/write data. This feature also allows irregular operations, such as zigzag rearrangement of array elements in RC Array

2.1.11. Frame Buffer

The Frame Buffer serves as a data cache for the RC Array. It consists of two sets of identical data memory (see Figure 3). Each set consists of two banks of memory with each bank having 64 by 8 bytes of storage. By using these two sets alternatively, the computation of the RC Array is overlapped with the load and store of the Frame Buffer, which makes the memory accesses transparent to the RC Array. MorphoSys performance benefits greatly from the streaming process of this data buffer.

The Frame Buffer is byte addressable. An important feature of the Frame Buffer is the ability to provide any eight consecutive bytes of data to the RC Array in one clock cycle. As shown in Figure 33, the Frame Buffer is implemented using an SRAM cell with two read ports and one write port.

To access eight consecutive bytes of data, the decoder enables the first read port of the associated decoded row and the second read port of the row next to the decoded row address. Then, these two rows of data are concatenated and a barrel shifter is used to select the desired eight bytes based on the column address. Figure 34 shows the block diagram of the Frame Buffer.

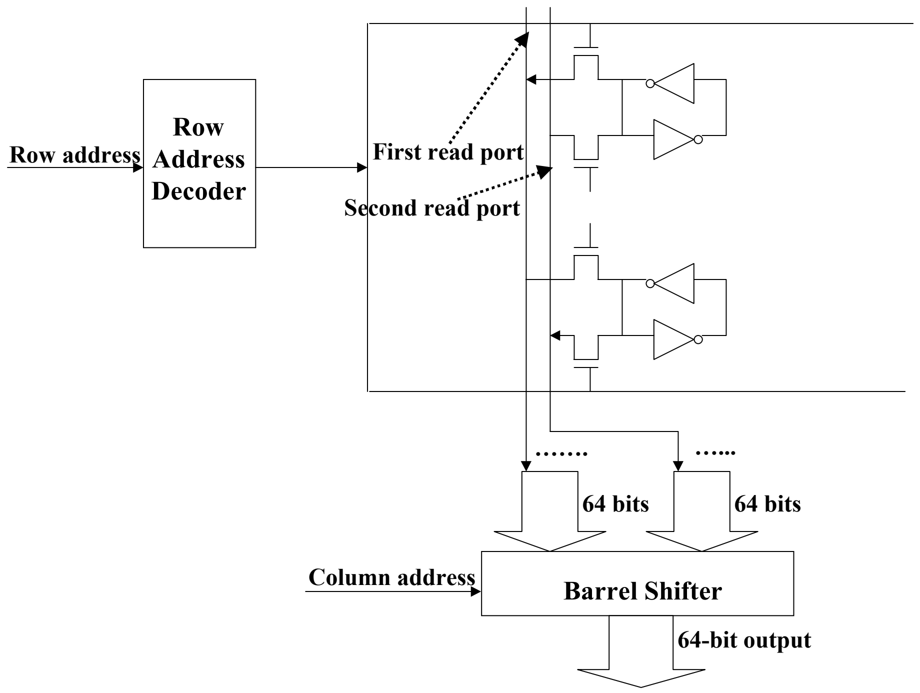


Figure 33. Frame Buffer Implementation

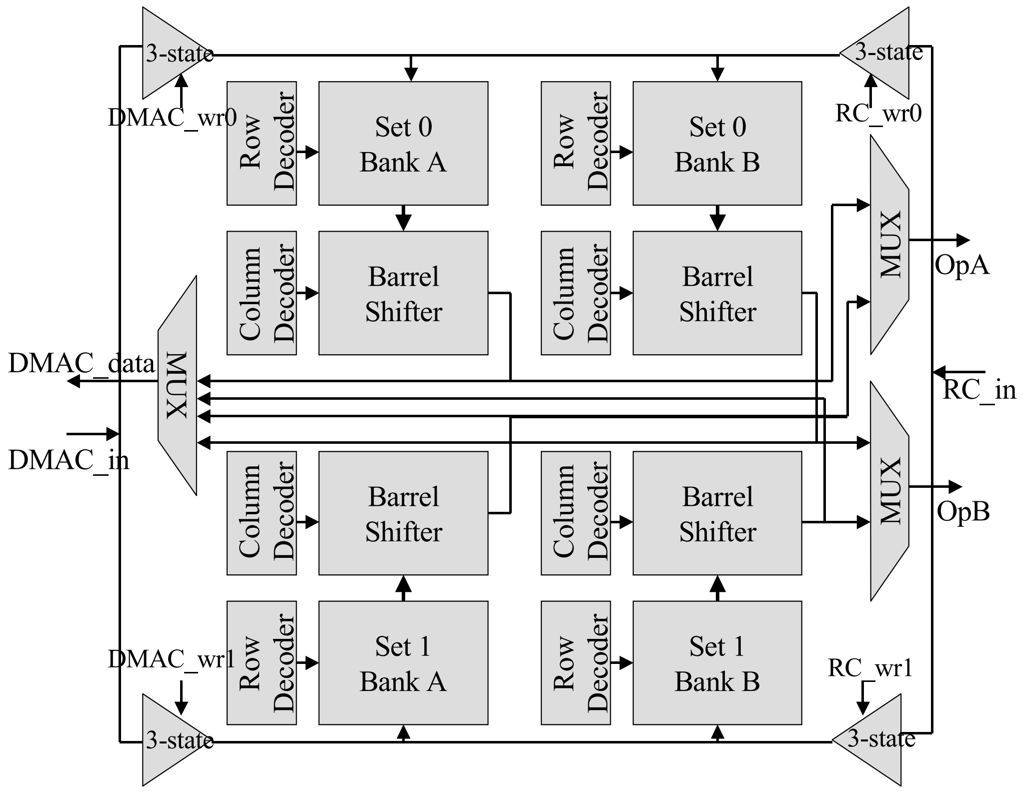


Figure 34. Frame Buffer Block Diagram

2.1.12. DMA Controller

The DMA controller (DMAC) block handles all data/context transfers between Context Memory, Frame Buffer, and main memory. Three TinyRISC instructions for MorphoSys are used to direct the operations of the DMAC. The DMAC consists of three components: DMAC state machine, data register unit (DRU), and address generator unit (AGU) (Figure 2.35). The DRU is used to pack or unpack data since the bus width between main memory and the DMAC (32 bits) is different from the bus width between DMAC and Frame Buffer (64 bits). The AGU generates the addresses for the main memory and Frame Buffer when reading or writing Frame Buffer and Context Memory addresses during context loading.

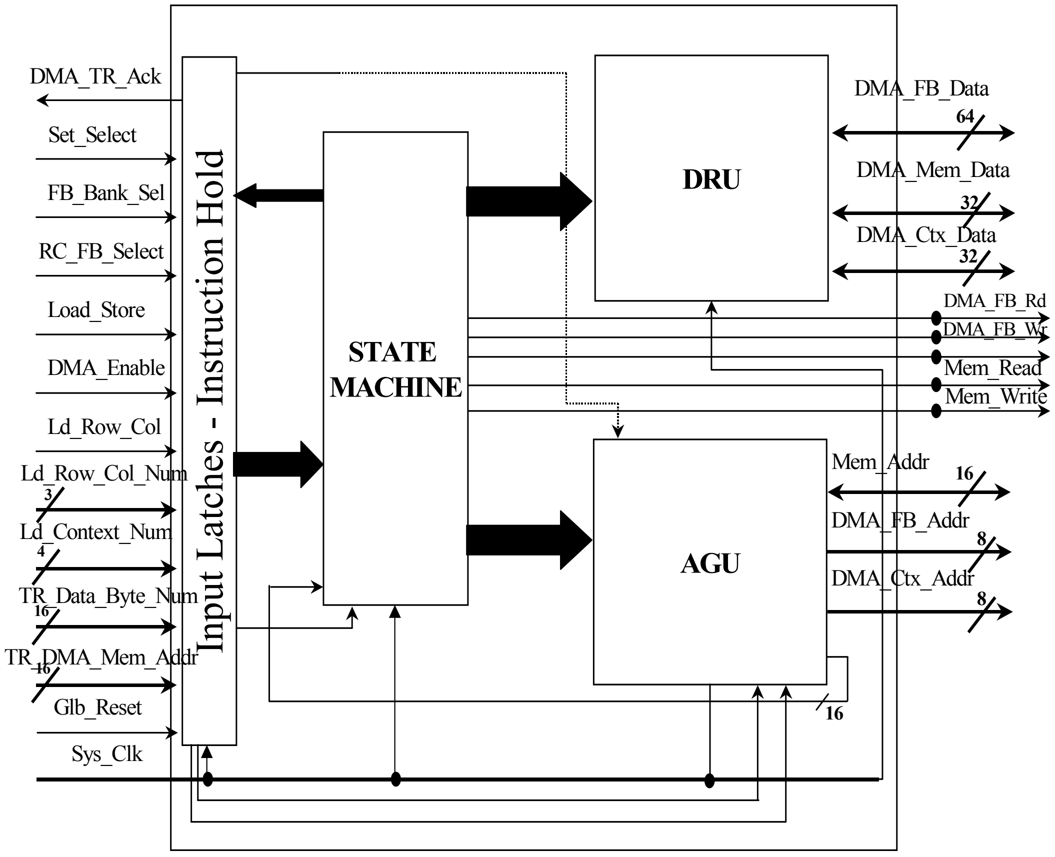


Figure 35. Internal Block Diagram of DMAC

2.2. RC Array Interconnection Network and Global Routing

The global routing network consists of three parts: RC Array interconnection and data/context bus network, clock tree, and power/ground network. The RC interconnection network is comprised of three hierarchical levels.

There are three levels of the RC Array interconnection network. The first level of the RC Array interconnection network is the nearest neighbor layer that connects the RCs in a 2-

D mesh (Figure 36). The second layer of connectivity is at the quadrant level (a quadrant is a 4 by 4 RC group), which provides complete row and column connectivity within a quadrant. Therefore, each RC can access data from any other RC in the same row/column within the quadrant. At the highest or global level, there are buses that support interquadrant connectivity (Figure 37). These buses are also called express lanes and they run across rows as well as columns. These lanes can supply data from any one RC (out of four) in a row (or column) of a quadrant to other RCs in same row (or column) of the adjacent quadrant. Thus, up to four cells in a row (or column) may access the output value of any one of four cells in the same row (or column) of the adjacent quadrant. The express lanes greatly enhance global connectivity. Even irregular communication patterns, that otherwise require extensive interconnections, can be handled quite efficiently. For example, an eight-point butterfly is accomplished in only three clock cycles.

A 128-bit data bus from the Frame Buffer to the RC array is linked to the column elements of the array. As illustrated in Figure 3, each bank of Frame Buffer provides 8 consecutive bytes of data, and a total of 16 bytes are interleaved and sent to the RC Array through a 128-bit bus.

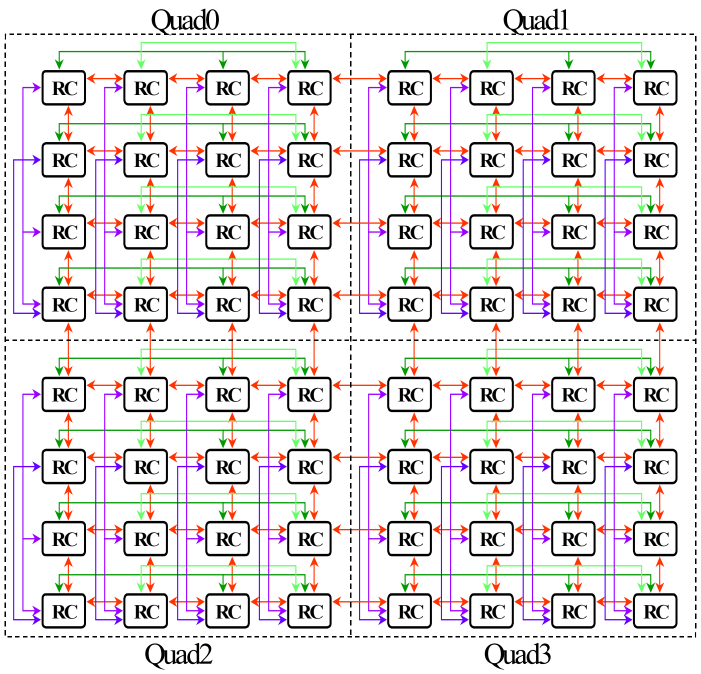


Figure 36. Levels 1 and 2 of Interconnection Network

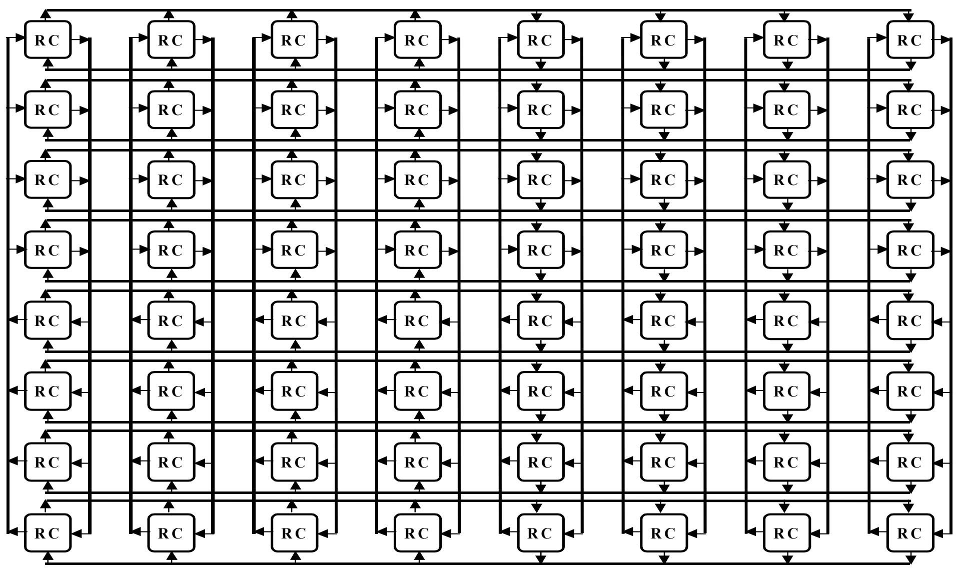


Figure 37. Level 3 of Interconnection Network (Express Lanes)

As shown in Figure 38, each RC in one column receives two operand data (operand A and operand B). It is possible to load the entire RC Array with the same set of 128-bit data in one cycle. To have each RC receive different data, eight cycles are required to load the entire RC Array. The outputs of the RC elements of each column are written back to the Frame Buffer through a 64-bit data bus. Only the lower 8 bits of the output data of each RC are sent back to the Frame Buffer.

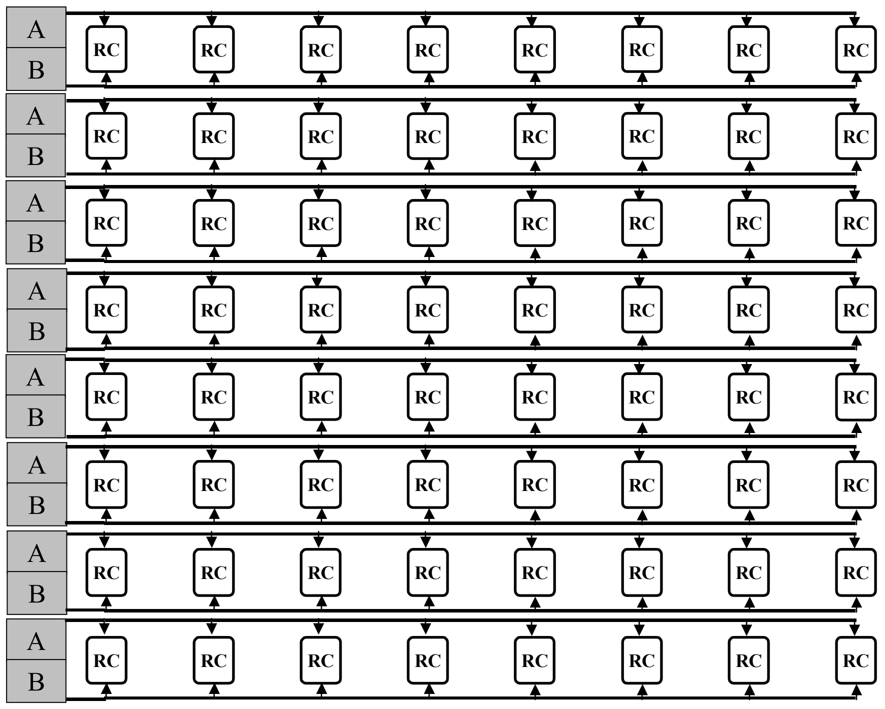


Figure 38. Frame Buffer to RC Array Data Bus

When a TinyRISC instruction specifies that a particular group of context words be executed, these must be distributed to the Context Register in each RC from the Context Memory. The context bus communicates this context data to each RC in a row/column depending upon the broadcast mode. Each context word is 32 bits wide, and there are 8 rows (columns). Hence, the context bus is 256 bits wide.

The dense connectivity of the interconnection network makes it difficult for automatic routing tools to maintain regularity of the global routing. A preliminary run of the Mentor Graphics' automatic router gave a highly irregular layout. Hence, the global routing layout was done using a combination of procedural and custom design approach as depicted below:

1. The clock tree was done as a custom layout using an H tree pattern (see Figure 39) with tree-levels of buffers to balance the clock skew of RC array, as shown in Figure 40. The clock delay from the clock source to each register in the RC was measured using SPICE simulation. The buffers were subsequently inserted for the clock path to other components of MorphoSys (e.g., TinyRISC, DMAC, and Frame Buffer) to balance the delay.
2. The minimum width of metal layers (metal 3 and metal 4) were used for power and ground (P/G) network (based on the technology electron-migration rules) and accordingly the routing channel, as shown in Figure 41, was created manually by editing the layout file.
3. There exist some regular patterns for the three levels of interconnection networks. The regular patterns of the interconnection network were captured using function calls that perform the procedural routing for creating the layout. We partitioned the interconnection network into four types of connectivity, which are intraquadrant row/column full connectivity, interquadrant context connectivity, interquadrant express lane connectivity, and cross quadrant boundary connectivity. For intraquadrant row/column full connectivity and inter-quadrant context connectivity, the routing channels are fixed for all RCs. For interquadrant express lane connectivity and cross quadrant boundary connectivity, the channels switch direction when crossing the quadrant boundary. Once the pattern of each RC is figured out, the routing of the interconnection network can be performed easily.

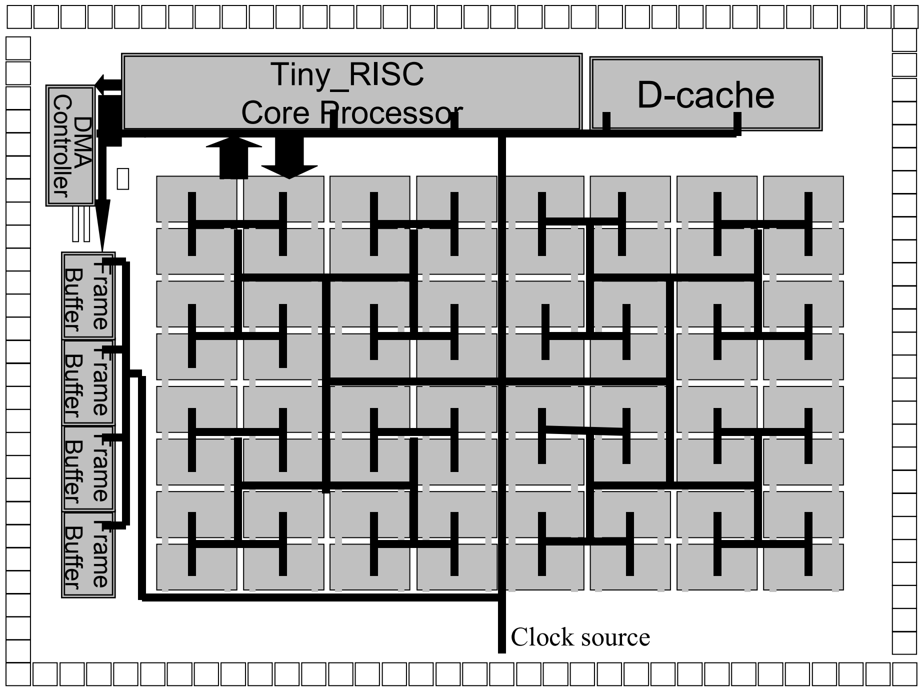


Figure 39. Clock Tree

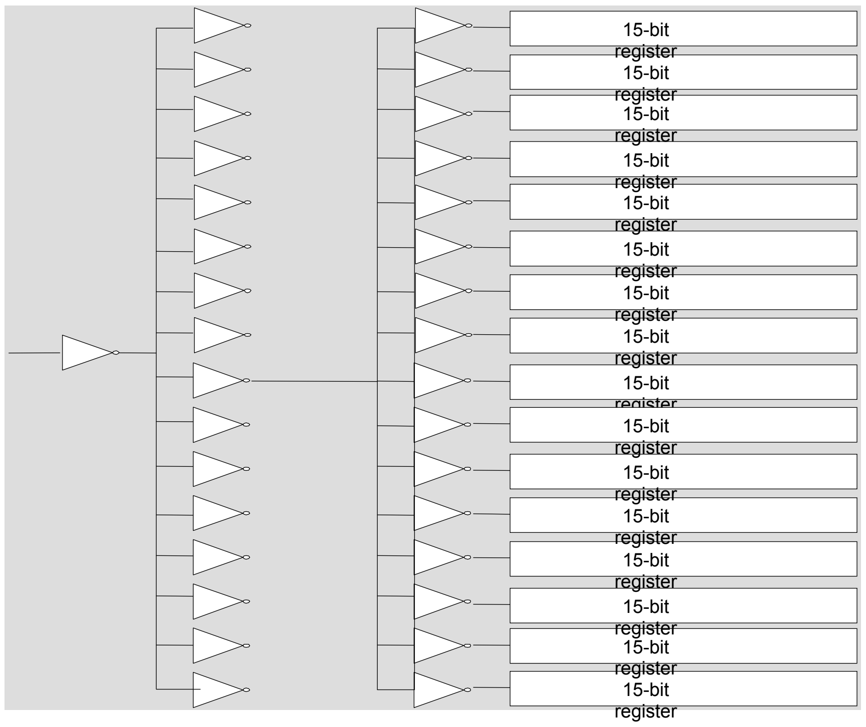


Figure 40. RC Array Clock Tree Buffer

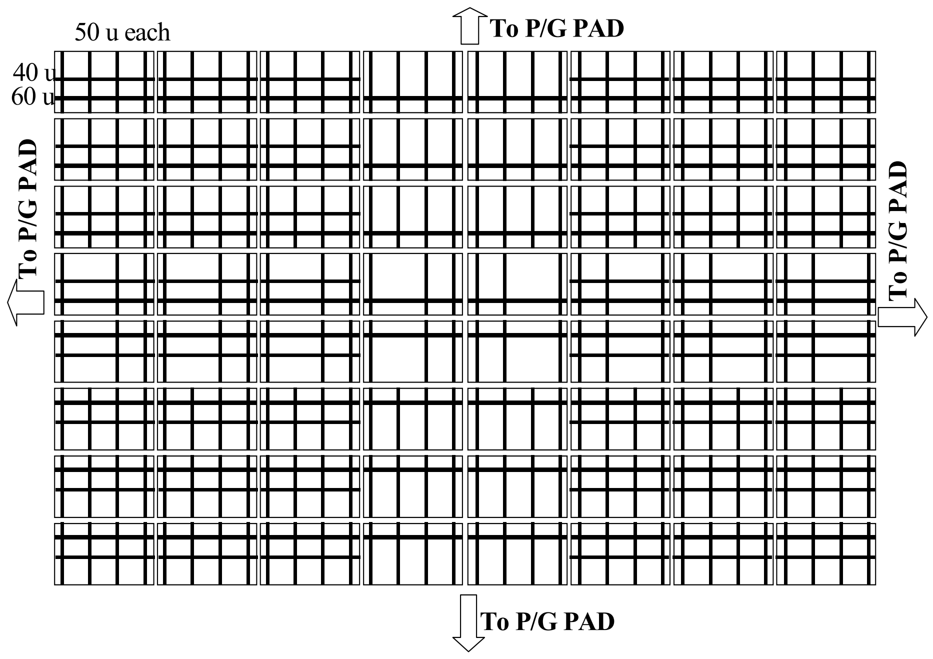


Figure 41. RC Array Power/Ground Network

2.3. The RC Array Layout

Because of dense connectivity and the need for symmetry in routing for the 8 by 8 RC Array, it is difficult for commercial router tools to handle routing for the RC Array properly. The result of an automatic router may not have any regularity, and may cause the clock tree to be unbalanced. Figure 42 shows the routing resulting using MicroRoute of Mentor Graphics. To solve the problem, we specified the desired routing network using L language script file. This generated regular routing result, and enabled the design of a balanced clock H tree for the RC array.

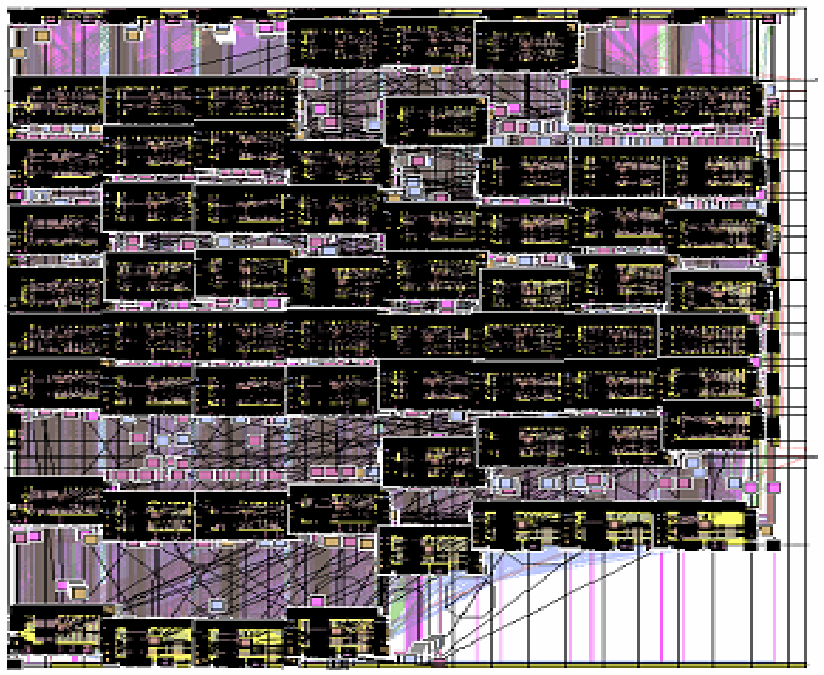


Figure 42. Irregular Routing Result Using Commercial Tools

Since we only used metal 1 and metal 2 for internal routing inside each RC, this allowed us to use metal 3 (vertical direction) and metal 4 (horizontal direction) for interconnectivity between RCs. Once the available vertical and horizontal routing channels are allocated, then these channels are partitioned based on their properties. There are four different kinds of connectivity:

- intraquad row/column full connectivity
- interquad context, power ground and common data bus connectivity
- interquad express lane connectivity
- cross quad boundary connectivity.

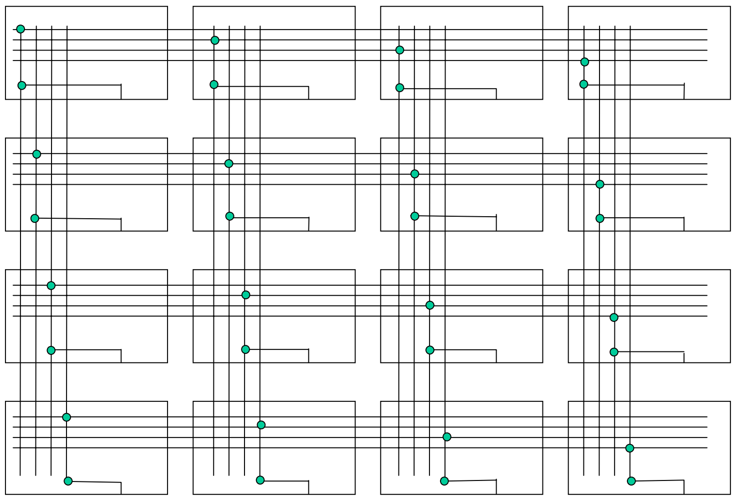


Figure 43. Routing Channel Allocation for the Intraquad Row/Column Full Connection

For the first two cases, the physical routing channels are fixed for each RC (called fixed routing channel). In other words, the channels for this connectivity are the same for different RCs. Figure 43 shows the routing pattern used for the intraquad row/column full connectivity and Figure 44 illustrates the power/ground routing pattern for the fixed interquad connectivity. For the second two cases, they will switch the direction when crossing the quad boundary (Figure 45). For instance, the routing channel for the input horizontal express lane in Quad 0 will be the routing channel for the output horizontal express lane in Quad 1. Since the original symmetric property is kept, we also can keep the balanced clock H tree and clock buffers for the whole RC array so as to minimize the clock skew.

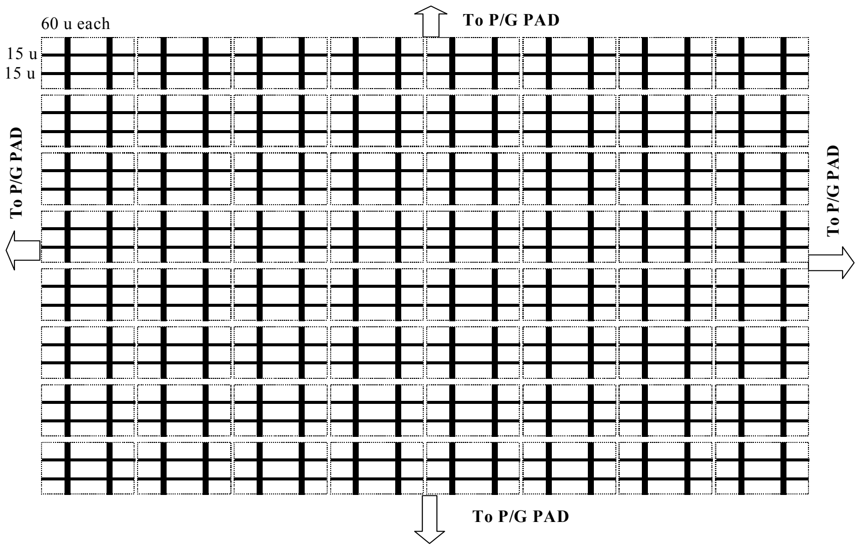


Figure 44. Regular Power/Ground Routing for the RC Array

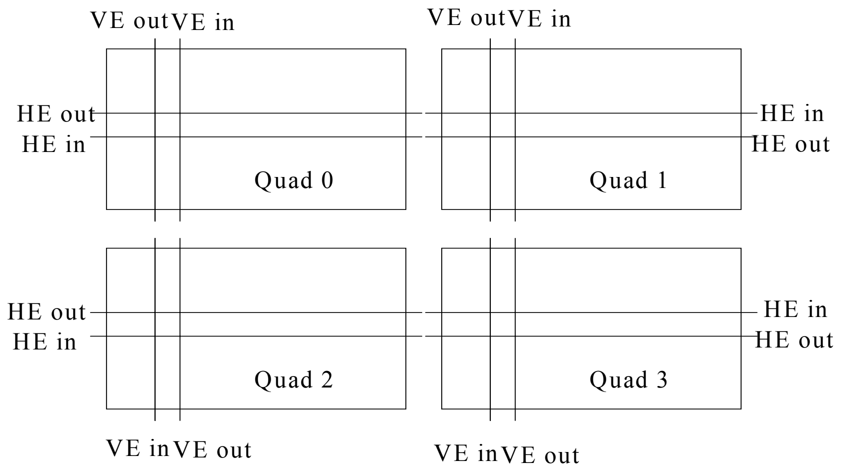


Figure 45. Direction Switching when Crossing Boundary

The routing over each RC is parameterized based on its location in the array. The routing cell is generated using the procedure OVER_RC_ROUTING_TEMPLATE, as follows:

```
CELL OVER_RC_ROUTING_TEMPLATE(ID=?,Quad=?,llx=?,urx=?,lly=?,ury=?)
 {

     INST RC RC AT (0,0);
     Fixed_routing_channel {
                 . . .
         };
     Floating_routing_channel{
         IF(Quad=0) {
             IF(ID=0) {
                 Routing for Quad 0, RC 0; }
             ELSE {
                 . . .
                 }
         ELSE IF (Quad=1) {
             . . .
             }
         }
     Power_ground_routing {
           . . .
           }
     Clock_H_tree_routing {
           . . .
           }
 }
```

llx, lly are the minimum coordinate in X and Y axis, respectively with regard to the origin point, while urx, ury are the maximum coordinate in X and Y. These parameters define a minimum rectangle that can cover the whole RC.

The routing of signals over RC(0) in Quad 0 is shown in Figure 46 and Figure 47 shows routing of signals over RC(0) in Quad 3. Both routings are generated from the same routing template with different location parameter.

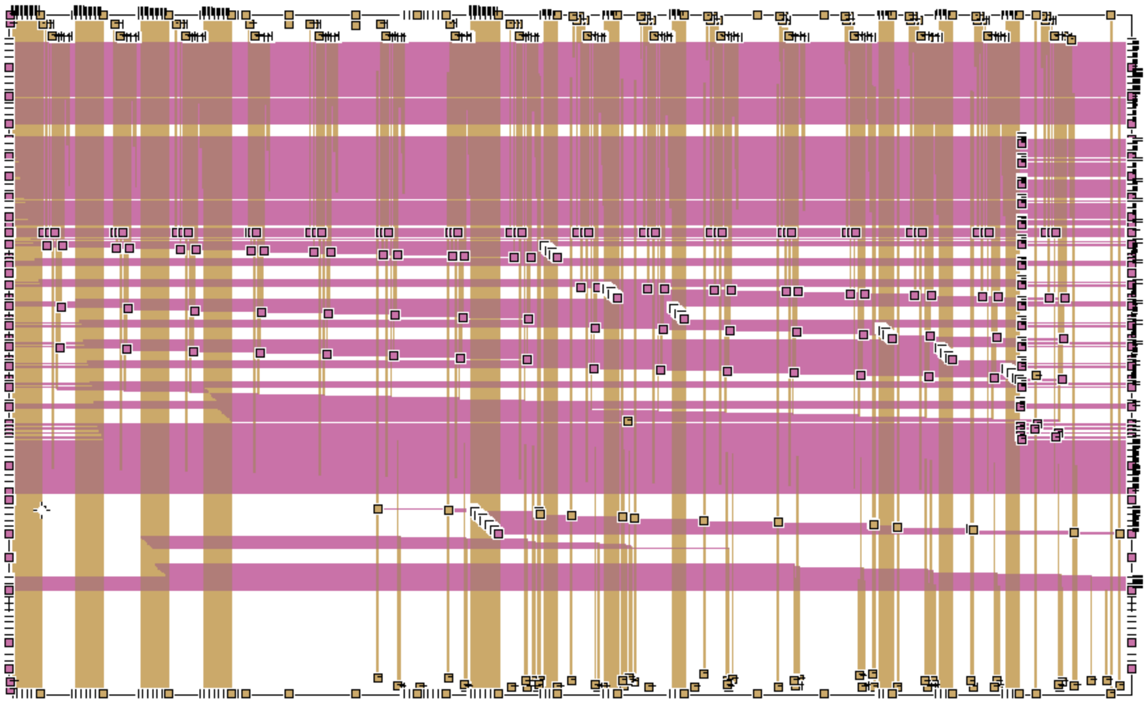


Figure 46. Routings over RC (0) in Quad 0

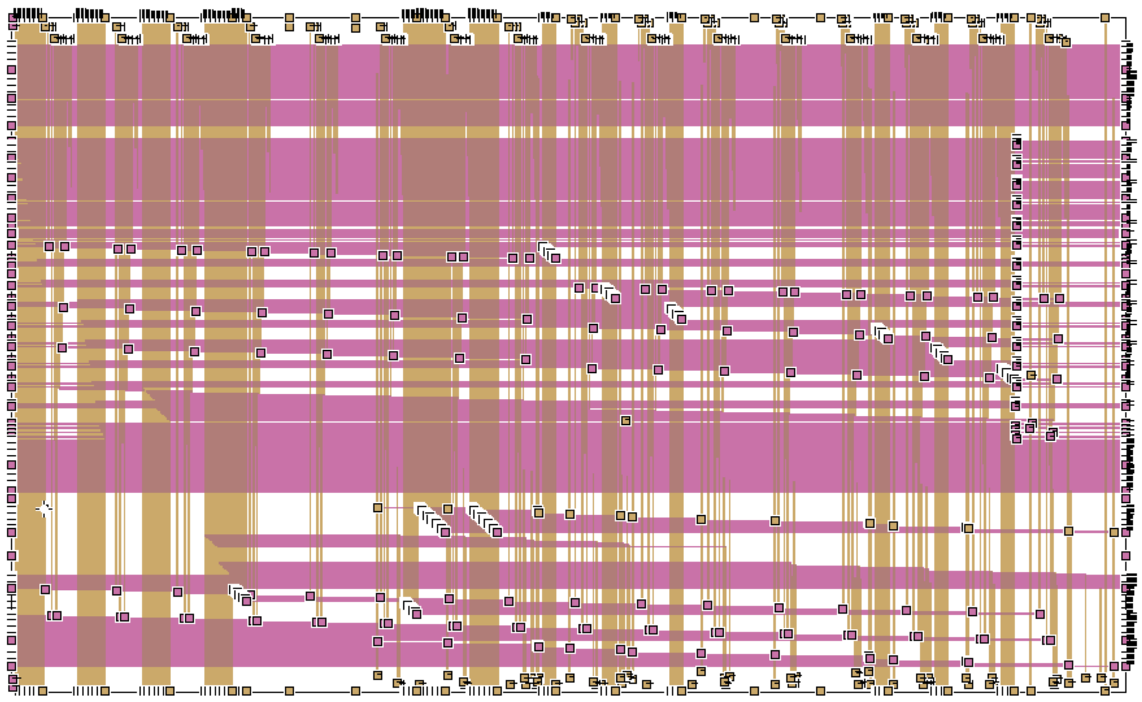


Figure 47. Routing of signals over RC(0) in Quad 3

Once the generic connectivity pattern for the RC is created, they can be tiled to form the routing of 8 by 8 RC Array. The 8 by 8 RC array routing can be generated by instantiating each routing template as following:

```
CELL rc_8x8 ( )
{
# Quad 0
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC00(ID=0,Quad=0);
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC01(ID=1,Quad=0);
  . . .


 # Quad 1
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC10(ID=0,Quad=1);
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC11(ID=1,Quad=1);
  . . .

# Quad 2
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC20(ID=0,Quad=2);
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC21(ID=1,Quad=2);
  . . .

# Quad 3
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC30(ID=0,Quad=3);
  CALL OVER_RC_ROUTING_TEMPLATE CELL RC31(ID=1,Quad=3);
  . . .

    Some other necessary routings;
}
```

2.4. Physical Layout Design Methodology Development

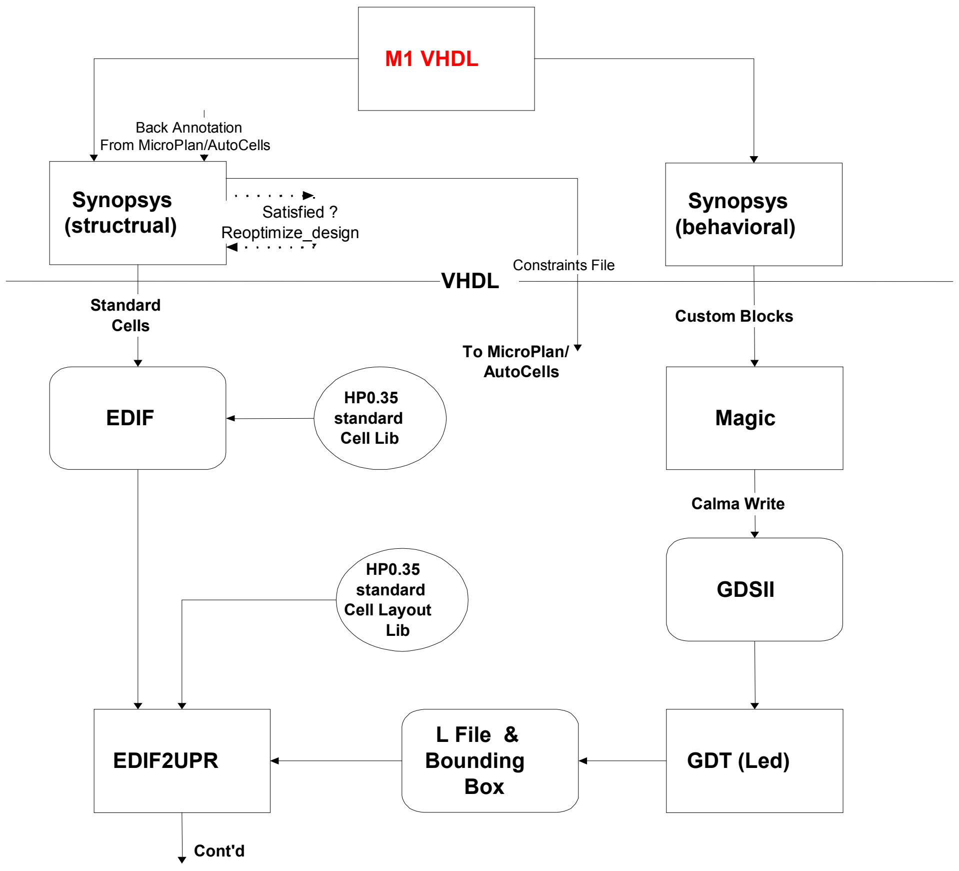


Figure 48. Layout Design Flow for the M1 Implementation

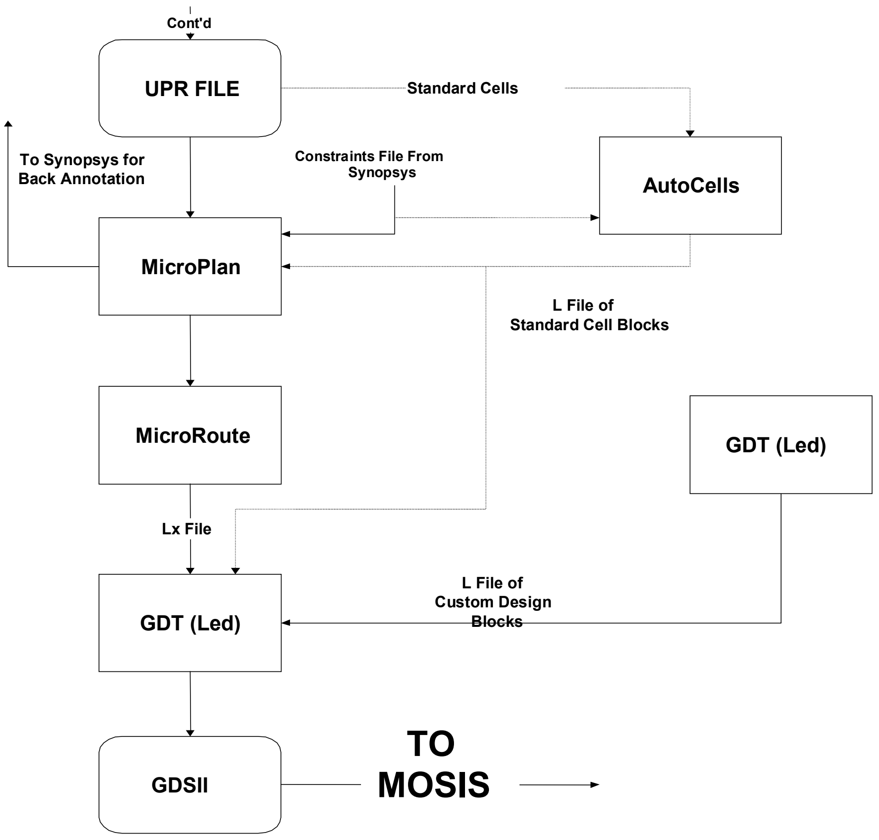

Cont'd
UPR FILE
Standard Cells
AutoCells
To Synopsys for Back Annotation
Constraints File From Synopsys
MicroPlan
L File of Standard Cell Blocks
MicroRoute
GDT (Led)
Lx File
GDT (Led)
L File of Custom Design Blocks
GDSII
TO MOSIS

Figure 49. Layout design flow for the M1 implementation (cont'd)

2.5. MorphoSys M1 Layout and Top-Level Verification

Figure 50 shows the layout picture of MorphoSys M1 chip. The integration of the five components (RC Array, TinyRISC, Frame Buffer, Context Memory, and DMAC) was carried out using Mentor Graphics CAD tools.

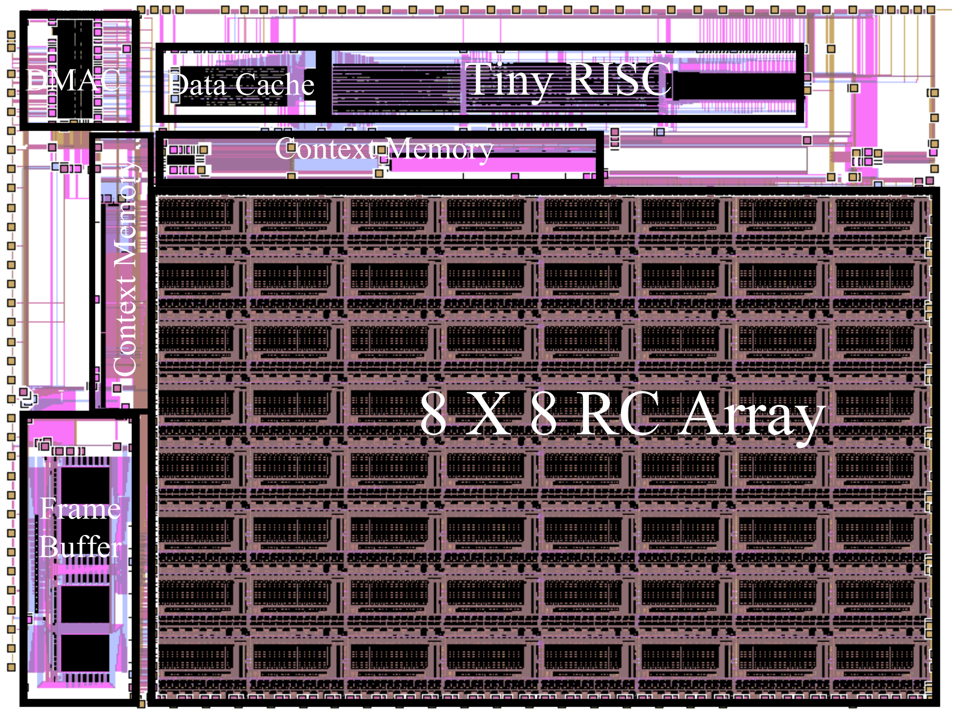


Figure 50. MorphoSys M1 layout picture

The M1 chip area is about 14 by 12 mm^2. The 8 by 8 RC Array, which is the largest component of the MorphoSys M1 chip, occupies more than 80 percent of the chip area. One thing to note is that the two Context Memory blocks (row and column Context Memory blocks) are actually laid out in a distributed form where the eight sets of the column block sit on the top and line up with the eight columns of the RC Array and the row block sit on the left side to line up with the eight rows of the RC Array.

Table 8 lists the transistor counts for each component in the MorphoSys M1 chip, and Table 9 summarizes the important features of M1 chip.

Table 8. Transistor Counts of M1 chip

Component	Transistor Count
RC Array	1,195,392
TinyRISC	94,038
Frame Buffer	139,106
Context Memory	105,096
DMAC	20,594
Main memory controller	1,180
M1	1,577,406

Table 9. Important Features of M1 Chip

Process technology	HP CMOS, 0.35-µm, 3.3-V, four-layer metal
Area	14 mm by 12 mm
Transistor count	1,577,406
Peak performance	6.4 GMAC/S on 16-bit data
Clock Frequency	100 MHz
Power consumption (100 MHz, 25 ^{o}C)	13.5 W in the worst case less than 6 W for DCT less than 5 W for motion estimation
Pin count	240 (132 I/O, 54 power, 54 ground)

One point worth mentioning is the power consumption measurement for M1 chip. The 13.5 W of the power consumption was measured by assuming that MAC operation is performed in all of the RCs every clock cycle. The worst-case input pattern, every bit of the inputs switches in the two consecutive clock cycles, was applied. This measurement gives us the worst-case scenario for the M1 chip power consumption. However, among the applications we have investigated, there is no such case that the multiplier of each RC is activated every clock cycle. In order to get a more realistic measurement, the power consumptions for DCT and motion estimation are also provided in Table 9. The detailed information on the power consumption analysis for M1 will be presented in the next section. The brief description of the power consumption measurement for DCT and motion estimation is provided here. We used Hspice to simulate the power consumption of each RC function, shown in Table 2, with the worst-case scenario, which means each bit of the input switches every clock cycle. The percentage of time for which each RC function is performed in a particular application can be derived by analyzing the TinyRISC assembly code and the context words involved to perform this application. By multiplying the percentage of each RC function with its associated power consumption and accumulating all of the numbers, we can get the power consumption for the application. The results show a difference from the worst case by more than a factor of 2.

3. M1 Chip Testing

Two kinds of test equipment were used for the MorphoSys M1 chip testing. They are ProTest equipment and commercial IMS ATS equipment. The testing methodology of each machine will be briefed in the following sections. For more details on ProTest and IMS please refer to http://www.microlab.ch/industry/protest.html and http://www.ims.com, respectively.

3.1. Generic Testing Methodology

In general, the MorphoSys M1 chip testing focuses on four tasks: 1) functionality test, 2) clock speed test, 3) power consumption test, and 4) working condition test.

Functionality test: Basically, the functionality test answers the question of whether or not the M1 chip is working under an optimal condition. Table 10 shows the programs for the functionality testing. It exercises all of the major components of M1 and the associated interconnections. The first seven programs (from simple to complex) have to be run in the correct order in order to analyze the faulty point quickly if something is wrong. The testing conditions for the functionality test is 3.3-V supply voltage at room temperature or below, and clock frequency set to a reasonable low frequency (from 6 MHz to 20 MHz). The low frequency will remove the error possibility related to the clock frequency.

Table 10. Test program for M1 chip

	Programs	Objectives
1	TinyRISC ALU	To test the very basic TinyRISC ALU operations, such as addition, subtraction.
2	TinyRISC sort	To test the complete operations of TinyRISC. The sort program implements a heap sort algorithm.
3	DMA-FB	To test the data transfer between external memory with Frame Buffer through DMAC.
4	DMA-CM-RC-TinyRISC	To test the context loading from external memory to Context Memory through DMAC, very basic RC operations and the data transfer from RC array to TinyRISC.
5	DMA-FB-RC-TinyRISC	To test the Frame Buffer reading operations from RC Array side.
6	DMA-FB-RC-FB-DMA	To test the Frame Buffer writing from RC Array.
7	FB-SHIFT	To test the shift operation in the Frame Buffer.
8	DCT program	To test the complete operation of DCT. This test will cover the reading/writing of one bank of Frame Buffer from both DMA side and RC Array side, Context Memory loading, very complex RC operations (multiplication, shift, addition, etc.), data exchange via express lane.
9	Motion Estimation	To test the complete operation of Motion Estimation. This test will cover the reading of two bank of Frame Buffer simultaneously, some RC operations (absolute value, accumulation, etc) and data transfer from RC array to TinyRISC.
10	ATR	To test the complete operation of ATR. This test will cover the reading/writing of one bank of Frame Buffer from both DMA side and RC Array side, Context Memory loading, some RC operations (one's count shift, addition, etc.).

As shown in Table 10, these programs test the basic functions. Some of them are divided into several subcategories in order to detect the possible faulty point or to help to analyze the real behavior of the MorphoSys M1 chip. Figure 51 shows the data transfer between the DMAC external memory and Frame Buffer through the DMAC. It represents data transfer for the DMA-FB program. The diagram of the FB-SHIFT data transfer is shown in Figure 51.

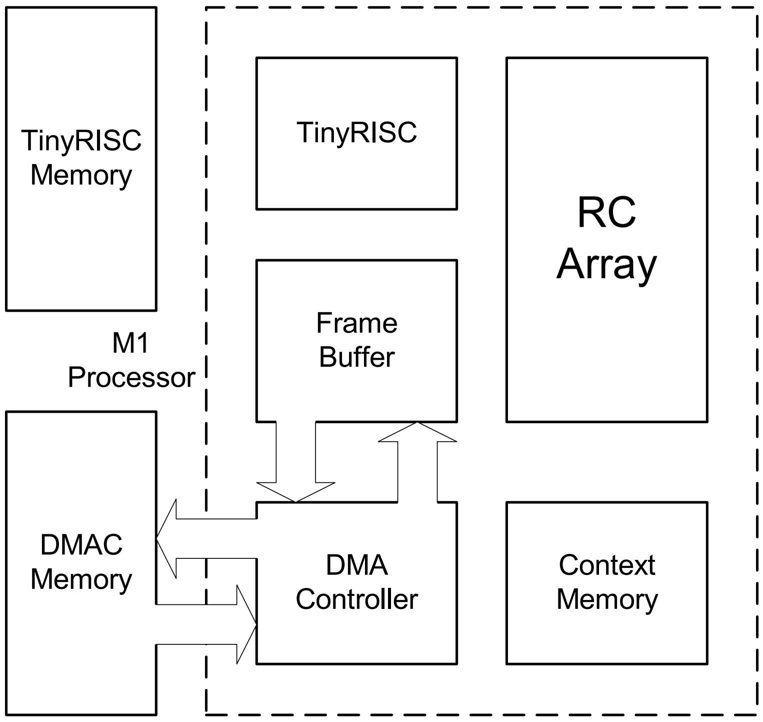


Figure 51. DMA-FB Program

The data transfer between the DMAC external memory and Context Memory through the DMAC for the DMA-CM-RC-TinyRISC program is shown in Figure 52. The RC array can get data from the context constant field. After some operations, the data will be written to the TinyRISC, then written to the TinyRISC external memory. Figure 53 shows the data transfer between the DMAC external memory and Frame Buffer as well as Context Memory through the DMAC for the DMA-FB-RC-TinyRISC program. The RC array gets data from the Frame Buffer. After some operations, the data will be written to TinyRISC, and then it will be written to the TinyRISC external memory.

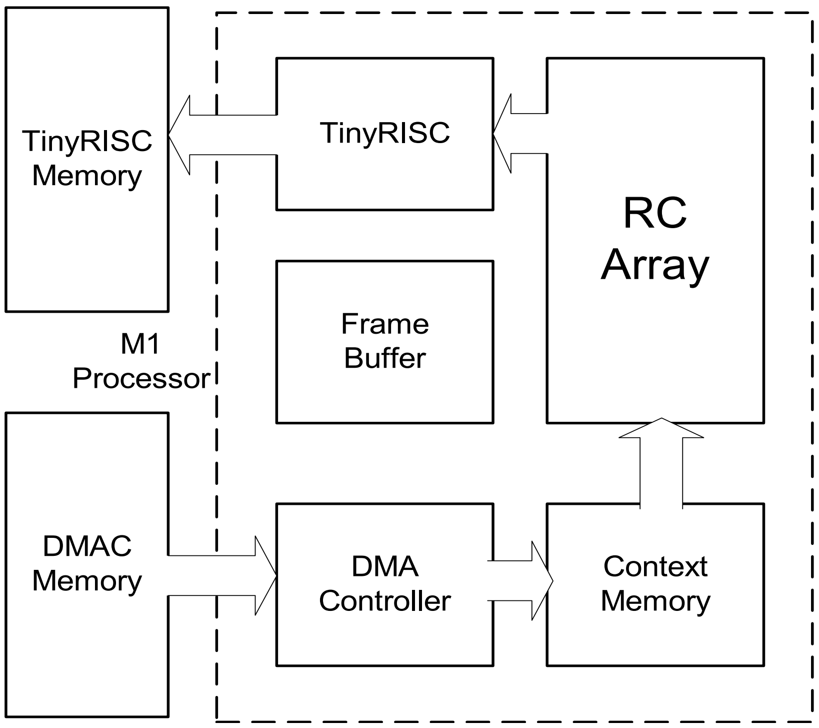


Figure 52. DMA-CM-RC-TinyRISC Program

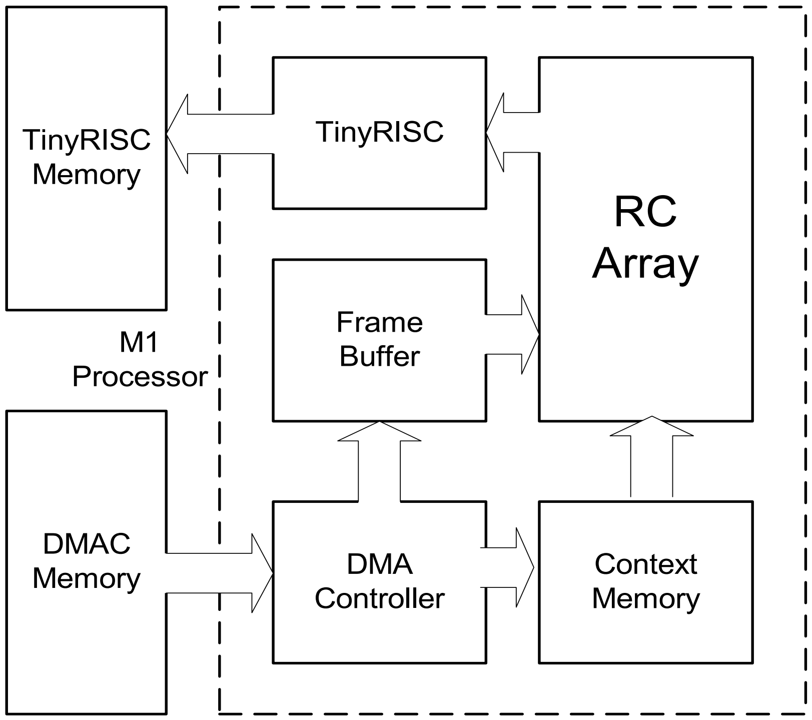


Figure 53. DMA-FB-RC-TinyRISC Program

Figure 54 shows the data transfer between the DMAC external memory and Frame Buffer as well as Context Memory through the DMAC for the DMA-FB-RC-FB-DMA program. The RC array gets data from the Frame Buffer. After some operations, the data will be written to the Frame Buffer, and then it will be written to the DMAC external memory.

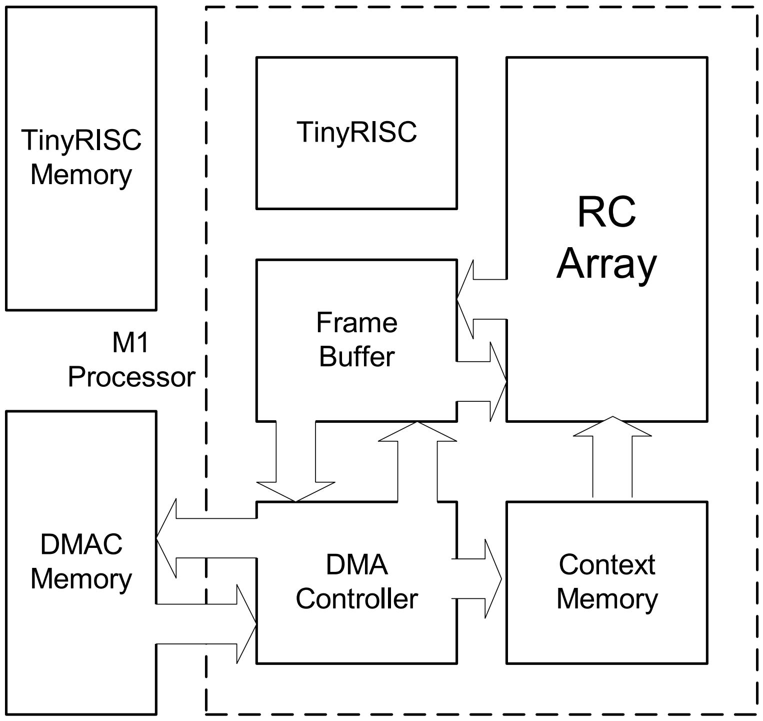


Figure 54. DMA-FB-RC-FB-DMA Program

Clock speed test*:* The testing conditions for clock speed test are the same as functionality test except that the testing frequency will be increased to the full speed that the M1 chip is supposed to run.

Power consumption test: Usually, this step covers the measurements of both static and dynamic power consumption. The static power consumption is measured when the system clock is permanently tied to logic 1 or logic 0. It is one of the very important and useful approaches, called I_{DDQ} testing. It is a technique for production quality and reliability improvement, design validation and failure analysis. I_{DDQ} testing can combine with other conventional approaches, such as functional or stuck-at fault (stuck-at 1 or stuck-at 0) testing, to improve the detection of defects. The dynamic power consumption is measured when the M1 chip runs at its maximum clock speed.

Working condition test: Essentially, the working condition test is to analyze the behavior of the M1 chip under different environments, such as fluctuation of the supply voltage and temperature.

3.2. ProTest Testing Machine

We use the ProTest equipment which is low-cost testing equipment, to test the functionality of the M1 chip. Other tests were performed with the IMS ATS. The ProTest environment uses the same test benches as those developed in the HDL simulation and verification phase since reuse of the simulation test bench is mandatory in order to guarantee rapid prototyping. Test patterns and simulation results captured during simulation are used to stimulate the real device under test (DUT). It can establish a rapid prototyping design and test environment. It basically uses the FPGA to generate the stimulus for the DUT based on the test vector. ProTest uses the RS232 serial port to communicate with a host PC or UNIX workstation. However, one of the disadvantages associated with the ProTest is that the only functionality test can be performed on it. Figure 55 shows the working behavior of the ProTest. For example, after one clock cycle, the ProTest captures the responses from the DUT; second, it shuts down the clock of the DUT; third, it sends the test results back to the host; fourth, it downloads the new test vector to configure the internal FPGA in order to provide the new stimulus; finally, it starts the new testing cycle. The period between two consecutive clock rising edges is several seconds. Since the strobe time is before the operations of uploading a result and downloading a new vector, they cannot be included in the test cycle of DUT. The maximum test frequency that ProTest can provide for DUT is 6.7 MHz. Another disadvantage associated with this observation is that ProTest is not good for some of the devices, which have dynamic circuits inside because of the discharge within the long period of data transfer between with the host. Figure 56 shows the M1 chip on the ProTest testing equipment.

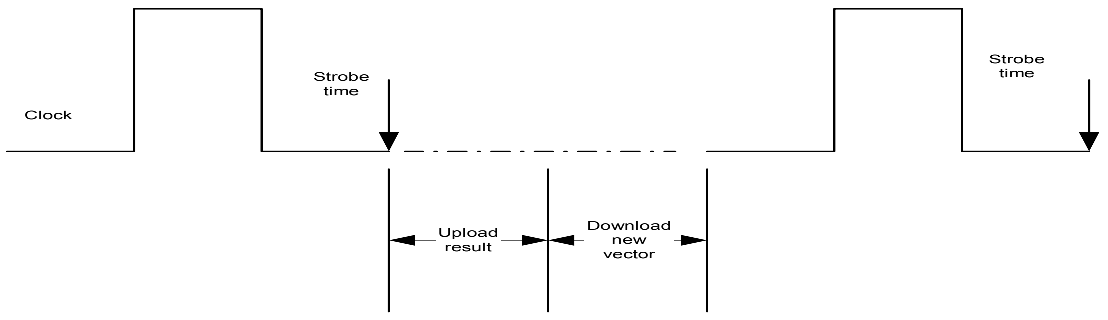


Figure 55. Clock Behavior of ProTest

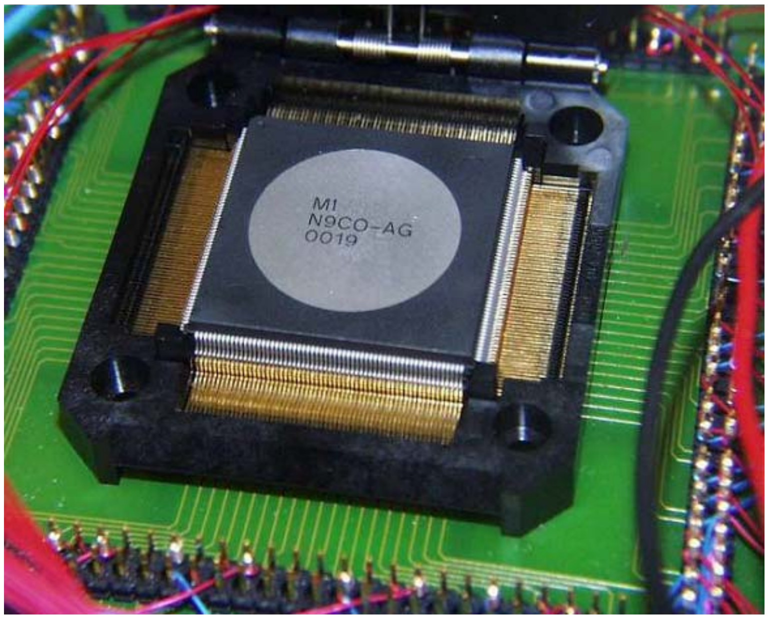


Figure 56. M1 chip on the ProTest equipment

3.3. IMS Testing Machine

IMS ATS is the commercial test equipment, which has more advanced features than ProTest. It has the capability to support all of the four different test schemes. IMS has the ability to regenerate the stimuli based on the data value of the test vectors and the signalling formats. As shown in Figure 57, the signaling formats, namely, nonreturn-to-zero (NRZ), delayed nonreturn-to-zero (DNRZ), return-to-zero (RZ), return-to-zero-inverted (RZI), RC (return-to-complement), R1, 2xRZ or 2xR1, together with the test vector values (0 or 1), will determine the waveform of the stimuli to the DUT. For example, if the signalling format is NRZ, then the stimulus to the DUT will be exactly the same as the test vector. If the signaling format is DNRZ, then the stimulus to the DUT will be a delayed version of the test vector. The amount of the delay can be specified in the environment settings of IMS translation. Properties RZ or RC can be used to generate a clock signal with the condition that the data value of test vector is equal to 1. In current IMS settings, all of the input signals except for the clock are with NRZ property, and the clock signal is with DNRZ property. Therefore, two test vectors are used to represent a real clock cycle. One test vector corresponds to the case that clock is high and the other corresponds to the case that clock is low. This can be improved by specifying the RC or RZ property for clock signal property. In this improved case, one test vector will correspond to one complete clock cycle including clock high and clock low.

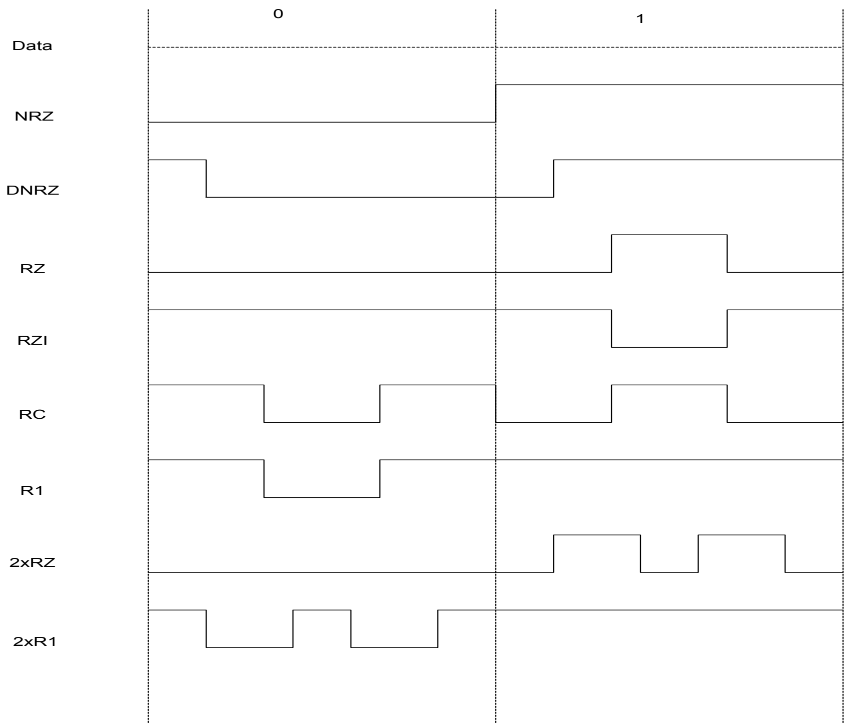


Figure 57. Sample Waveforms for Different Signaling Formats

The translation from the Lsim format to the IMS format (*.ims file) is done by using an IMS tool called Scenario. As shown in Figure 58 m1_ME_reduced_WB_step.tv (in Figure 58) represents one of the test vectors in Lsim format. The final information needed for this equipment is the definition of the correspondence between the DUT pins and signal channels in the IMS test board.

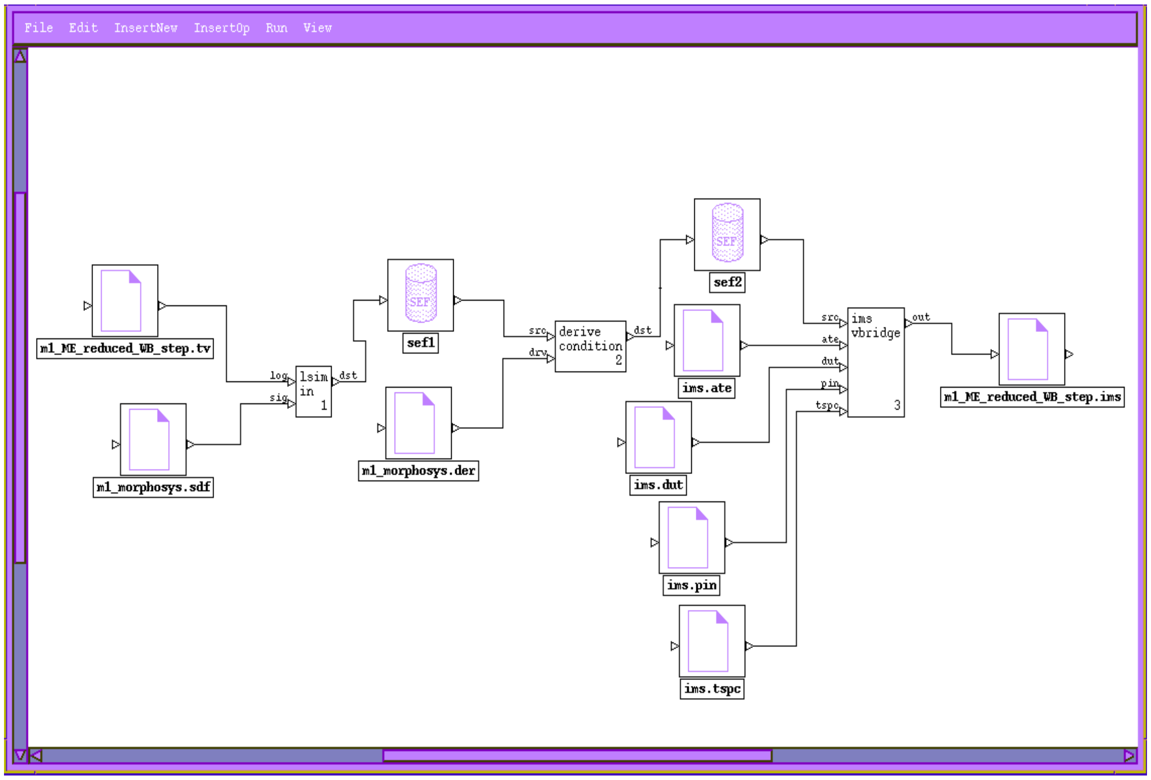
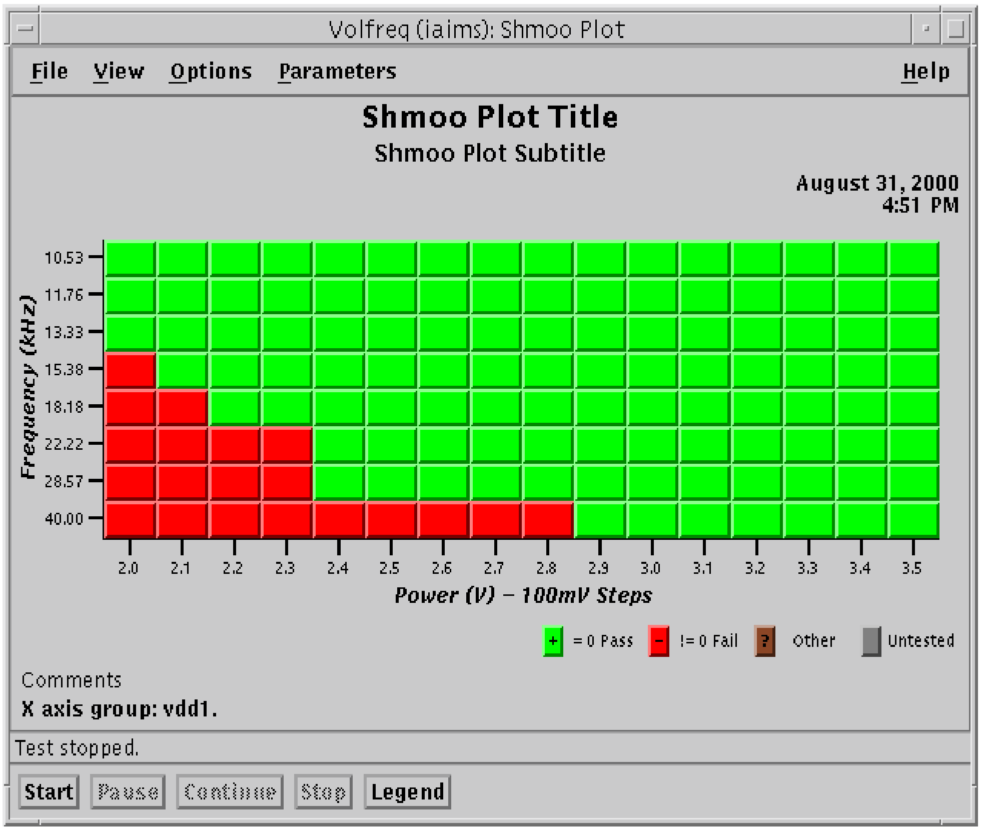


Figure 58. Scenario Translation Procedure

The IMS machine is able to change the power supply voltage together with the frequency during testing. This allows us to find out the working conditions of the M1 chip. This is called Shmoo, as shown in the Figure 59. The lighter color means that the DUT works well at that voltage and frequency. The darker color means the failure of the test.



Comments
X axis group: vdd1.

Test stopped.

Figure 59. Shmoo plot

3.4. Testing Results and Analysis

The maximum clock frequency was measured as 40 MHz for the M1 chip. The following section shows the testing results for the static power consumption, functionality tests, etc.

3.4.1. Static Current of MorphoSys M1 Chip

Table 11 shows the current measurements for power supply pads under different configurations where the supply voltage is 3.3 V. The temperature of the heat spread in the MorphoSys M1 chip will go up to 65˚C if Power is on and the clock is tied to either (VDD) or (GND) for a long time (such as more than 1 hour). From this table, we can see that there is a large static current going to the RC Array and Context Memory and there is a difference for this static current between clock high and clock low.

Table 11. Static Current for Different Power Supply Pads

Power Pads	When Clock=0 or floating	When clock = 1
PWR_PAD(RC Array & CM)	0.60 A	0.16 A
PWR_PAD(Pads self)	0.02 mA	0 mA
PWR_PAD(TinyRISC Execute)	0	0
PWR_PAD(RF)	11.25 mA	0
PWR_PAD(Data Cache)	0.04 mA	0.04 mA
PWR_PAD(Inserted Buffers)	0	0
PWR_PAD(TinyRISC Decode)	0	0
PWR_PAD(DMAC)	0	0
PWR_PAD(FB)	0.06 mA	0.03 mA

3.4.2. Dynamic Current

Table 12 shows the dynamic current for the M1 chip at 20 MHz and 50 MHz clock frequency.

Table 12. Dynamic Current of M1 Chip

20 MHz	0.387 A
50 MHz	0.562 A

3.4.3. Functionality Testing

Table 13 shows the functionality test results for different test programs. In most of the cases, the test results of the ProTest match the test results of the IMS machine. However, ProTest requires extra cooling during testing. This is because 1) the M1 chip sinks a decent amount of static current, as mentioned in Table 11 and 2) ProTest takes a long time to emulate the real testing environments.

Table 13. Functionality Test Result for Different Test Programs

Programs	Test results	Extra Cooling requirement
TinyRISC ALU	OK with/without cache	No for both
TinyRISC sort	OK with/without cache	No for both
DMA-FB	OK for both Bank A and B	Yes for ProTest No for IMS
DMA-CM-RC-TinyRISC	OK	Yes for ProTest No for IMS
DMA-FB-RC-TinyRISC	OK if NOP instructions are inserted between SBCB instructions.	Yes for ProTest No for IMS
FB-SHIFT	OK	Yes for ProTest No for IMS
DCT program	OK if NOP instruction are inserted between SBCB instructions	Yes for ProTest No for IMS
Motion Estimation	Works OK	Yes for ProTest No for IMS
ATR	Works OK	Yes for ProTest No for IMS

4. Programming Environment and Performance Analysis

This section introduces the software environment for programming MorphoSys M1 and describes the various applications mapped to M1 from different target domains such as video compression and image processing. Performance analysis will also be provided for these applications along with the comparisons against other contemporary systems.

4.1. Programming Environment

Figure 60 depicts the complete programming environment for MorphoSys M1. The MorphoSys software tool set consists of *mView, mLoad, mSched, mcc, MuLate*(2), and *MorphoSim*. *mView, mLoad, mSched,* and *mcc* are used for code generation and application scheduling while *MuLate* and *MorphoSim* model the MorphoSys architecture in C++ and VHDL, respectively, and are used to simulate the execution of application codes on MorphoSys M1. The steps involved in the application program development are described below.

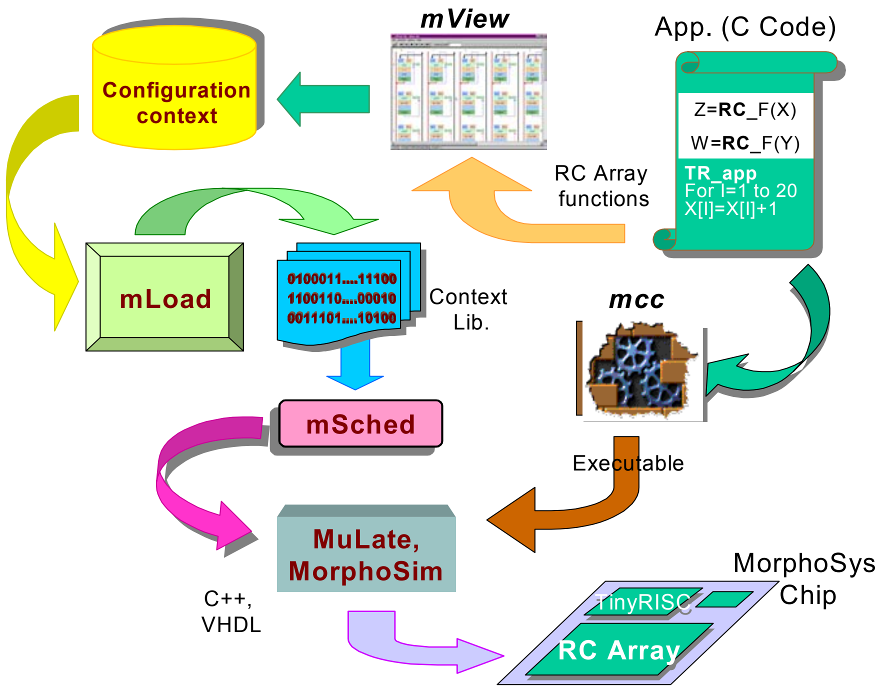


Figure 60. Programming Environment for MorphoSys M1

1. The application code in high-level language, for example C++, is preprocessed to identify the functions to be executed on TinyRISC and the functions to be mapped to RC Array. This preprocessing is currently done manually by the users.
2. The designer uses the *mView* to generate the optimized RC assembly context program for the functions that are mapped to the RC Array. *mView* (Figure 61) is a graphical user interface developed to program the RC Array. Its major advantage is that it hides the details of the context assembly syntax from the users. Users can therefore program the RC Array without any knowledge of the assembly syntax. In addition to generating the context assembly codes, m*View* can also be used to simulate the RC Array execution for a given configuration program.

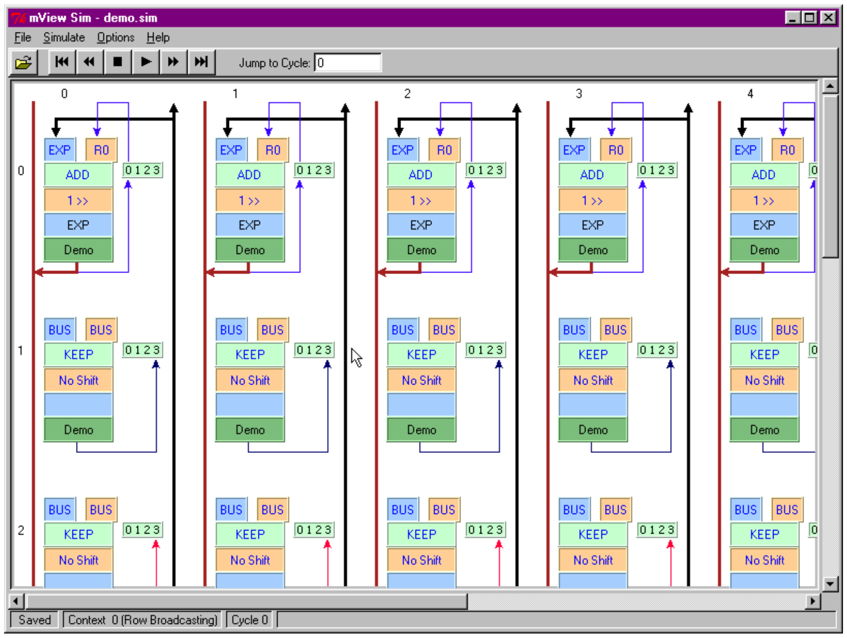

mView Sim - demo.sim
File Simulate Options Help
Jump to Cycle: 0
EXP R0 ADD 1 >> EXP Demo
BUS BUS KEEP No Shift Demo
0 1 2 3
Saved
Context 0 (Row Broadcasting)
Cycle 0

Figure 61. Graphical Programming Tool *mView*

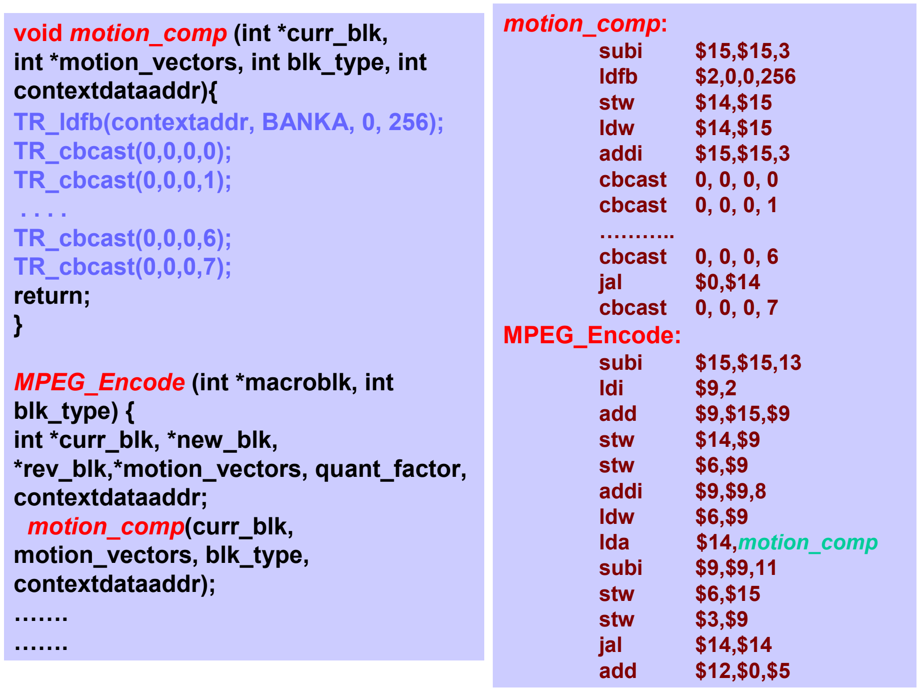
set	p,	q	function	A	B	LSL	n	> R	WE ;
set	0,	0	BYPASS	R0	I			> 3	;
set	0,	1	ADD	VE	R0	LSL	2	> 0	WE ;
set	0,	2	SUB	T	R1	LSL	3	> 2	WE ;
set	0,	3	ADDSUBF	R0	R2	LSR	3	> 0	;
set	0,	4	KEEP	def	def			> 3	;
set	0,	5	CLOAD!0x004	I	I			> 2	;
set	0,	6	CMUL!0x1D9	B	def			> 1	;
set	0,	7	CMUL!0x0C3	R0	R1			> 1	;

Figure 62. RC Context Assembly Code Example

3. Figure 62 illustrates an example of the RC context assembly. The syntax is described as follows. Each line starts with "set" and ends with ";". The first field, "p," specifies the row or the column number of the RCs to be programmed. This corresponds to one of the eight context sets in either the row context block or the

column context block The column context sets are numbered from 0 to 7, whereas the row context sets are numbered from 8 to 15. The second field, “q,” represents the location of the context word within a context set. There are 16 locations in each set, therefore “q” is ranging from 0 to 15. The “function” field specifies the RC function. The mnemonics in this field corresponds to one of the values for the RC functions listed in Table 15. For the RC functions with a constant, the constant data is concatenated to the mnemonic. The fields, “A” and “B” specify the inputs to be selected for INA_MUX and INB_MUX in the RC, respectively. The possible values for each field correspond to all of the inputs of the two multiplexers as depicted in Figure 20. The default values, specified as “def” for these two fields are the “I” input. The next field specifies the shift direction and the shift amount to be applied to the output of each RC. “LSL n” signifies the output of the RC to be shifted left by n bits, and “LSR n” implies a right shift of n bits. The default behavior (when this field is not specified) is no shifting. The field, “> R,” specifies the register in which the output of the RC is to be stored. This field is preceded by a “>” symbol and followed by the register number “R.” The value of this field is in the range of 0 to 3. The last field is optional. If “WE” is specified, the data from the RC output registers of a particular row/column will be written to the column/row express lanes in the next clock cycle. After the context assembly is generated, *mLoad* is used to translate the context assembly to binary context words.

4. The functions mapped to the RC Array are represented by in-line functions in the high-level language code. Figure 63 shows an example of motion compensation function. The combined codes are then compiled by the *mcc* complier into the TinyRISC assembly instructions.



```
void motion_comp (int *curr_blk,
int *motion_vectors, int blk_type, int
contextdataaddr){
TR_ldfb(contextaddr, BANKA, 0, 256);
TR_cbcast(0,0,0,0);
TR_cbcast(0,0,0,1);
 . . . .
TR_cbcast(0,0,0,6);
TR_cbcast(0,0,0,7);
return;
}

MPEG_Encode (int *macroblk, int
blk_type) {
int *curr_blk, *new_blk,
*rev_blk,*motion_vectors, quant_factor,
contextdataaddr;
  motion_comp(curr_blk,
motion_vectors, blk_type,
contextdataaddr);
…….
…….
```

```
motion_comp:
        subi    $15,$15,3
        ldfb    $2,0,0,256
        stw     $14,$15
        ldw     $14,$15
        addi    $15,$15,3
        cbcast  0, 0, 0, 0
        cbcast  0, 0, 0, 1
        ……….
        cbcast  0, 0, 0, 6
        jal     $0,$14
        cbcast  0, 0, 0, 7
MPEG_Encode:
        subi    $15,$15,13
        ldi     $9,2
        add     $9,$15,$9
        stw     $14,$9
        stw     $6,$9
        addi    $9,$9,8
        ldw     $6,$9
        lda     $14,motion_comp
        subi    $9,$9,11
        stw     $6,$15
        stw     $3,$9
        jal     $14,$14
        add     $12,$0,$5
```

Figure 63. TinyRISC Assembly Code Generation for Motion Compensation

5. The application assembly code for the TinyRISC and the corresponding context assembly for the RC Array are then used as the executable program for *MuLate* and *MorphoSim* simulator along with the appropriate input data such as image frames (see Figure 64).

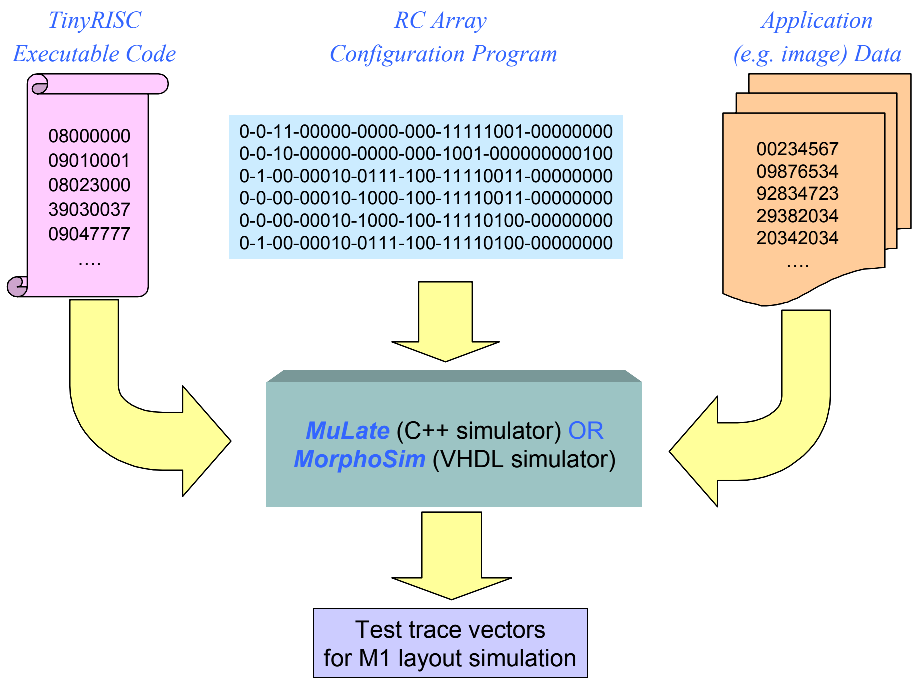


Figure 64. Inputs for *MuLate* and *MorphoSim* Simulations

6. For complex applications, *mSched* can be used to obtain an optimized schedule for the data and context movements required for the application. A typical complex application is composed of a sequence of subtasks or kernels that are executed repeatedly as a loop. A kernel is a well-defined task that can be independently executed after the previous tasks in the execution flow graph. The algorithm employed by *mSched* uses three major criteria, minimizing context reloading, maximizing data reuse, and maximizing overlap of computation and data movements. The algorithm finds the actual schedule to obtain the total execution time for a given application. It follows a two-pronged approach of partitioning the execution data flow graph and then scheduling each partition. A backtracking technique is used to support the search process in both tasks. This process is guided by a heuristic method that tries to explore the best candidate solution first. Bounding heuristics are employed in order to prune the solution space early. The resulting schedule minimizes the total execution time.

4.2. Application Mapping and Performance Analysis

This section describes the application mapping process for MorphoSys M1. The performance analysis will also be discussed through the mapping of video compression and ATR on MorphoSys. Video compression has a high degree of data parallelism and tight real-time constraints. ATR is one of the most computation-intensive applications with bit-level operations. We also provide performance estimates based on VHDL simulations. Notice that we mapped ATR manually, in this reseach.

The basic steps for mapping an application to MorphoSys M1 are summarized as follows:
1. The user identifies the data-parallel paths in a given application.
2. The computations for one of the data-parallel paths of the application are mapped to a group of RCs, preferably a complete row or column. This mapping utilizes the RC functions and the RC Array interconnection network.
3. After a mapping is established for one parallel path, it is replicated over the rest of the RC Array to represent the other parallel paths of the application.

It should be noted that the context broadcasting modes enable the parallel (SIMD) operations in the eight rows or columns of the RC Array. Also, eight sets of one-operand or two-operand data may be loaded simultaneously to one or more columns of the RC Array. Typically, this loading of data is done for one column at a time. When more than one column needs to access the data set, it is possible to load this data set concurrently to multiple columns. Therefore, a set of 8 (or 16) operands may be broadcast to the entire RC Array. In this case, the RCs in the same row receive the same data.

For mapping a specific application kernel, it is desirable to follow a data-centric computation approach. This implies that the data should be processed for as long as possible within an RC Array. The data-centric approach reduces the amount of data movements between the Frame Buffer and the RC Array and enhances throughput. For example, consider an application in which an image frame has to be processed through multiple tasks. It is more efficient to perform all of the tasks together on a subblock of the frame in the RC Array than executing one task at a time on the entire image frame and reloading data for the next task. In other words, the maximum number of computations should be executed on a set of data resident on the RC Array by utilizing the multiple context planes present in the MorphoSys before storing back the data to the Frame buffer and loading new data.

4.2.1. Motion Estimation

Motion estimation is widely adopted in video compression to identify the redundancy between frames. Motion Estimation is the most computation-intensive portion of video compression algorithms. The most popular technique for motion estimation is the block-matching algorithm because of its simple hardware implementation. The block-matching algorithm is also recommended by several standard committees (e.g., MPEG and H.261 standards). Among the different searching methods, full search block matching (FSBM) involves the maximum computations. FSBM, however, gives an optimal solution with low control overhead and is found to be an ideal candidate for VLSI implementation.

Typically, the mean absolute difference (MAD) criterion is the one implemented in VLSI due to its relative accuracy and compactness of hardware implementation. With the MAD criterion, the FSBM algorithm can be formulated as follows:

$MAD(m, n) = \Sigma_{1 \le i \le N} \Sigma_{1 \le j \le N} \mid R(i, j) - S(i+m, j+n) \mid$ given $-p \le m, n \le q$, (1)

Where p and q are the maximum displacements, $R(i, j)$ is the reference block of size N by N pixels at coordinates (i, j), and $S(i+m, j+n)$ is the candidate block within a search area of size $(N + p + q)^2$ pixels in the previous frame. The displacement vector is represented by (m, n), and the motion vector is determined by the least $MAD(m, n)$ among all of the $(p + q + 1)^2$ possible displacements within the search area.

Figure 65 shows the configuration of the RC Array for FSBM computation. The operations of the RC Array are denoted row by row. Initially, one reference block and the search area associated with it are loaded into one set of the Frame Buffer. The RC Array starts the matching process for the reference block resident in the Frame Buffer. During this computation, another reference block and its associated search area are loaded into the other set of the frame buffer so that the loading and computation time are overlapped.

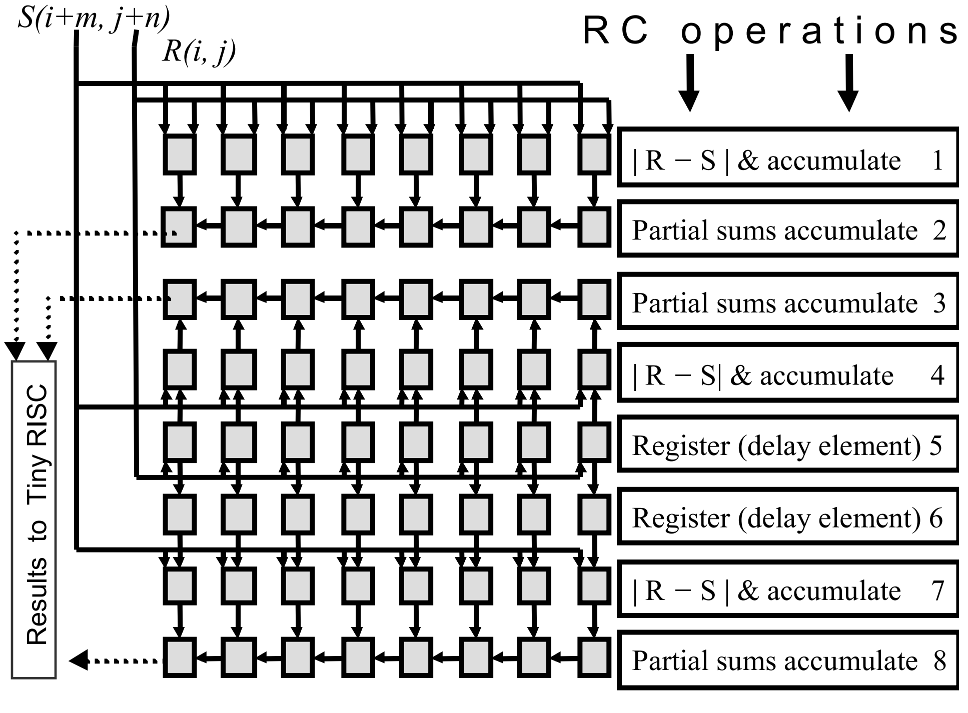

S(i+m, j+n)
R(i, j)
RC operations
| R − S | & accumulate 1
Partial sums accumulate 2
Partial sums accumulate 3
| R − S| & accumulate 4
Register (delay element) 5
Register (delay element) 6
| R − S | & accumulate 7
Partial sums accumulate 8
Results to Tiny RISC

Figure 65. Configuration of RC Array for FSBM

For each reference block, three consecutive candidate blocks are matched concurrently in the RC Array. As shown in Figure 65, each RC in the first, fourth, and seventh row performs the following computation:

$$P_j = \Sigma_{1 \le j \le N} \mid R(i, j) - S(i+m, j+n) \mid \qquad (2)$$

The data of the reference block is sent to the first row of RCs and passed to the fourth row and seventh row through the delay elements configured in the fifth and sixth rows of RCs. The eight partial sums (P_j) generated by the first, fourth, and seventh rows are then passed to the second, third, and eighth rows respectively to perform the following computation:

$$MAD(m, n) = \Sigma_{1 \leq j \leq N} P_j \qquad (3)$$

Subsequently, three *MAD* values corresponding to the three candidate blocks are sent to the TinyRISC for comparison, and the RC Array starts the block matching for the next three candidate blocks.

Based on the computation model presented above and using $N = 16$, implying a reference block size of 16 by 16 pixels, the RC Array accomplishes the matching of 3 candidate blocks every 36 clock cycles. For $p = 8$ and $q = 8$, there are 289 candidate blocks (102 iterations) in each search area, and it takes a total of [102x(36+10)] = 4692 cycles for the RC Array to finish the matching of the entire search area. The 10 extra cycles are required for TinyRISC to compare the three *MAD* values and update the motion vector for best match. For an image frame size of 352 by 288 pixels at 30 frames per second (MPEG-2 main profile, low level), the number of reference blocks per frame is 22 by 18 = 396. The total processing time of an image frame is 4692 by 396 = 1,858,032 cycles. The clock rate of the MorphoSys M1 is 100 MHz, so the computation time is about 18.5 ms. This is much smaller than the frame period of 1/30 second. The context loading time is only 73 cycles, which is negligible compared to the computation time.

In Table 14 MorphoSys M1 performance is compared with two ASIC architectures, and the Intel Pentium MMX processor for matching one 8 by 8 reference block against its search area of 8-pixel displacement. The two ASIC designs employ customized hardware units such as parallel adders to enhance performance. Intel Pentium MMX is a general-purpose processor that uses the enhanced instruction set to explore the data parallelism for multimedia applications. The number of processing cycles for MorphoSys is comparable to the cycles required by the ASIC designs. Pentium MMX takes about 29,000 cycles, which is almost 30 times more than MorphoSys. The processing unit that constitutes the critical path in the two ASIC implementations is the absolute difference and accumulation unit. We implemented the absolute difference and accumulation unit using 0.35-μm CMOS technology and performed Hspice simulation on this circuit. Based on our simulation, the two ASIC systems can operate at about 200 MHz in 0.35-μm technology. We used a 233 MHz for the Pentium MMX implementation, which is the highest clock rate for a 0.35-μm Pentium processor. Taking into account the clock rate, we depict the performance comparison in Table 14. The result shows that MorphoSys can deliver an order of magnitude performance speedup over general-purpose processors.

Table 14. Motion Estimation Performance Analysis

	MorphoSys	ASIC 1	ASIC 2	Pentium MMX
# of Clock cycles	1020	581	1159	29000
Processing time	10.2 µs	2.9 µs	5.8 µs	145 µs

4.2.2. Discrete Cosine Transform (DCT)

The forward and inverse DCTs are frequently used in MPEG encoders and decoders for transformation of image pixels to the frequency domain and back to the spatial domain. The DCT and IDCT are applied in a 2-dimensional manner over an image pixel block (typically an eight-by-eight block). Each 2-D DCT or 2-D IDCT may be computed in terms of the 1-D DCT and 1-D IDCT, respectively, using the separability property. The 2-D DCT algorithm may be realized by the computation of eight 1-D DCTs along the pixel rows followed by the computation of eight 1-D DCTs along the image columns.

In the following mapping and performance analysis, a fast algorithm for an eight-point 1-D DCT is considered. This algorithm involves 16 multiplications and 26 additions; therefore, 256 multiplications and 418 additions are required for the 2-D DCT implementation.

Mapping to the RC Array

The standard block size for DCT in most image and video compression standards is eight-by-eight. Since the RC array has the same size, each pixel of the image block may be directly mapped to each RC. Each pixel of the input block is stored in one RC. The 2-D DCT algorithm is implemented in the following sequence of steps:

1. The eight-by-eight pixel block is loaded from the frame buffer to the RC Array. The data bus between the frame buffer and the RC array allows concurrent loading of eight pixels at a time. The entire block is loaded in eight cycles.
2. The DCT coefficients needed for computation are provided through the constant field of the context words.
3. Using the separability property, eight 1-D DCTs are first computed in parallel along the rows using the column context broadcast mode, as illustrated in the left half of Figure 66.
4. Eight 1-D DCTs are then computed concurrently across the columns using the row context broadcast mode. This operation is illustrated is the right half of Figure 66.
5. This completes the 2-D DCT transformation. The eight-by-eight result block is then written out to the Frame Buffer (in eight cycles) and another image block is loaded.

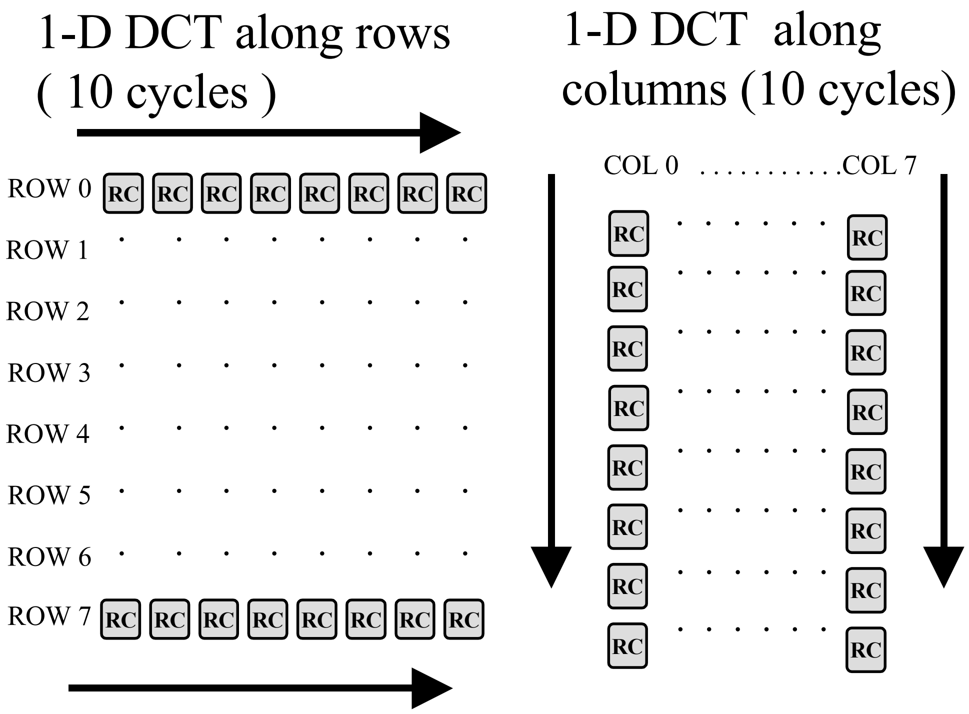

1-D DCT along rows (10 cycles)
1-D DCT along columns (10 cycles)
ROW 0
ROW 1
ROW 2
ROW 3
ROW 4
ROW 5
ROW 6
ROW 7
COL 0 ....COL 7
RC

Figure 66. Computation of 2-D DCT Across Rows/Columns

Each sequence of a 1-D DCT involves:
1. *Butterfly computation:* It takes three cycles to perform this using the interquad connectivity layer of express lanes.
2. *Computation and rearrangement*: For a 1-D DCT (row/column), the computation takes six cycles. An extra cycle is used for rearrangement of computed results.

The 2-D IDCT may be computed in a similar way in the RC Array.

Computation Cost

The cost for computing a 2-D DCT on an 8 by 8 block of the image is as follows: 6 cycles for butterfly, 12 cycles for both 1-D DCT computations, and 3 cycles are used for rearrangement and scaling of data (giving a total of 21 cycles). This estimate is verified by VHDL simulation. Assuming the data blocks to be present in the RC Array (through overlapping of data load/store with computation cycles), it would take 0.49 ms for MorphoSys at 100 MHz to compute the DCT for all 8 by 8 blocks (396 by 6) in one frame of a 352 by 288 image. The cost of computing the 2-D IDCT is the same, because the steps involved are similar. Context loading time is quite significant at 270 cycles. However, this effect is minimized through transforming a large number of blocks (typically 2376 blocks) before a different configuration is loaded.

Performance Analysis

MorphoSys requires 21 cycles to complete a 2-D DCT (or IDCT) on an 8 by 8 block of pixel data. This is in contrast to 240 cycles required by Pentium MMX. Even a dedicated superscalar multimedia processor requires 201 clocks for the IDCT. REMARC [4] takes 54 cycles to implement the IDCT, even though it uses 64 nanoprocessors. For the comparison of processing time, we use the clock rate of 200 MHz V830R/AV implemented using 0.25-μm technology. REMARC has similar processing power (in terms of processing elements) to MorphoSys, so we assume 100 MHz for REMARC. The comparison is summarized in Table 15.

Notably, MorphoSys performance scales linearly with the array size. For a 256-element RC array, the number of operations possible per second would increase fourfold, with corresponding effect on throughput for 2-D DCT and other algorithms. The performance figures (in GOPS) are summed up in Figure 67, and these are more than 50 percent of the peak values. Once, again the figures are scaled for future generations of MorphoSys M1, conservatively assuming a constant clock of 100 MHz.

Table 15. Performance Comparison for DCT/IDCT

	MorphoSys	**REMARC**	**V830R/AV**	**.Pentium MMX**
Clock cycles	21	54	201	240
Processing time	210 ns	540 ns	1005 ns	1200 ns

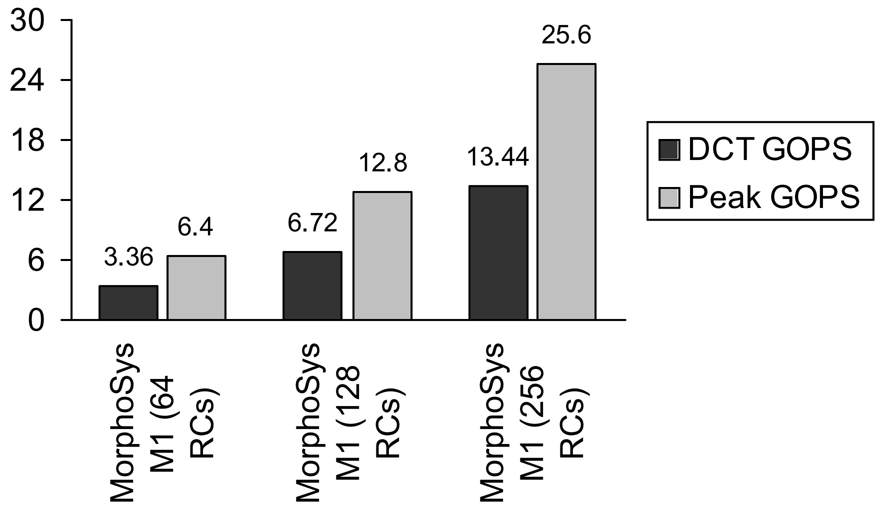


Figure 67. Performance Scaling for DCT/IDCT (GOPS)

Some other points are worth noting: first, all rows (columns) perform the same computations; hence, they can be configured by a common context (thus enabling broadcast of context word), which leads to saving in context memory space. Second, the RC array provides the option of broadcasting context either across rows or across columns. This allows computation of second 1-D DCT without transposing the data. Elimination of the transpose operation saves a considerable amount of cycles, and is important for high performance. This operation generally consumes valuable cycle time. For example, even a hand-optimized version of IDCT code for Pentium MMX (that uses 64-bit registers) needs at least 25 register-memory instructions for completing the transpose. Processors, such as the TMS320 series, also expend valuable cycle time on transposing data.

Precision analysis

We conducted experiments for the precision of the output IDCT for MorphoSys as specified in the IEEE Standard. Considering that MorphoSys is not a custom design, and performs fixed-point operations, the results were impressive. We satisfied worst-case pixel error. The overall mean square error (OMSE) was within 15 percent of the reference value. The majority of pixel locations also satisfied the worst-case reference values for mean error and mean square error.

Quantization, Inverse Quantization and Zigzag Scan

The quantization step follows the transformation of pixel data using DCT, and the inverse quantization step is used prior to IDCT while regenerating the image frame. Both types of quantization require some conditional operations that are implemented using *ADDSUBF* RC instruction. The computation time is five cycles without considering the data input and output time. This I/O time may be ignored since quantization is performed on blocks that are already present in the RC Array. Otherwise, this task requires 21 cycles.

Zigzag and Variable Length Coding (VLC)

The zigzag scan function has a very irregular nature and is currently implemented through transferring the image data to the TinyRISC in 64 cycles. The data will be entropy coded in the TinyRISC using VLC. Currently, VLC is done on the TinyRISC.

4.2.3. MPEG-2 Encoder Performance

Video compression is an integral part of many multimedia applications. In this context, MPEG standards for video compression are important for realization of digital video services, such as video conferencing, video-on-demand, HDTV and digital TV. MPEG standards specify the syntax of the coded bit stream and the decoding process. Based on this, Figure 68 shows the block diagram of an MPEG encoder.

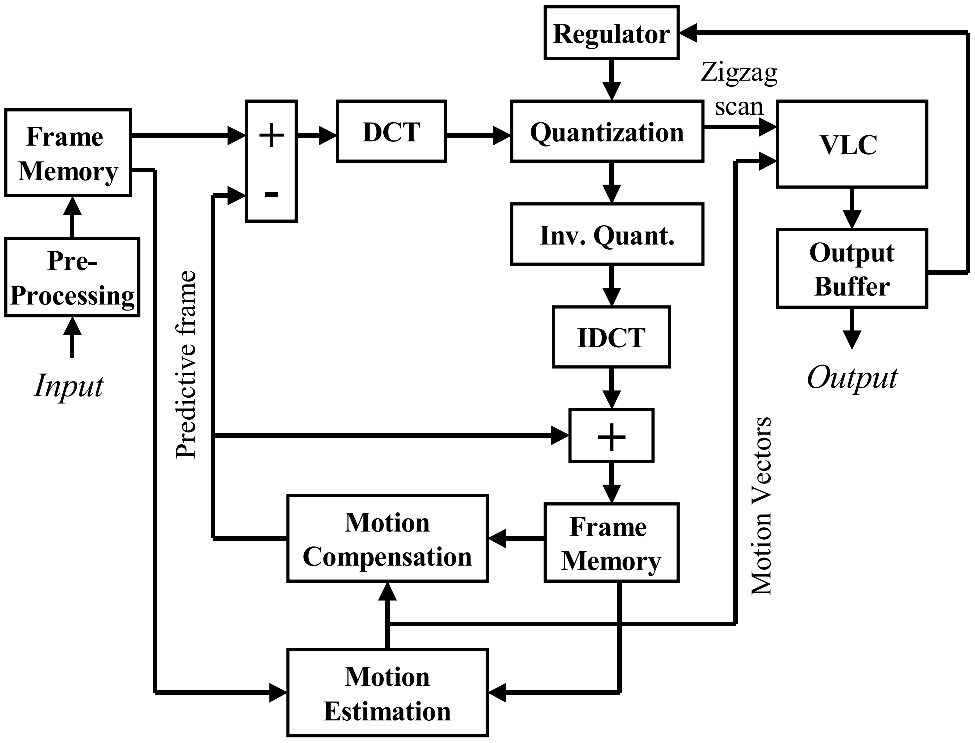


Figure 68. Block Diagram of an MPEG Encoder

As depicted in Figure 68, the functions required of a typical MPEG encoder are as follows:

- Preprocessing: for example, color conversion to YCbCr, prefiltering and subsampling.

- Motion Estimation and Compensation: After preprocessing, motion estimation is used to remove temporal redundancies in successive frames (predictive coding) of P and B type.
- DCT and Quantization: Each macroblock (typically consisting of six blocks of size 8 by 8 pixels) is then transformed using the DCT. The resulting DCT coefficients are quantized to enable compression.
- Zigzag scan and VLC: The quantized coefficients, are rearranged in a zigzag manner (in order of low-to-high spatial frequency) and compressed using VLC.
- Inverse Quantization and Inverse DCT: The quantized blocks of I and P type frames are inverse quantized and transformed back into the spatial domain by an inverse DCT. This operation yields a copy of the picture that is used for future predictive coding, i.e., motion estimation.

We mapped all of the functions for MPEG-2 video encoder, except the VLC, to MorphoSys. We assume that the Main profile at the low level is being used. The maximum resolution required for this level is 352 by 288 pixels per frame at 30 frames per second. We further assume that a group of pictures consists of a sequence of four frames in the order IBBP (I = Intracoded picture, P = Forward predicted picture, and B = Bidirectionally predicted picture; IBBP sequence is a typical choice for broadcasting applications). The number of cycles required to compute each subtask of the MPEG encoder for each macroblock type are listed in Table 16. Besides the actual computation cycles, we also take into account the configuration load cycles and the cycles for loading the data from memory.

All the macroblocks in each P and B frame are first subjected to motion estimation, and then we perform motion compensation, DCT, and quantization for all macroblocks of a frame. These are written out to frame storage in main memory. Finally, we perform inverse quantization, inverse DCT, and reverse motion prediction for each macroblock of I and P type frames. Each frame has 396 macroblocks, and the total number of cycles required for encoding each frame type is shown in Figure 69. It may be noted that motion estimation takes up almost 90 percent of the computation time for P and B type frames.

Table 16. MorphoSys Performance for I, P and B Macroblocks

	Motion Estimation			Motion Compensation, DCT and Quantization		
	Context load	Memory load	Compute	Context load	Memory load	Compute
I type	0	0	0	258/258	102/102	624/624
P type	73	322	4692	258/258	204/204	672/672
B type	73	597	9384	258/ NA	306 / NA	720/ NA

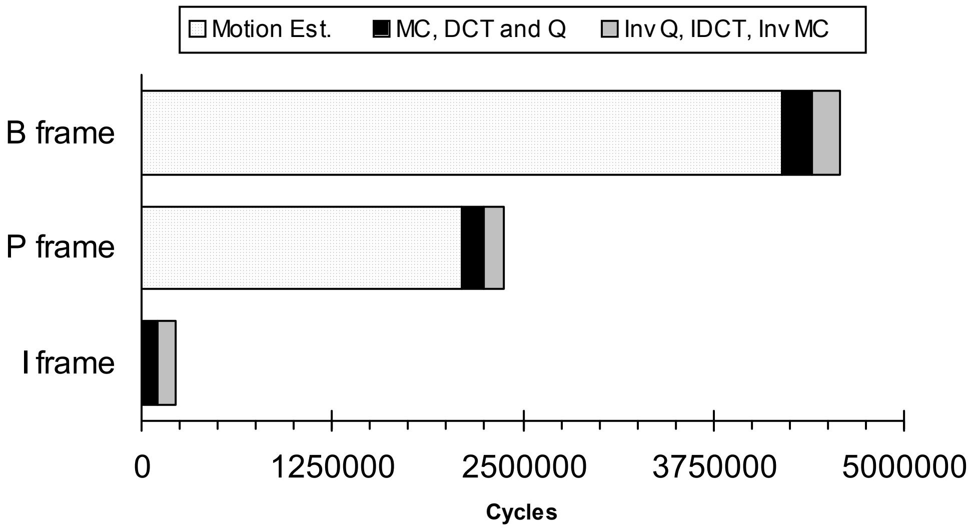


Figure 69. MorphoSys Performance for I, P and B Frames

From the data in Figure 69, and using the assumption of frame sequence of IBBP, the total encoding time is 108.4 ms. This is 81 percent of the total available time (133.3 ms). From empirical data values in [7], the remaining 19 percent of available time is sufficient to compute VLC. For the frame sequence of IBBPBBP, the total encoding time is 207.5 ms, which is 90 percent of the available time (233 ms). We compare the MPEG video encoder performance with that of REMARC [4] in Table 17. Even though MorphoSys figures do not include VLC, they are almost two orders of magnitude less than REMARC. The motion estimation algorithm (the major computation) is the same for REMARC and MorphoSys (FSBM).

Table 17. Comparison of MorphoSys with REMARC for MPEG Encoder

	MorphoSys M1 (64 RCs)	REMARC (64 nanoprocessors)
I frame	0.5×10^6	52.9×10^6
P frame	2.3×10^6	69.6×10^6
B frame	4.0×10^6	81.5×10^6

4.2.4. ATR

ATR is the machine function of detecting, classifying, recognizing, and identifying an object without the need of human intervention. The ACS Surveillance challenge has been quantified as the ability to search 40,000 square nautical miles per day with 1 meter resolution [8]. The computation levels for this problem when the targets are partially obscured reaches the hundreds-of-teraops range. Currently, there are many algorithmic choices available to implement an ATR system.

An ATR processing model has been developed at Sandia National Laboratory to detect partially obscured targets in synthetic aperture radar (SAR) images generated by the radar imager in real time [9]. There are three major steps for this algorithm- focus of attention (FOA), second level of detection (SLD), and final identification (FI).

SAR images that are approximately 600 to 1000 pixels (8-bit pixels) on a side are input to a FOA processor to create 2 times down-sampled subimages of the original SAR data, where each subimage contains a single target centered within the subimage. These subimages are referred to as chips and are 128 by 128 pixels. The SLD step processes the chips generated by the FOA step. The operations involve binary template correlation and thresholding. SLD also uses adaptive threshold levels determined by the overall image intensity. The FI step is similar to SLD step but is carried out on full resolution image data and templates with finer angular resolution. The output of FI is a location of the target and the confidence level corresponding to the level of correlation between the target and the FI images.

The algorithm discussed in this section is a modification of SLD called Chunky SLD. Chunky SLD adds a level of complexity to SLD by using more templates to represent objects that have been partially obscured. This allows better target recognition at the cost of higher computational requirements. The processing model developed at Sandia National Laboratory is shown in Figure 70 [9, 10]. Each target is represented as a set of 40 template pairs where each pair of templates is a digital representation of some salient feature of the specific target. Target templates appear in pairs, one of which is a *bright template* and the other is a *surround template*. The bright template identifies locations where a strong radar return is expected, while the surround template identifies locations where strong radar absorption is expected. Each pair of bright and surround templates is referred to as a *chunk*. Each set of 40 chunks represents a single target at a specific rotation. There are 72 orientations where each orientation represents a different target orientation and radar incidence angle. Each set of 72 orientations is referred to as a *class* and is the complete set of templates that must be correlated with a chip to detect the presence of a specific target.

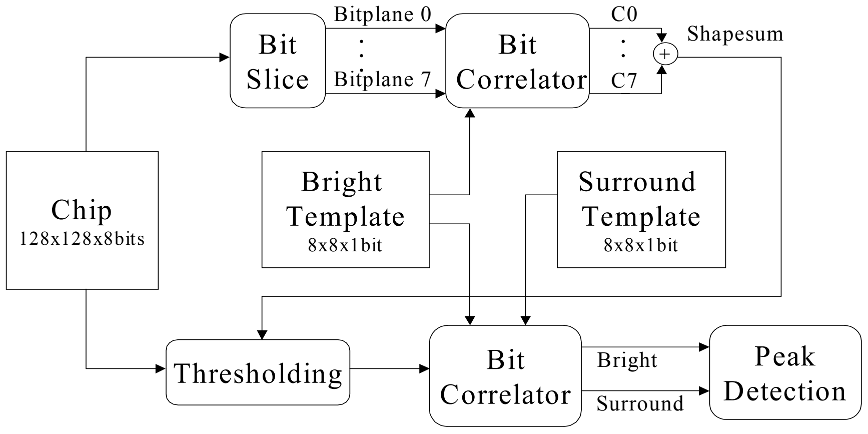


Figure 70. ATR Processing Model

The sequence of steps involved are as follows: The first step is to generate the shapesum. The 128-by-128-pixel chip is sliced into eight bit-planes; and the shapesum is generated by correlating each bit-plane with the bright template and then computing a weighted sum of the eight results. The second step is correlating the actual target templates with the chip. The correlation is performed on eight different binary images that are generated by applying eight predefined threshold values to the chip. The binary images are correlated with both the bright and surround template to generate eight pairs of brightsum and surroundsum, and the shapesum is used to select one of the eight pairs of results. The selected pair of results is subsequently forwarded to the peak detector. The selected brightsum and surroundsum are compared against the predefined maximum and minimum values to generate a hitcount. A hit is obtained if the threshold value, brightsum, and surroundsum meet certain criteria. The hits from all of the templates in a class are counted to calculate a hitcount that is used is the next stage of ATR.

The Chunky SLD algorithm has been successfully mapped to MorphoSys M1. To illustrate the mapping, a 64-by-64-pixel chip is assumed where each pixel is 8 bits. The bright and surround templates are binary and have 8 by 8 values. It is also assumed that the 8 bit-planes of the chip are provided to the system through preprocessing. Each bit-plane is loaded to the Frame Buffer and a 64 by 8 (columns x rows) block is loaded into the RC Array. This image block is correlated with the 8 by 8 bright template that is loaded into RC Array as constant values in the context words. The shapesum for each bit-plane is generated using the bit-correlation process. The shapesums for the eight bit-planes are accumulated in a weighted manner. Since there are 64 offsets within a 64 by 8 image block, 64 shapesums are generated and stored in the Frame Buffer as an 8 by 8 matrix with 8-bit entries.

Next, the bright template and surround template are correlated with the eight binary images generated by applying the eight predefined threshold values to the chip. It is also assumed that these binary images are generated by preprocessing. Each binary image is loaded into the Frame Buffer in 64 by 8 blocks. This image block is then correlated with the bright template and surround template using a bit-correlation process to generate the brightsum and surroundsum, respectively. The brightsum and surroundsum are stored in the Frame Buffer as two 8 by 8 matrices, respectively. Since there are eight bit-planes, eight brightsum matrices and eight surroundsum matrices are generated.

The final step is peak detection. The shapesum is used to select one of the eight pairs of brightsum and surroundsum for each offset. The selected pair is compared against the predefined constant to generate a hit. This process is done for all of the 64 offsets and the 64 hit values are accumulated.

The steps discussed above are repeated after writing out the hit values and loading in another 64 by 8 block of the chip into Frame Buffer. This continues until the image block at each offset within the chip is correlated with the target templates. For a 64 by 64 chip and 8 by 8 templates, there are 57 (= 64 – 8 + 1) iterations.

Bit-correlation is required for the computation of the shapesum, brightsum, and surroundsum. The parallel mapping of this operation in the RC Array is depicted in Figure 71. Each row of the 8 by 8 binary template is packed as an 8-bit number. Each template is broadcast to the RC Array through the constant field of the context words. Each row of the RC Array receives the complete template. Two bytes (16 bits) of the image data are loaded to each RC from the Frame Buffer. Sixteen cycles are required to load data to the entire RC Array.

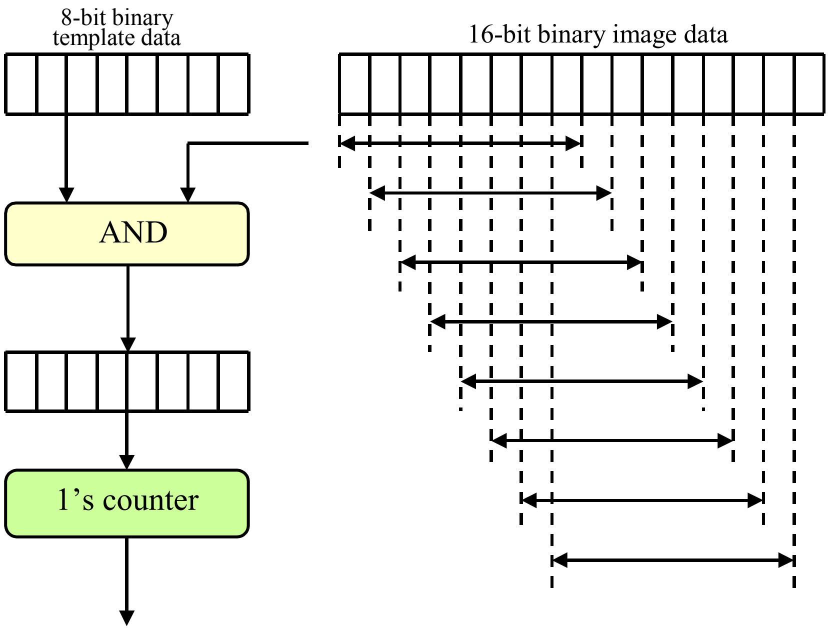


Figure 71. Bit-correlation in each RC

Each row of the RC Array performs correlation of its stored target template with one candidate block; hence, eight candidate blocks are correlated concurrently in the RC Array. In the first iteration, the most significant 8 bits of the image data in each RC are ANDed with the template data. The BTM RC function (see Table 15) is used to perform the AND operation and count the number of one's in the ANDed output (in one clock cycle). The values generated above are summed up for a row of RCs to generate the correlation result for one candidate block. This correlation is completed in only five cycles, and eight such correlations are performed in parallel. Next, the image data in each of the RCs is shifted left one bit and the bit-correlation process is repeated again to perform matching of the second candidate block. This continues until the image data is shifted eight times, and a total of 64 correlations are completed in the entire RC Array.

The computation above is accompanied by loading more image data into the Frame Buffer in parallel. Therefore, when the above computation (bit-correlation for 64 candidate blocks) concludes, a new 16-bit data is loaded into each RC and the bit-correlation process for the next set of 64 candidate blocks starts fresh in the RC Array.

The bit-correlation of 64 candidate blocks is completed in 56 cycles (with some variations due to differences in shapesum, brightsum, and surroundsum computations). This indicates that the MorphoSys M1 throughput for the Chunky SLD is more than one candidate block per cycle. Table 18 gives the MorphoSys M1 performance for Chunky SLD algorithm using a 64 by 8 image block and a pair of 8 by 8 bright and surround templates. Based on these figures, MorphoSys M1 requires about 2400 cycles to complete the Chunky SLD algorithm on a 64 by 8 image block, which leads to 1.4×10^5 cycles or 1.4 ms for M1 (at 100 MHz) to compute the hitcount for the entire 64 by 64 chip.

We used the Mojave system [29] Splash-2 systems [28] for performance comparison. Two Xilinx XC4013 FPGAs (one dynamic FPGA for the computation and one static FPGA for control) are used in the Mojave system. The Splash-2 system consists of 17 Xilinx 4010 FPGAs on each board. For ATR implementation, one FPGA is used for control and the other 16 FPGAs can be configured as 2 processing elements for Chunky SLD algorithm. In this study, the size of chip is 128 by 128 pixels, and template size is 8 by 8 bits. Figure 72 depicts the performance of these three systems for processing one pair of templates through Chunky SLD computations. The processing time for MorphoSys M1 (at 100 MHz) is 6 ms, while the corresponding time for the Mojave system (at 12.5 MHz) is 24 ms and the computation time for each processing element on the Splash-2 board (at 19 MHz) is 12 ms.

Table 18. MorphoSys M1 Performance for Chunky SLD

Kernels	**Cycles** (For 64 candidate blocks)	**Total cycles** (Include context and image data loading)
Shapesum	80	750
Brightsum	65	535
Surroundsum	65	535
Thresholding & Peak Detection	60	575

The Mojave and Splash-2 systems operate at a lower frequency than MorphoSys M1. The reasons for this difference of clock frequency between MorphoSys M1 and these two FPGA systems are the inherent long wire propagation delays for FPGA and the different technology used for these two systems. The Xilinx 4000 series uses 0.6-µm technology. Without loss of generality, we can scale the clock frequency by a factor of 3 when counting for 0.35-µm technology used for the design of MorphoSys.

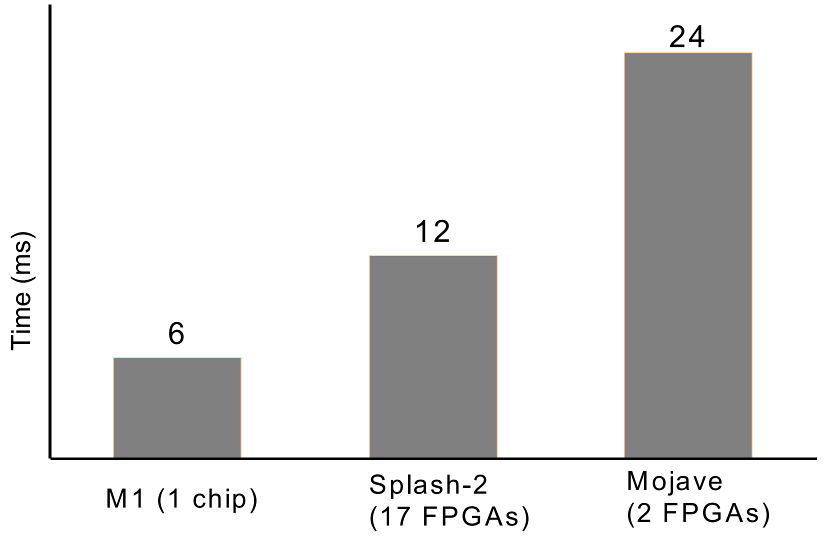


Figure 72. Performance Comparison for Chunky SLD Algorithm

Figure 73 depicts the performance comparison after the technology scaling is taken into account. Although MorphoSys is a coarse-grained system, it achieves a comparable performance compared to the FPGA-based systems (after accounting for speed scaling) for this fine-grain application. The FPGA-based systems are more appropriate for bit-level operations such as chucky SLD, and are inefficient for coarse-grain operations. These results demonstrate the flexibility of MorphoSys M1.

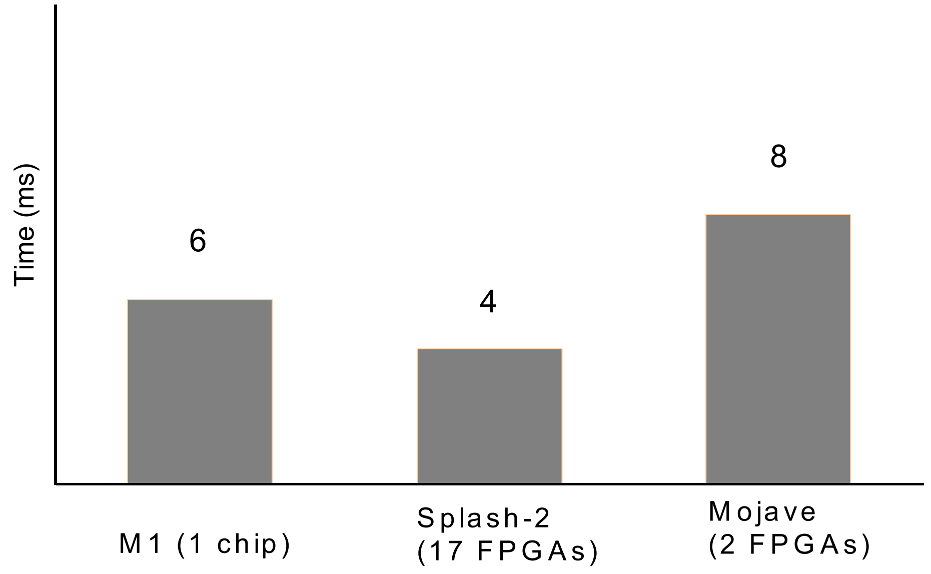


Figure 73. Performance Comparison for Chunky SLD after Scaling

A quantified measure of the ATR problem states that 100 image chips (128 by 128 pixels) have to be processed each second for a given target. Each target is represented by 40 binary template pairs (bright and surround) for every 5-degree rotation. For full 360-degree rotation, 72 by 40 = 2880 pairs of templates are required. Table 19 presents the number of system units (as defined in the table) required to process the Chunky SLD computations.

Table 19. System Requirements for ATR Specification

System	MorphoSys M1 (1 chip)	MorphoSys (4-M1 board)	Mojave (2 FPGAs)	Splash-2 (17 FPGAs)
Units required	1728	432	1728	6858

The Chunky SLD computation uses 6858 sets of the Mojave system. For Splash-2 system (17 FPGAs configured as two processing elements), 1728 boards are required. A total of 1728 MorphoSys M1 chips are enough to meet the specifications. If a MorphoSys board with 4 M1 chips is used, only 432 boards are needed.

4.3. Power Consumption Estimation

In this section, the power consumption for three applications, motion estimation, DCT/IDCT, and Chunky SLD, on MorphoSys M1 will be discussed. We used the following procedures to estimate the power consumption of MorphoSys M1.

1. Hspice is used to measure the power consumption of each block (e.g., TinyRISC, Frame Buffer, Context Memory, RC Array, and DMAC) independently. Worst-case scenario where each bit of the inputs switches in two consecutive clock cycles is

assumed. For each block, reasonable capacitive loads (extracted from the layout) are included at the outputs in the spice netlist.

2. For TinyRISC, we use instruction-based measurement. The power consumption associated with the basic processing needed to execute each instruction is simulated independently, and a table is built in which each instruction is associated with its own power consumption.
3. For the Frame Buffer, we use block-based measurement, and differentiate it into four power consumption models,
    - RC read: The power consumption of Frame Buffer when RC Array is accessing data from Frame buffer.
    - RC write: The power of Frame Buffer when RC Array is writing data to Frame Buffer.
    - DMAC read: The power consumption of Frame Buffer when DMAC is writing Frame Buffer's data back to main memory.
    - DMAC write: The power consumption of Frame Buffer when DMAC is loading image data to Frame Buffer.
4. For Context Memory, we separate the measurement into context write and context read.
5. For DMAC, the three different power consumption models are Context Memory load, Frame Buffer load, and main memory write.
6. We also use instruction-based measurement for the RC. For each function listed in Table 15, we use Hspice to measure its power consumption independently. The power consumption of the RC Array interconnection network is considered by replacing it with the corresponding capacitive load in the netlist.
7. To estimate the power consumption of M1 for an application, TinyRISC assembly and RC context assembly are used. The TinyRISC assembly provides the information required to estimate the power consumption for TinyRISC, Frame Buffer, Context Memory, and DMAC, and the RC context assembly provides the information required to compute the power consumption for the RC Array.
8. In the TinyRISC assembly, each instruction contains the information about the operation performed by TinyRISC, and the access type of Frame Buffer, Context Memory, and DMAC. The power consumption of each instruction can be calculated by this information. It should be noted that for some instructions, not all of the information is provided. In this case, for those blocks that the access type is not specified, the one that was previously used is assumed. For example, consider the TinyRISC assembly illustrated below:

```
LDFB $1, 0, 0, 16
LDCTX $2, 0, 1, 0, 128
CBCAST 1, 0, 1, 0
CBCAST 1, 0, 1, 1
WFBI 0, 0, 0, 0, 0
WFBI 1, 0, 0, 0, 8
STFB $1, 0, 0, 16
```

This program starts with loading image data in to the Frame Buffer and contexts into Context Memory through DMAC. When the *CBCAST* instruction is issued, both Frame Buffer and Context Memory are in read mode, and the access type of the DMAC is not specified. In this case, since DMAC was previously in Context Memory load mode, the same mode is used for this instruction. Using this method, the power consumption for each block, except RC Array can be calculated.

9. To estimate the power consumption of the RC Array, both the TinyRISC assembly and the RC context assembly are required. For '*CBCAST 1, 0, 1, 0*' instruction in the above example, every RC will access row context 0. The context assembly for row context 0 is illustrated below:

```
Set 0, 0 KEEP I I >0;
Set 1, 0 CLOAD!0x004 def def >2;
Set 2, 0 ADD HE r0 LSL 2 >0;
Set 3, 0 ADD XQ r0 LSL 2 >0;
Set 4, 0 SUB XQ r0 LSL 2 >0;
Set 5, 0 SUB HE r0 LSL 2 >0;
Set 6, 0 CMULSUB!0x0C3 r0 def LSR 9 >1;
Set 7, 0 KEEP I I >0;
```

The operations performed on all of the RCs are specified through the RC context assembly, and the power consumption of the RC Array can be calculated by accumulating the power consumption of each RC. For the TinyRISC instructions where no RC Array functions are specified, *No-Op* is assumed.

10. By adding the numbers from steps 8 and 9, we can get the power consumption associated with each instruction. Summing up the power consumption of each instruction and dividing it by the total number of instructions performed, we can get the average power consumption for an application.

Table 20 illustrates the average power consumption for the three applications, motion estimation, DCT/IDCT, and Chunky SLD using the methodology described above.

Table 20. Power Consumption Analysis for Motion Estimation, DCT/IDCT, and Chunky SLD

Motion Estimation	DCT/IDCT	Chunky SLD
4.8 W	5.5 W	4.3 W

The worst-case power consumption for MorphoSys M1 is 13.5 W. This is measured by assuming that MAC function is performed in every RC. Table 20 shows at least a 2 times power consumption reduction for motion estimation, DCT/IDCT, and Chunky SLD. This power saving is due to the selective enabling feature of the multiplier in each RC. The Hspice simulation shows that the power consumption of one RC is 200 mW when the MAC function is performed, and the multiplier alone is responsible for 150 mW. If the selective enabling of the multiplier was not implemented, MorphoSys M1 would have consumed about 13 W for these three applications. This demonstrates the importance of implementing the selective enabling feature for MorphoSys design.

5. M2 Design

Within the ongoing MorphoSys research project we have developed version M2 of MorphoSys, with some major functional enhancement over M1, as follows:

1. A local RAM for each RC, with auto-increment indexed mode addressing and cycle-stealing DMA to the FB. This memory is used to store intermediate values and local data.
2. Multiple RISC-type load instruction exploiting instruction level parallelism: up to three load operations can be packed into one 32-bit instruction, and fetched in one cycle, and then executed in the next three available cycles.
3. CISC-type store instructions: the result of arithmetic and logical operations may directly be stored in the local memory.
4. Complex arithmetic instructions supporting signed/unsigned operands.
5. 3-address multiply, ADD, and accumulate (MAC) instruction.
6. Register pair arithmetic shift instructions.
7. Guarded instructions, and pseudobranches, to achieve higher operation autonomy and higher instruction level parallelism than in M1.
8. A modified 32-bit, 4-stage pipelined central RISC processor (TinyRISC), with a Morpho decoder in charge of generating proper control signals for the RC array and all other peripherals in M2.

All blocks have been modeled, simulated and synthesized in a 0.13-µm technology. Several postsynthesis simulations have also been performed with a variety of test instructions, demonstrating the functionality of the modeled system. Three basic tools, namely mLoad, MuLate and TR assembler have also been modified to support M2.

As described earlier there are 8 by 8 RCs in a MorphoSys chip controlled by the TinyRISC, which is in charge of providing the RCs with the right *contexts* or instructions at the right times. Each context determines not only the specific operation for the corresponding RC, but those RCs that should supply the required operands as well. This programmability of operands makes the RC matrix interconnection *reconfigurable*. A context originating from the *Context Memory* may be broadcast to the matrix either vertically or horizontally under TinyRISC control, that is, 8 RCs share the same context but possibly with different data, realizing the concept of *SIMD machines*. Through a DMA controller the main memory is directly accessed by the Context Memory, which is shared by all 64 RCs.

Each RC is provided with external data by different sources, namely 10 different neighboring RCs, a centralized data memory called *Frame Buffer*, and Express Lanes, which facilitate the interquarter interconnection among the RCs. The Frame Buffer also has direct access to the main memory via a DMA controller. Notice that data transfer from the Frame Buffer to the RC matrix is also performed eight items at a time, that is, every 8 RCs have to share the same data if it is going to be received from the Frame Buffer.

Figure 74 illustrates a top-level block diagram of M2, consisting of 1) CPU, 2) SRAM, 3) SRAM Controller, 4) instruction cache, 5) instruction cache controller, 6) data cache, 7) data cache controller, 8) Morpho decoder (DCD), 9) Context Memory (CM), 10) Context Memory DMA controller, 11) Frame Buffer (FB), 12) Frame Buffer external DMA controller, 13) Frame Buffer internal DMA controller, and 14) RC Array.

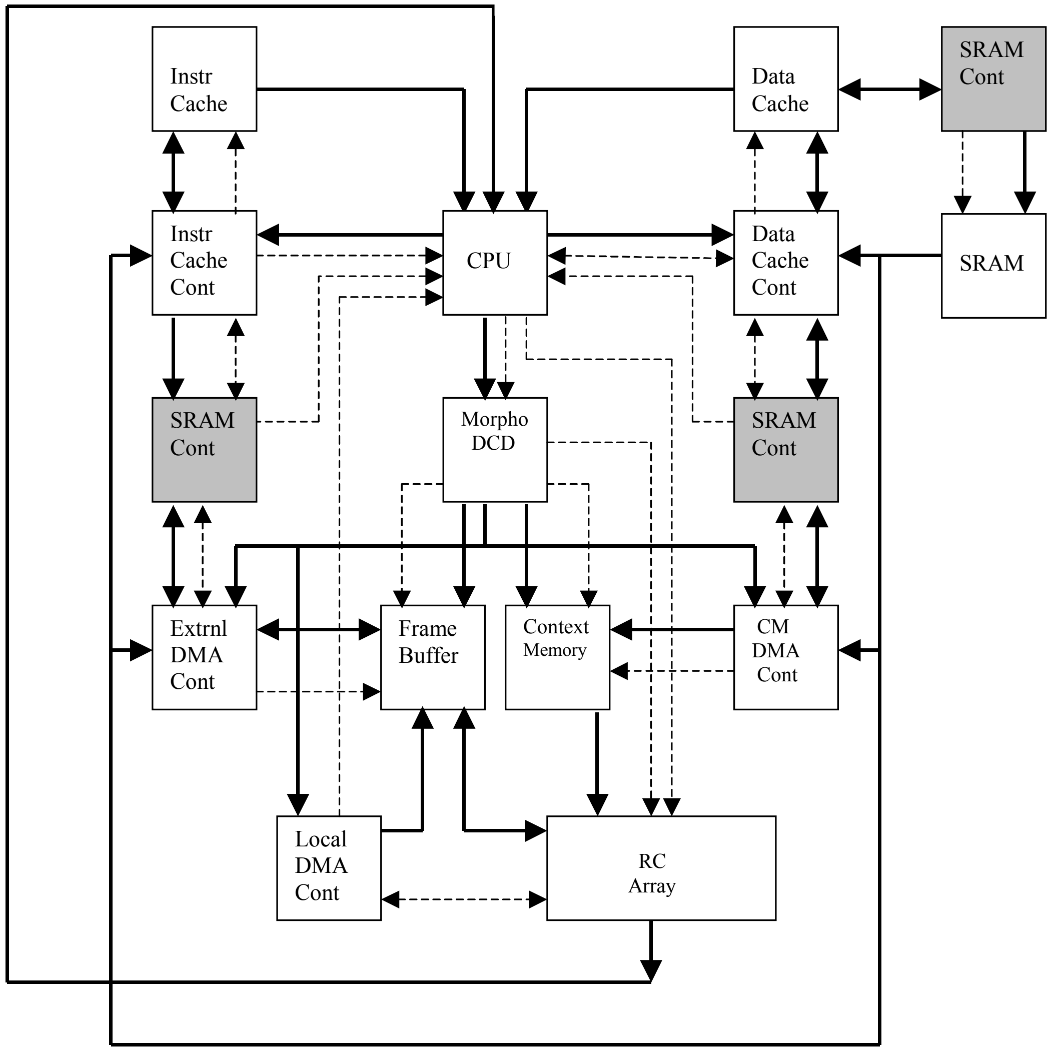


Figure 74. A Block Diagram of M2; *Solid*: Data/Address, *Dotted*: Control Signal

The CPU in conjunction with the Morpho decoder is in charge of providing the appropriate signals to broadcast data (originating from FB) and instructions (originating from CM) to the RC array, to initiate DMA data transfer between the SRAM and FB (through the FB external DMA controller), to initiate DMA instruction transfer between SRAM and CM (through CM DMA controller), and to initiate DMA data transfer between the FB and the local memory in each RC (through FB internal DMA controller). There are two instruction and data cache memories, which facilitate fast access by the CPU to the corresponding data and instructions in the SRAM.

Tools, Technology and Design Methodology Used In This Work

•RAM Generator: Artisan HS-SRAM-SP RAM Generator
TSMC 0.13-um Low-Voltage Process
•Compiler: Cadence NCVHDL, V3.11
•Elaborator: Cadence NCELAB, V3.11
•Simulator: Cadence NCSIM, V3.11
•Synthesis Tool: Ambit BuildGates, V4.0
•Technology: TSMC 0.13um LV LowK process
•Timing Library Format (TLF) V4.1, Vendor: Artisan Components, Inc.; and V4.3 converted from Synopsys .lib file.

Design Methodology

1. Model each module based on the specification, and then simulate and test.
2. Synthesize each module; do postsynthesis simulation and test.
1. Identify the major subsystems; do presynthesis and postsynthesis simulations and test.
2. For complex subsystems, the above stage may be performed in two or more substages.
3. If required, go back to stage 1 and revise the specification of one or more modules.
4. Use all synthesized modules in the test bench, do postsynthesis simulation, and test.
5. If required, go back to stage 1 and revise the specification of one or more modules.

Different blocks of M2 are discussed in the following sections:

5.1. TinyRISC

The CPU that controls the RC array and all of the peripherals in the M2 is a four-stage, pipelined, 32-bit RISC processor called the TinyRISC, or TR for short. We use CPU and TR interchangeably in this document. Figure 75 shows the three major blocks in the top-level TR.

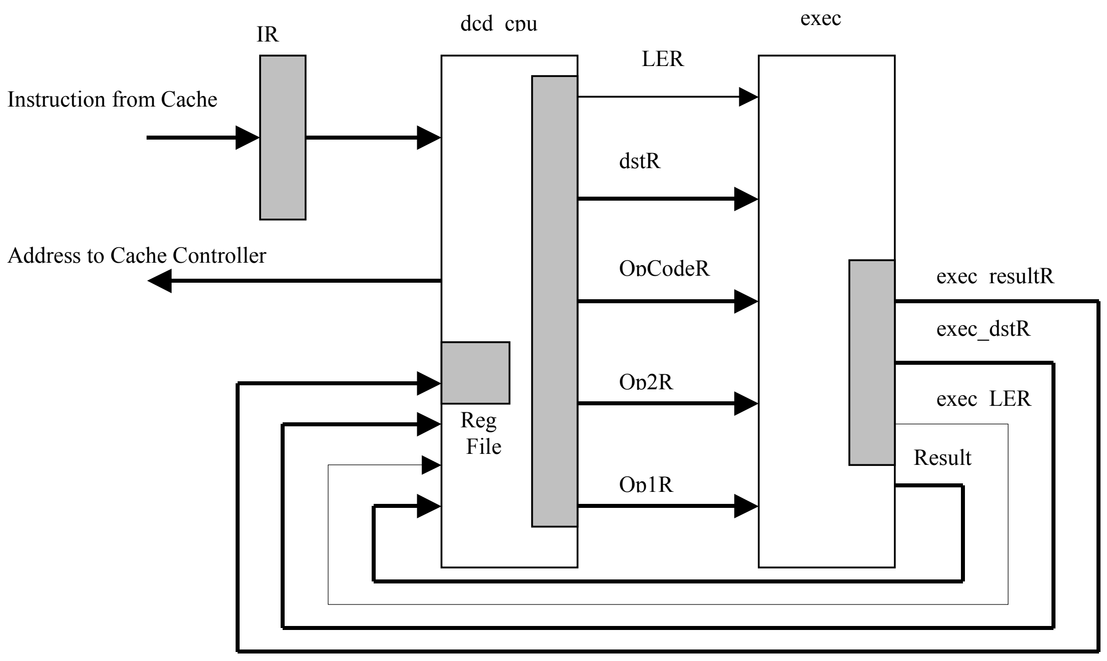


Figure 75. A Top-Level Block Diagram of the CPU

We categorize different instructions in M2 as follows:

1. Type A or TR instructions, which refer to the TinyRISC traditional instructions. These instructions deal with non-RC issues. Except for two new instructions (see Appendix F), we have used the same instruction set as the M1's.
2. Type B or Morpho instructions (Appendix G): the execution of an instruction from this group results in broadcasting a command (a type-C instruction) to the RC array.
3. Type C or contexts (Appendix H), which are instructions executable by the RC.

In this section TR instruction execution is discussed. The execution of type B instructions and type C instructions will be introduced later.

An instruction is first fetched into the IR (the Instruction Register), then the two operands, op1R and op2R, are properly worked out in block dcd_cpu, and then passed to the execution unit, exec, tagged with other information such as destination register (dstR), and the corresponding operation code (OpCodeR), and finally the result (if any) is written back into the register file located in the dcd_cpu.

5.1.1. dcd_cpu

A block diagram of dcd_cpu is illustrated in Figure 76. The major functions performed in this block are discussed here:

1. According to the TR instruction set two different instruction fields may carry the destination register code (dst). This ambiguity is resolved in dcd_cpu and then the dst is passed to the next stage, namely the execution unit.

2. All instructions that need data write back are identified, and the corresponding enable signal, that is, LE in Figure 76, is passed to the next pipeline stage.
3. Two 32-bit registers addressed by the two instruction fields (bits 16 to 19 and 20 to 23) are read out of the 16 by 32 register file in order to possibly participate in the corresponding data manipulation or branch instruction.
4. Data hazards are identified and removed by proper information fed back from the execution stage.
5. Other sources of possible operands are also considered here: The traditional immediate operands, and two external data namely status signals from the three DMA controllers, and the direct data from the RC array. As shown in Appendix F these two data may be read in a TR register by two different instructions, that is, *Read Status* and *Read RC*, respectively.
6. The 32-bit result of an instruction execution (if any) is written back into the register file. The destination code (exec_dstR), value (exec_resultR), and the load enable signal (exec_LER) are provided by the execution unit.
7. The Program Counter (PC) is updated in dcd_cpu. This register is either incremented or in case of a *successful jump* loaded with the appropriate address. Notice that with the TR architecture one-instruction delay for branch instructions is inevitable, that is, the instruction immediately following a jump instruction, which now resides in the IR, is always executed. A two-read-port register file has been used in this block, as two registers at a time have to be read out of the file. This feature simplifies the pipeline architecture and reduces the pipeline latency as well.

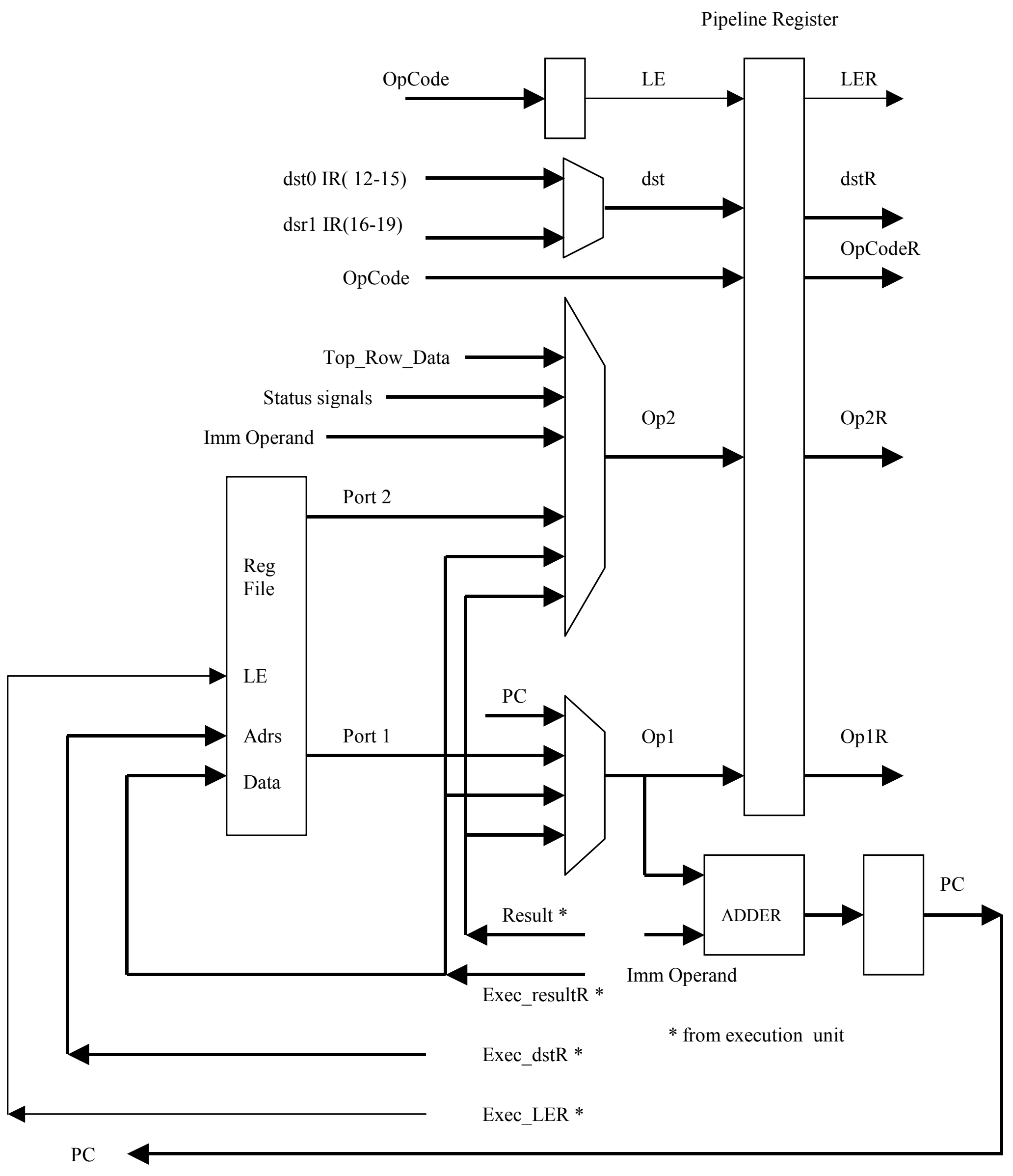


Figure 76. A Block Diagram of dcd_cpu.

5.1.2. Execution Unit

Figure 77 illustrates a block diagram of the execution unit.

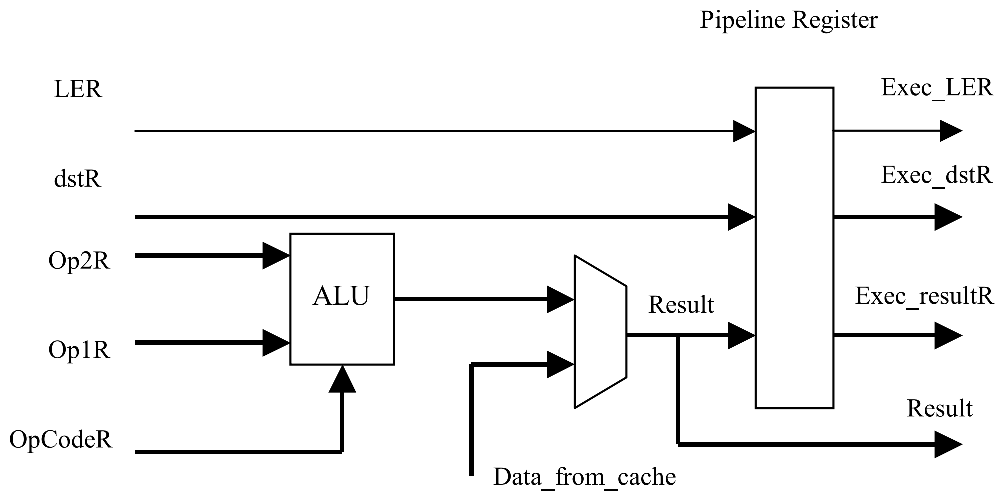


Figure 77. A Block Diagram of the Execution Unit

The major functions carried out in this unit are as follows:

6. An arithmetic or logic operation is performed on the two operands received from the previous pipeline stage.
7. In case of a memory reference instruction, the request is sent to the data cache controller. Furthermore, if this is a *read from memory* instruction, the corresponding data, either already available in the cache or fetched from the SRAM, will be delivered to the execution unit.

The following are the operations performed by the ALU:

1. ADD/SUB
2. Basic logic operations
3. Signed/unsigned, left/right shift
4. Signed/unsigned compare

Notice that there is no explicit flag register in TR. The result of different compare instructions is stored in a general-purpose register, which is then used as the condition for the following conditional branch instruction. For more details, see Appendix A.

5.2. SRAM

We have used a behavioral description for the off chip asynchronous static RAM in our simulations. The wait time for this slow memory can be a multiple of clock period specified as a 3-bit binary number in the SRAM controller. The SRAM has two unidirectional 32-bit data buses with active high write enable control signal.

5.3. SRAM Controller (Arbitrator)

There are four clients in M2 requesting read and/or write operations from/into the SRAM, namely the instruction cache controller, the data cache controller, the context memory

DMA controller, and the Frame Buffer external DMA controller. Any such request has to be made to the SRAM controller together with the corresponding address, the operation type and the possible data. Observing the fixed priority scheme mentioned above, the SRAM arbitrator performs the required task through a two-line fully responsive handshake protocol. Figure 78 shows the state diagram of this arbitrator.

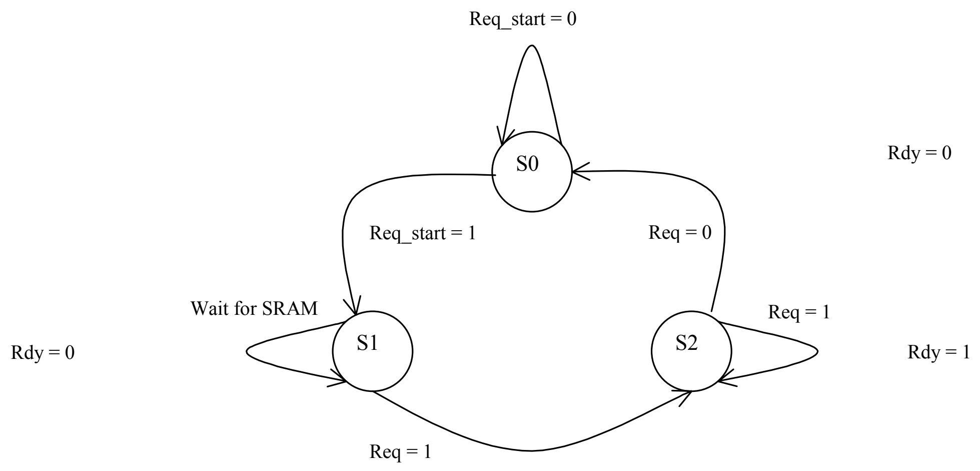


Figure 78. The State Diagram of the SRAM Controller

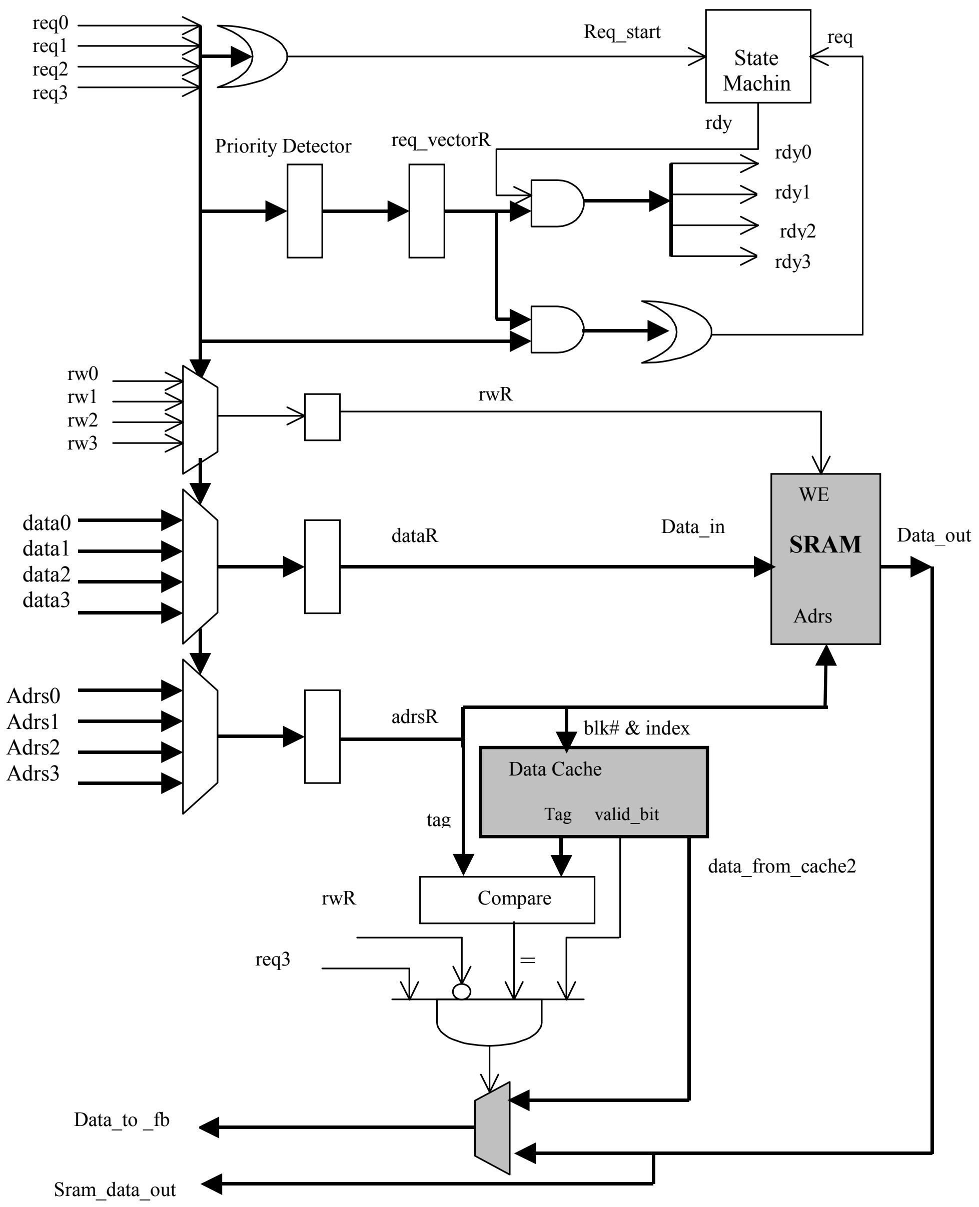


Figure 79. A Block Diagram of the SRAM Controller (excluding shaded areas)

S0 is the reset state. As soon as a request arrives the arbitrator jumps to S1, in which the SRAM is addressed while the data cache is also searched if the request is a *read* and is issued by the frame buffer external DMA controller. This concurrent search solves the cache coherency problem when the SRAM is read by the frame buffer external DMA controller. Notice that the other three clients only need the SRAM to be looked up. The arbitrator leaves S1 for S2 if a hit occurs or the SRAM does the requested function. Finally, the reset state is reached when the request signal is deactivated by the corresponding client. Figure 79 shows a block diagram of the SRAM controller. Notice that port 0 has the highest priority.

Any request initiates the state machine and makes it leave the reset state; however, only the one with the highest priority among the requesting ports will be acknowledged and served. The pending requests have to wait until there are no more requests with higher priority. If a request is received while a lower priority request is being served, the SRAM arbitrator will ignore the new request until the current transfer cycle is carried out and the machine returns to S0.

5.4. Instruction cache

As shown in Figure 80 a 32-word direct mapped instruction cache with a block size of 4 has been utilized in M2 based on the direct mapping technique. This implies 25-, 5- and 2- bit-wide *tag*, *index,* and *block-number* fields, respectively, in a 32-bit-wide address assumed in TR, resulting in a 25 by 32 tag memory, a 4 by 32 by 32 cache core, and a 1 by 32 valid bit memory, or 4928 bits of memory cells in total.

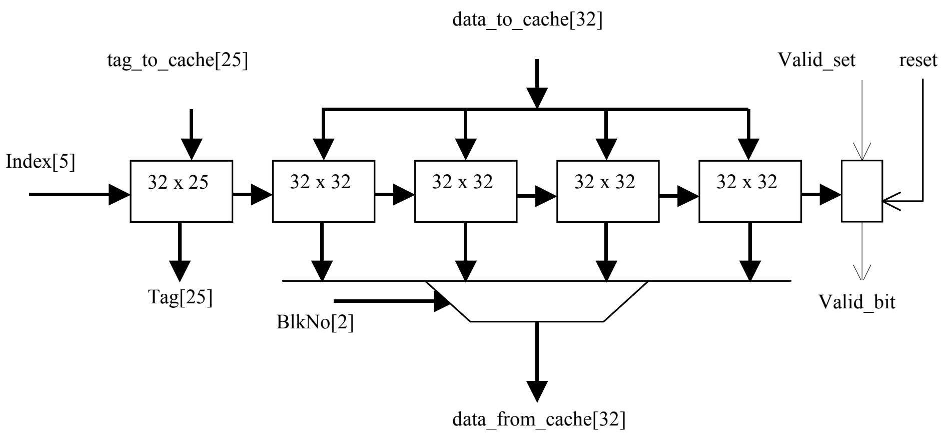


Figure 80. A Block Diagram of the Instruction Cache

Using D-type FFs in this memory, the read operation becomes asynchronous, while a clock edge is required to write into the cache.

5.5. Instruction Cache Controller

Figure 81 shows how the instruction cache controller (which has the highest priority in using the SRAM) is sitting among its neighbors. It receives instruction address streams from the CPU and stalls the CPU if an instruction-cache-miss happens. The CPU can also be stalled by the data cache controller as explained before.

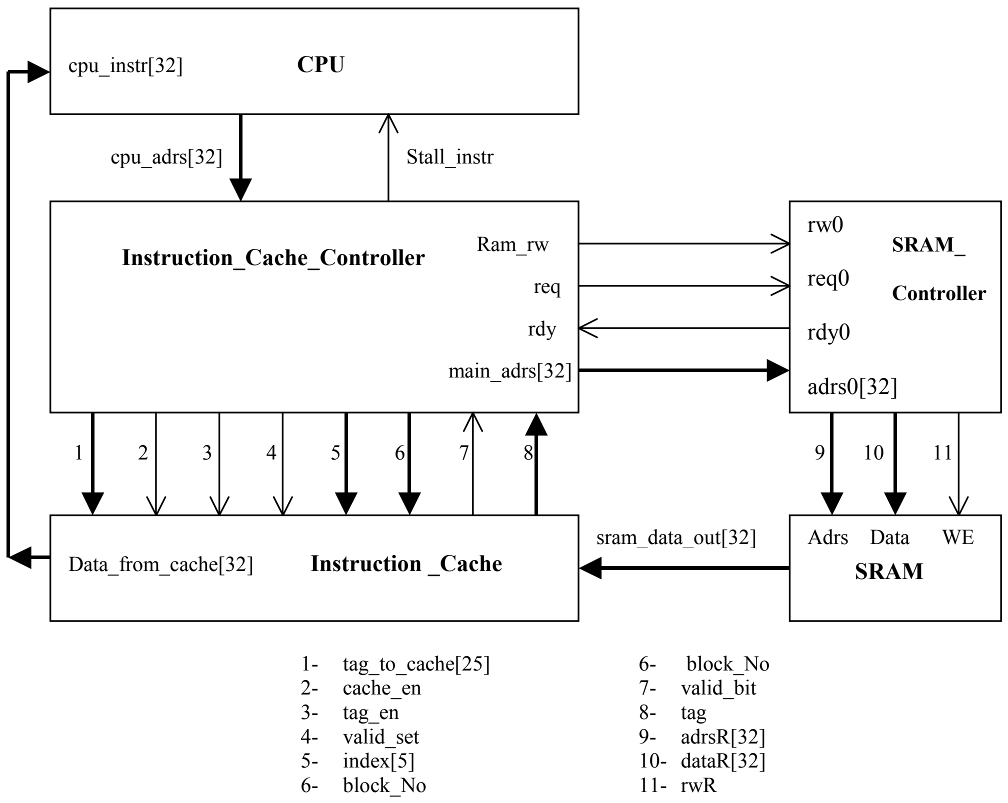


Figure 81. Instruction Cache and the Neighboring Blocks

The state diagram of the instruction cache controller, which interacts with the SRAM arbitrator, is shown in Figure 82. An instruction cache miss forces the controller to leave the reset state, S0. Then the S2-S4 transfer loop is carried out four times, (that is, equal to the cache block size), after which S0 is reached again.

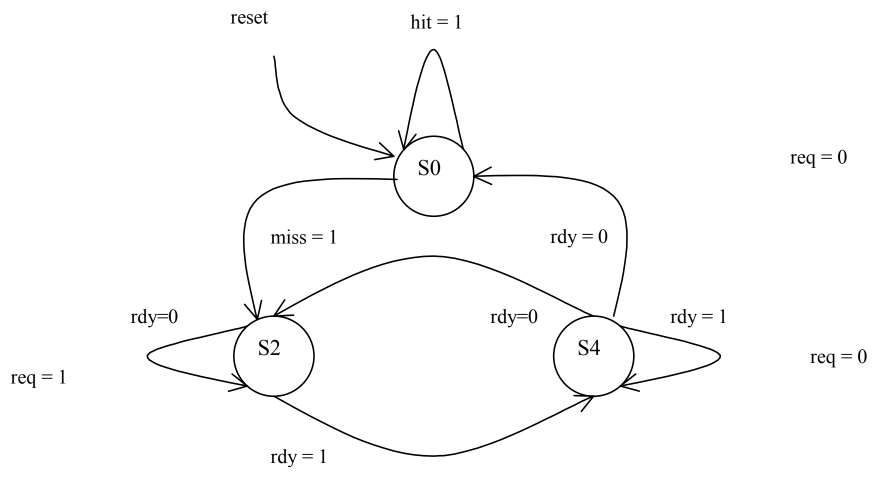


Figure 82. The State Diagram of the Instruction Cache Controller

Data Cache Memory

M2 has a 32 by 32 direct mapped, write back, two-port data cache with a block size of 4. Therefore, the three tag, index, and block-number fields remain the same as those in the instruction cache. Figure 83 illustrates a block diagram of the data cache memory. Comparing to the instruction cache, a dirty register has been added to the data cache in order to mark the overwritten blocks of the cache to be written back into the SRAM during the write back procedure, as explained in the following section.

In addition to the data cache controller, and in order to avoid the cache-coherency problem, the SRAM controller also has to have read access to the data cache. This feature has been realized by using a two-read-port data cache. Figure 84 shows how the data cache is positioned in M2.

5.6. Data Cache Controller

As a traditional load/store machine, the TR may only have a memory access request when either of the two read or write instructions happen to appear in the execution unit.

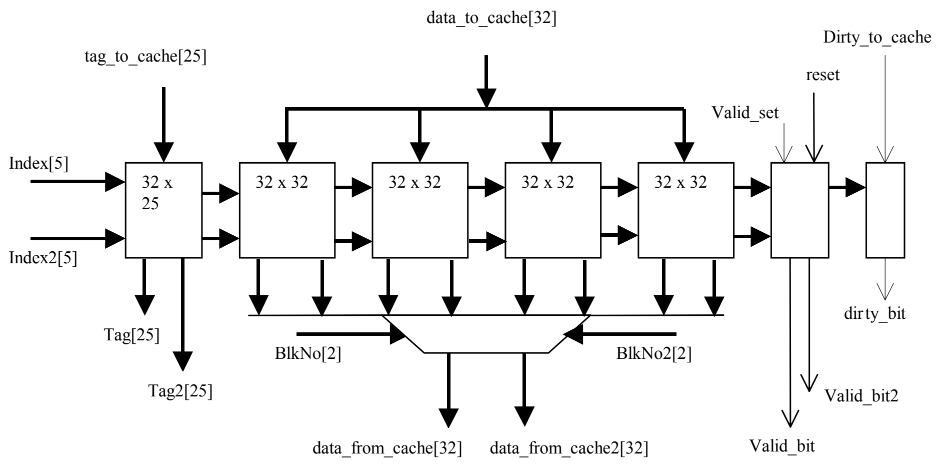


Figure 83. A Block Diagram of the Two-Port Data Cache

Now OP1R and OP2R in Figure 84 become the corresponding address and data (for write instruction only). Furthermore, the request type is specified by the two lines, namely cacheR and cacheW. The cache controller will stall the CPU and perform a possible block write back and a block fetch cycle, as illustrated in Figure 85 if a data cache miss occurs. A miss makes the cache controller leave the reset state, S0. The destination state is either S2 or S4 if the intended block has been overwritten or not, respectively. Both S2-S3 and S4-S5 data transfer loops are again performed four times, that is, equal to the block size. Notice that both loops interact with the same state diagram of the SRAM controller shown in Figure 78. As soon as the state machine returns to S0, the stall_data signal is deasserted, letting the CPU continue its stalled instruction execution. Notice further that the SRAM controller search into the data cache does not intervene in the data cache and its controller interactions, as the data cache has two ports.

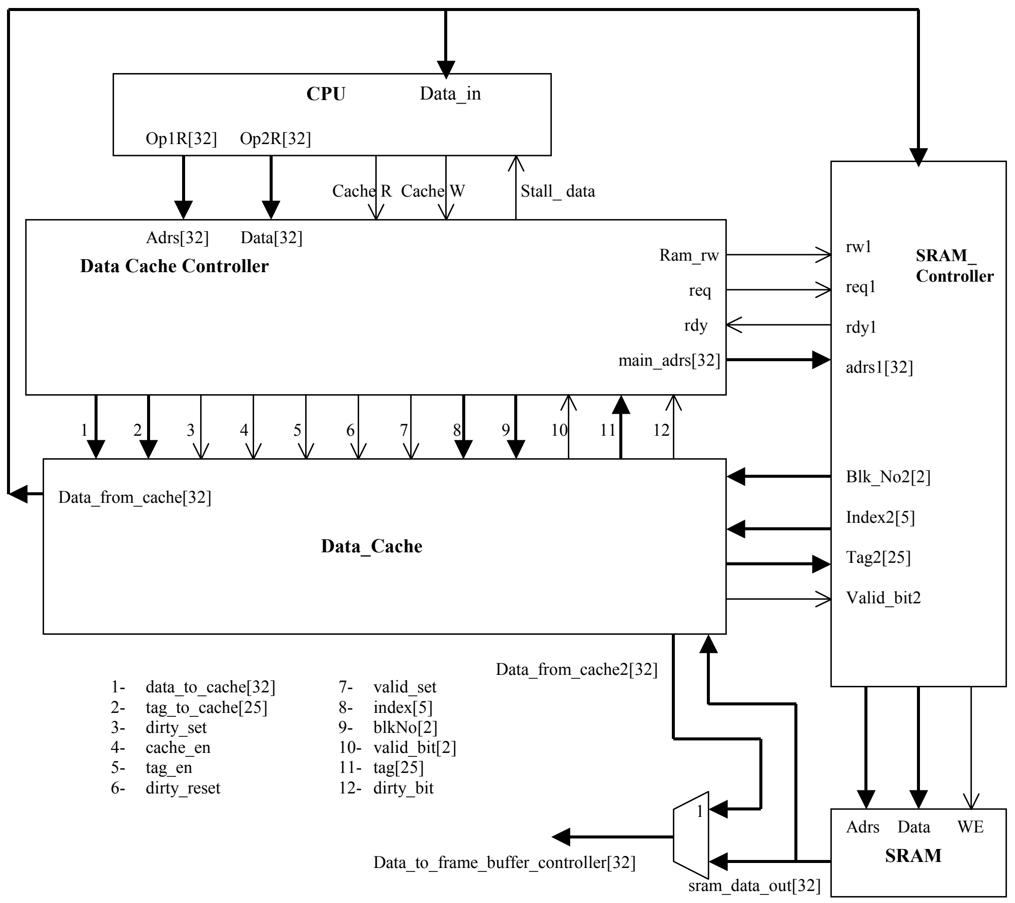


Figure 84. Data Cache and the Neighboring Blocks

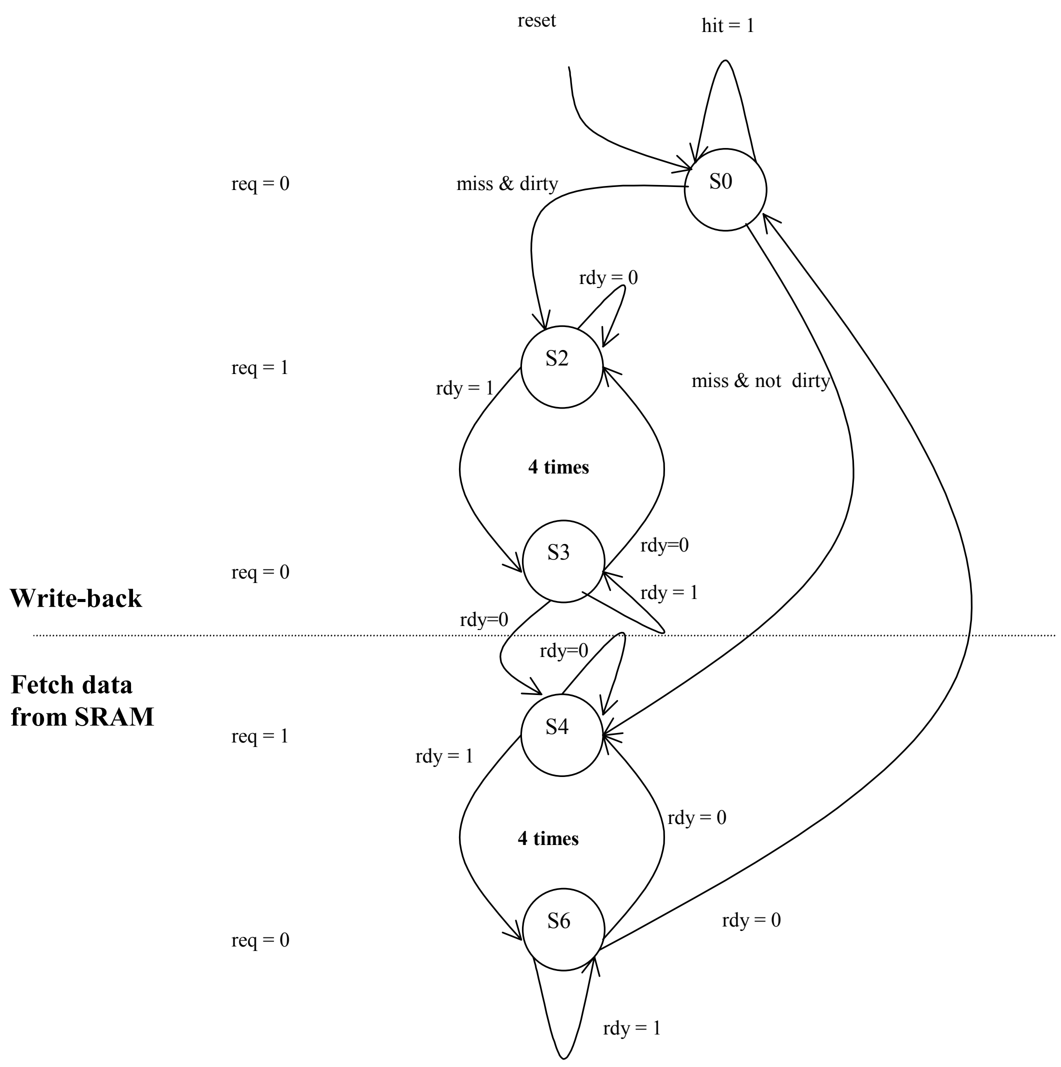


Figure 85. The State Diagram of the Data Cache Controller

5.7. Morpho Decoder

As far as the CPU pipeline stages are concerned, the Morpho decoder is sitting in parallel with the execution unit of TR, handling type-B instructions and generating 1) the signals required to control the three DMA controllers (subtype B1 instructions), 2) most of the control signals directed to the frame buffer and the context memory (subtype B2 instructions), and 3) the control signals supporting subtype B3 instructions. Notice that the rest of the control signals for these two memories are generated by the corresponding DMA controllers. Subtype B.1 instructions are translated to a 57-bit-wide command shown in Figure 86, carried on a shared bus and directed to the three DMA controllers. The intended DMA controller is notified of the CPU's decision when the unique device number allocated to each of them together with the transfer direction bits (if any) is identified in the 57-bit-wide command.

```
56        53 52                    21  20                  9  8              0
  device#          SRAM Address            CM or FB Address     No of planes

     0111:  SRAM to CM
     0010:  SRAM to FB
     0001:  FB to SRAM
     0110:  local RAM to FB; see the following pattern
     1110:  FB to local RAM; see the following pattern

56        53 52                25  24     22  21   20            9  8           0
device #               X             col#    bank      FB address     word count
```

Figure 86. The 57-bit-wide Command Broadcast to the three DMA Controllers

Figure 87 shows a simple top-level view highlighting how a second pipeline is shaped up in M2. Notice that both the frame buffer and the context memory are synchronous, which make up the second stage of the new pipeline staring from the execution unit of the CPU. The context and data registers sitting in each RC are the last stage of this pipeline. Notice that both pipelines are stalled at the same time.

5.8. Context Memory

The CPU stores the intended type B instructions or *contexts* in the context memory from the SRAM through a DMA transfer procedure. Then these contexts can be broadcast eight at a time to the intended RCs in the RC array either column wise or row wise, as one of the type B2 or Morpho instructions is executed. As shown in Figure 88, there are two memory banks, horizontal and vertical, in the context memory, each 512 by 256 in size, which can be written by the context memory DMA controller and read by the CPU when a context broadcast instruction (subtype B2) is executed. The corresponding 256-bit data buses of the horizontal and vertical banks run horizontally and vertically, respectively, in the RC array to facilitate horizontal and vertical broadcast, respectively, in which one specific column or row, respectively, will receive eight different contexts on the corresponding 256-bit-wide data bus. Notice that the context memory is 32-bit word aligned, that is, every 256-bit-wide addressed row (one context plane) may begin with any of the eight 32-bit words located in that row. This feature requires extra three bits, resulting in a 12-bit address bus for the context memory. The double bank context memory used in M2 may overlap a context broadcast from one bank and a DMA transfer to the other one, resulting in a possible performance improvement.

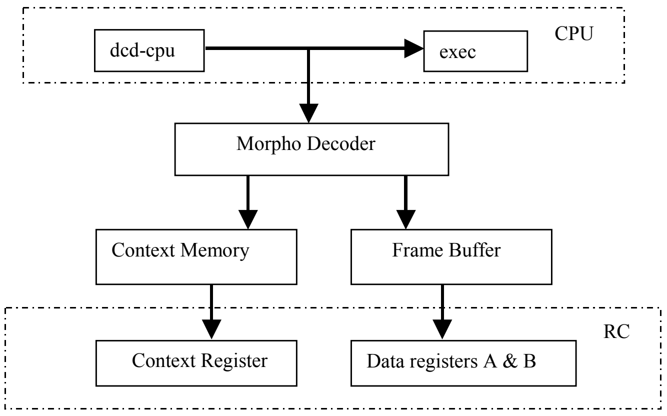


Figure 87. A Block Diagram of the Second Pipeline in M2

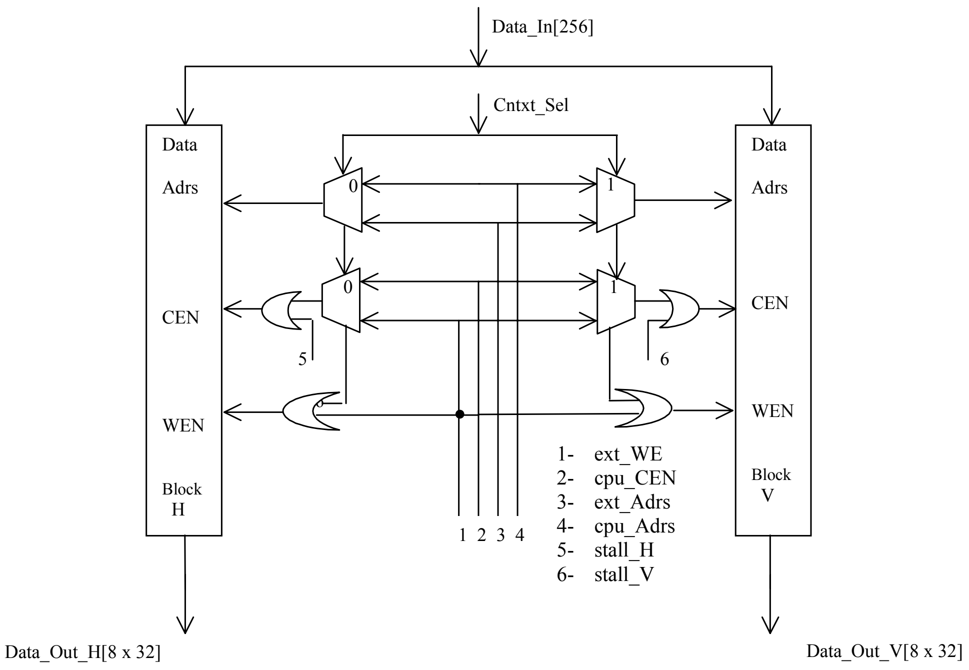


Figure 88. A Block Diagram of Context Memory

5.9. Context Memory DMA Controller

Figure 89 shows the context memory controller interface with its neighboring blocks. The Morpho decoder broadcasts the relevant 57-bit command on the corresponding bus specifying the device number allocated to the context memory DMA controller. Then the DMA controller communicates with the SRAM controller (if it is available) and the context memory to perform data transfer according to the state diagram presented in Figure 90. In order to get this data transfer carried out properly, the DMA controller also needs to perform data packing, as the data bus sizes are 32 and 256 for the SRAM and the context memory, respectively.

As soon as the unique device number allocated to the context memory DMA controller appears in the command the DMA controller leaves the reset state, S0. Then the S1-S2 transfer loop is iterated 8 by n times, where n is the number of context planes to be transferred to the context memory.

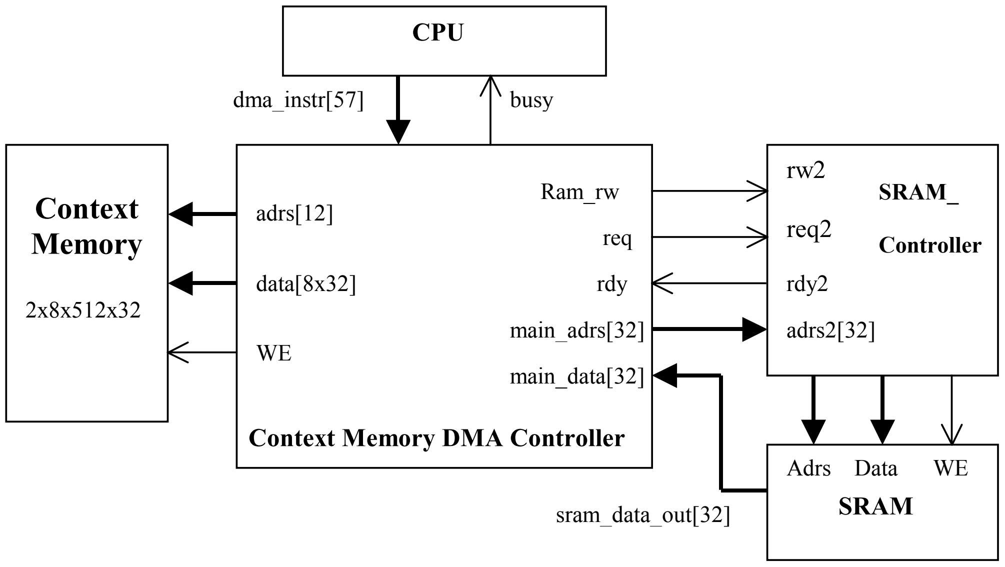


Figure 89. Context Memory DMA Controller and the Neighboring Blocks

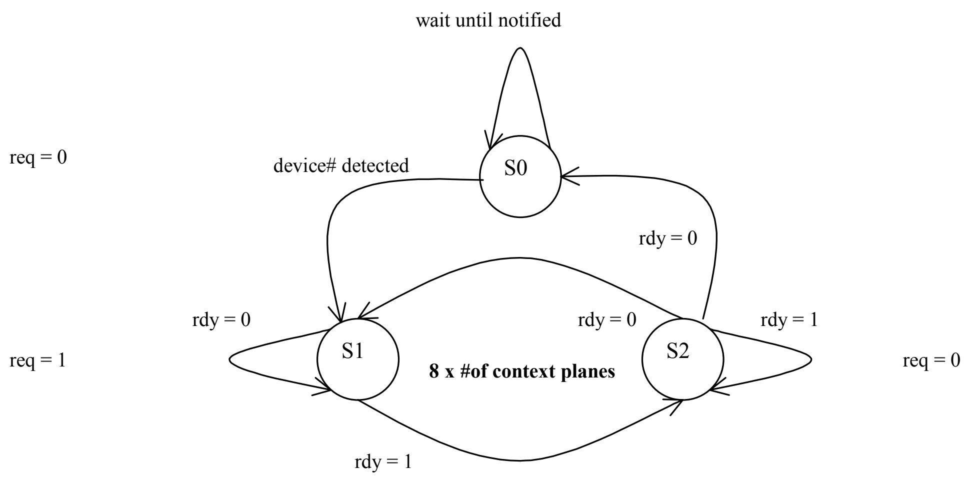


Figure 90. The State Diagram of the Context Memory DMA Controller

5.10. Frame Buffer

The Frame Buffer is a global, on-chip data memory shared by all 64 RCs, with 2 sets, 2 banks/set, 8 parallel blocks/bank, and 512 by 16 bits/block (or 32 KB in total), so that each bank (comprising 8 blocks) is organized as a 512 by 128 bit array, as shown in Figure 91. Notice that the 128-bit-wide words of these arrays are word aligned. The alignment is determined by the extra 3 bits appended to the address buses, increasing them to 12 lines. The 32 identical blocks have been generated by the Artisan RAM generator. There are three clients reading/writing from/to FB, namely, the internal and external DMA controllers (DMAC), and the CPU itself through their dedicated address buses. The external DMA controller has its own 128-bit-wide input and output data buses, while the other two address buses share the same 128-bit input and 2 by 128-bit output data buses, as shown in Figure 94. These two 8 by 16-bit-wide buses run horizontally throughout the RC's network, feeding eight RCs located on a column with different data. This architecture forces all eight RCs located on any row to receive the same FB data or use non-FB operands during that specific clock cycle. The 8 by 16-bit input data bus, on the other hand, carries eight words from eight RCs located on one column at a time. The rest of the RCs have to be prevented from outputting data on this bus in order to avoid bus contention; in other words, only one column at a time may be read out. Any access to FB is limited to one set at a time. Furthermore, all writes to FB are carried out on one bank at a time as well. While FB allows the CPU to read two banks at a time (with the same addresses), the external DMAC read operation is limited to only one bank at a time. As illustrated in Figure 92 the two DMACs have simultaneous but mutually exclusive access to the two sets of FB. The CPU, on the other hand, has priority to use either of the sets. The unused set can still be used by the external DMAC but not by the other one for obvious reasons. Notice that because of the bandwidth of 16 of the local RAMs, internal DMA transfer may only occur one bank at a time; however, the programmed I/O technique under the CPU's control may use both banks in the intended set to provide the addressed RCs with two operands.

As mentioned above, the memory cells in FB all are synchronous SRAMs, so that the data read out of FB becomes valid at the end of the clock cycle during which the corresponding address has been valid. Therefore, the data read out of FB does not become available in the destination register until the end of the following clock cycle.

5.11. Frame Buffer External DMA Controller

Figure 92 shows the interface of the frame buffer DMA controller with its adjacent blocks. The DMA controller monitors the device number field in the 57-bit-wide DMA instruction bus and leaves the reset state, as shown in Figure 93, as soon as a match between that field and its own device number occurs. The data transfers "from SRAM" and "to SRAM" happen in the left and right main loops, respectively, of the state diagram. To get this data transfer carried out properly, the DMA controller needs to perform either data packing (for the transfers from the SRAM to the context memory) or data unpacking (for the transfers from the context memory to the SRAM) as the data bus sizes are 32 and 126 for the SRAM and the frame buffer, respectively. That is why the inner loops S1-S2 and S5-S6 are iterated 4 by n times.

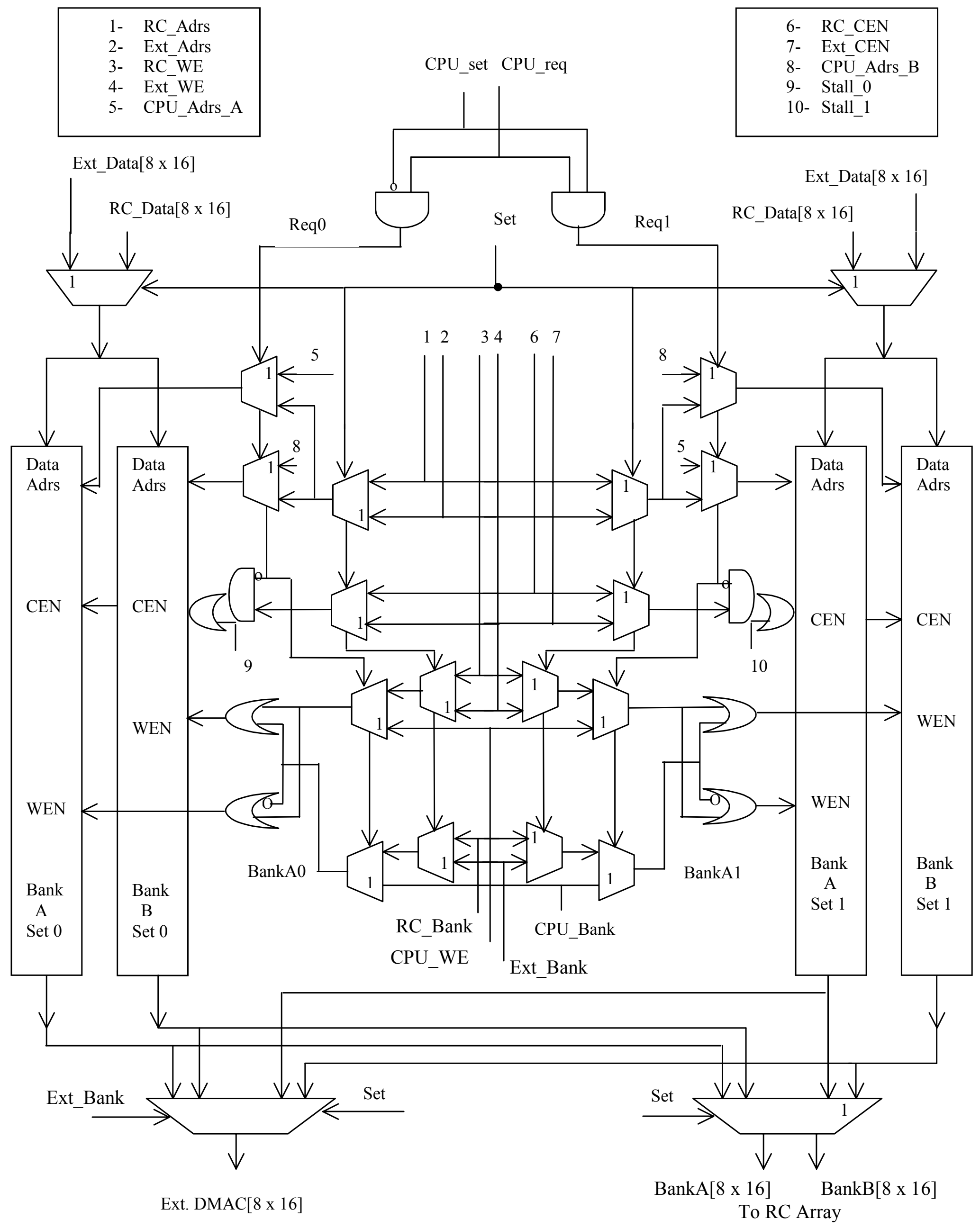


Figure 91. A Block Diagram of Frame Buffer

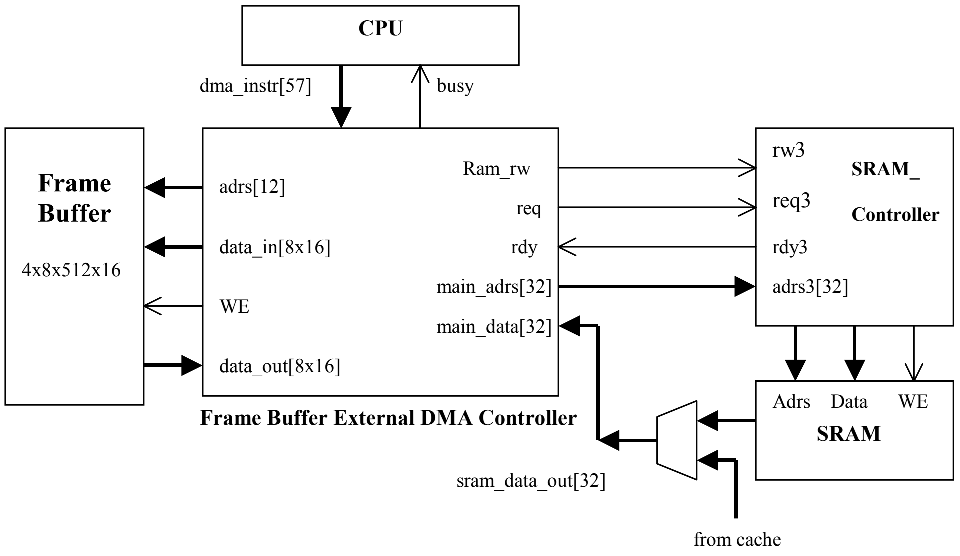


Figure 92. Frame Buffer DMA Controller and the Neighboring Blocks

5.12. Frame Buffer Internal DMA Controller

As mentioned above, the FB may be used by either the two internal and external DMA controllers or the external DMA controller and the CPU itself simultaneously. The internal DMA controller is in charge of transferring data bi-directionally between the FB and the eight local RAMs of eight RCs located on any column of the RC array, where the RC side has priority to use the local RAM over the DMA controller side. Figure 94 illustrates how this controller is interfaced with the neighboring blocks. Notice that unlike the other two DMA controllers, the memory pointer of the local RAM is located inside each RC, which is initialized by a specific context (type-C instruction). Therefore, every RC may have its own local memory starting address for data transfer. As implied by Figure 93, there are eight dmaReq and eight dmaRdy lines carrying the hand shake signals in this data transfer. We have used a different handshake protocol from what was utilized in the other two controllers, as each 128-bit-long word transfer between the FB and the local memories occurs in one clock cycle.

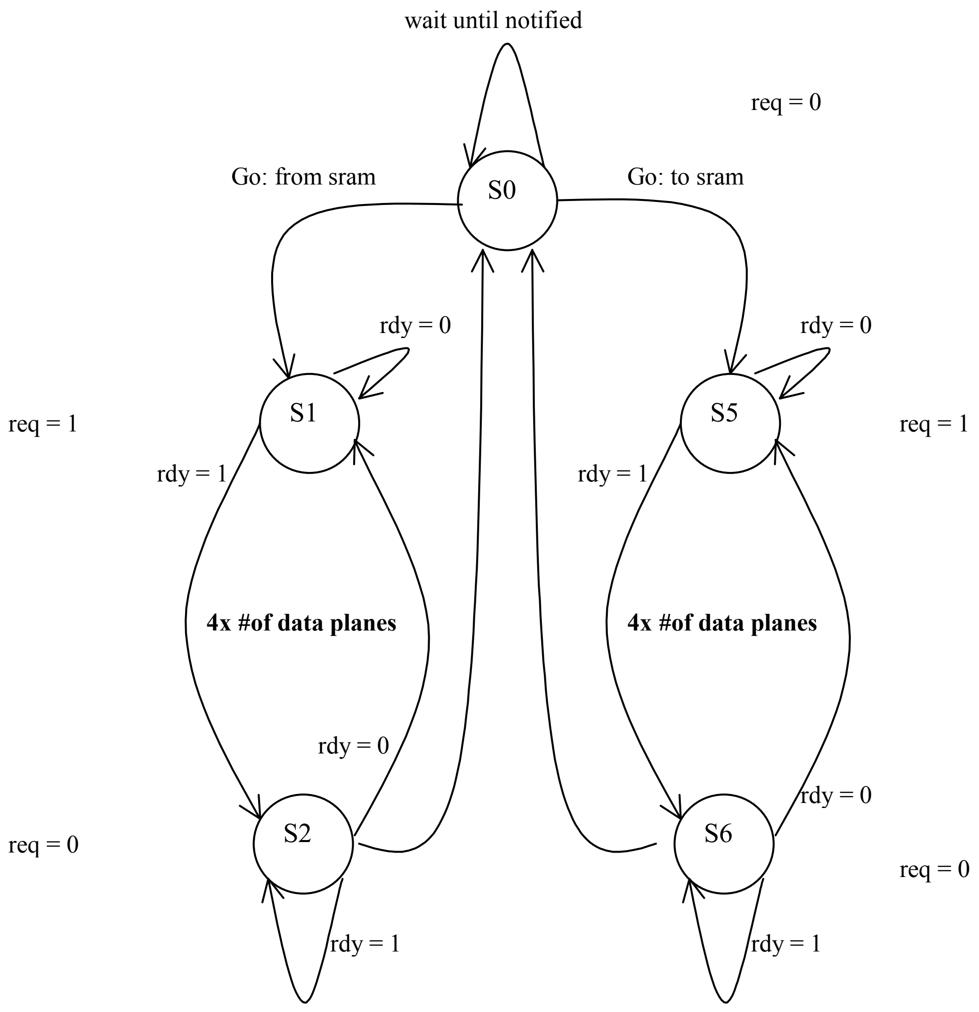


Figure 93. The State Diagram of the Frame Buffer External DMA Controller

As soon as the internal DMA controller receives a relevant command, the corresponding request line is asserted. This lets the DMA controller use the eight local RAMs if they are not used by the RCs themselves. The intended RCs notify the DMA controller of their need by pulling the ready line down. Therefore, the data transfer occurs in a cycle stealing manner. The internal memory pointer is updated after every access to the local memory. As shown in Figure 94, the DMA controller also provides the FB with a 12-bit address (including word alignment bits) and two other control signals, WEN and CEN. Notice that the DMA controller is not concerned about the set number and block number, as they have already been set properly by the Morpho decoder. Only one RC is shown in Figure 94. The other RCs are similarly located in the system.

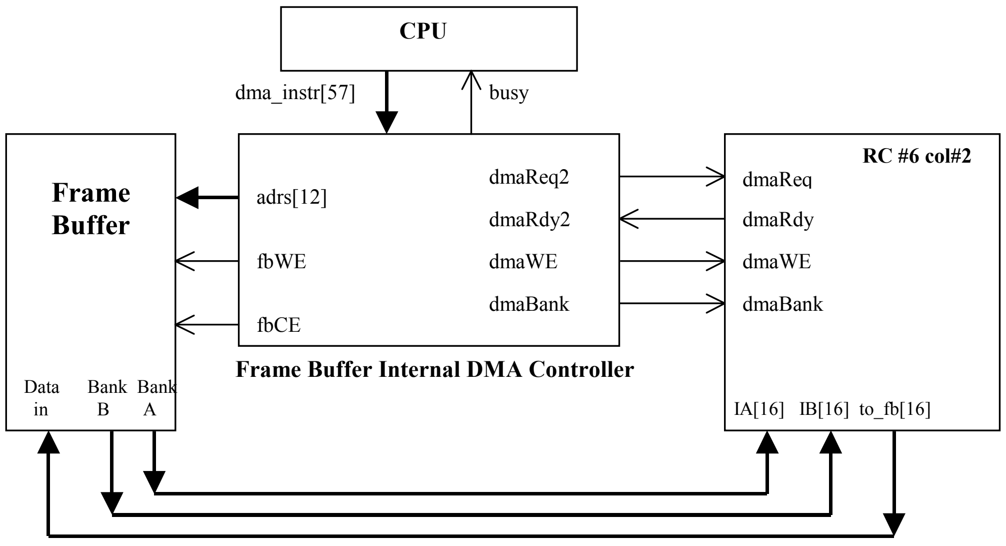


Figure 94. Frame Buffer Internal DMA Controller and the Neighboring Blocks

The state diagram of the internal DMA controller is depicted in Figure 95. Similar to the other two DMA controllers, as soon as the allocated device number is detected on the command bus, the internal DMA controller leaves the reset state and enters its busy mode. There are again two main loops in the state diagram controlling "read from frame buffer" (right) and "read from local RAM" (left). Since both the destination and source memories are synchronous in this data transfer, and furthermore, the data transfer is interrupted whenever the local memory is accessed through a memory reference context, the controlling logic is more complicated than those in the other two DMA controllers. For more details, refer to the VHDL source code.

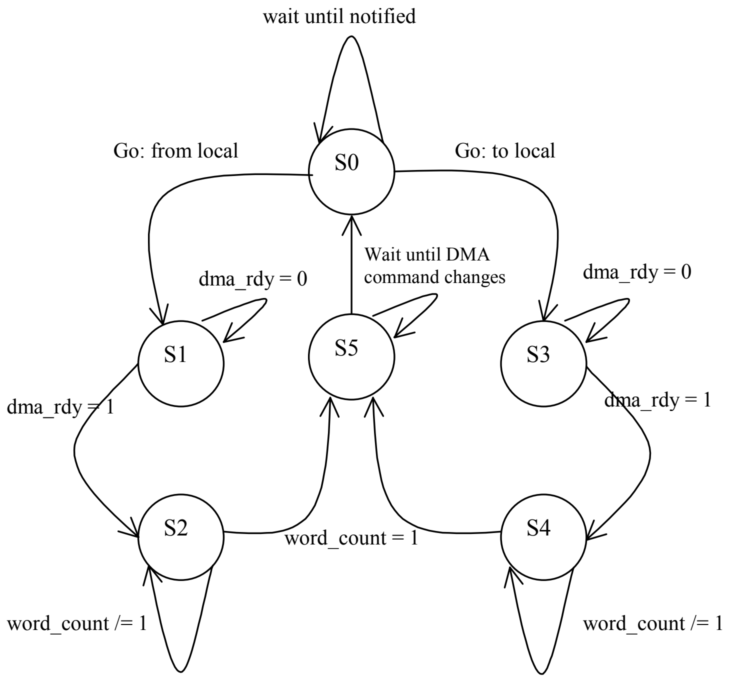


Figure 95. The State Diagram of the Frame Buffer Internal DMA Controller

5.13. RC Array

The RC array is comprised of 64 RCs arranged as an 8 by 8 matrix. In this section, one single RC is discussed first, and then the RC array bus structure is demonstrated.

RC Instruction Set: Figure 96 illustrates a basic block diagram of version M2 of RC whose instructions are categorized and described in this section. Notice that the instructions in groups 1 and 4 are *guarded*. Furthermore, the flag register may remain unchanged under the instruction control. Also notice that the result worked out in groups 1, 2, and 4 can be shifted either right or left by a maximum of 15 bits.

Legend: Rd: R15 and a register in Register File

Rs: A register in Register File
A & B: Two operands from MuxA & MuxB
Rp: A register pair

1- ALU Operations:

Bypass:	Rd <= A
OR AB:	Rd <= A or B
AND AB:	Rd <= A and B
XOR AB:	Rd <= A xor B
ADD AB:	Rd <= A + B
SUB AB:	Rd <= A-B

SUB BA:	Rd <= B-A
ABS AB:	Rd <= |A-B|
CX ADD:	Rd <= Cmplx (A+B)
CX SUB:	Rd <= Cmplx (A-B)
RESET:	Rd's <= 0

2- MAC Operations: Notice that here the destination in the register file may be either a single or double register.

UC MUL:	Rd <= Unsigned Cmplx (A*B)	
C MUL:	Rd <= Signed Cmplx (A*B)	
U MUL:	Rd <= Unsigned (A*B)	
MUL:	Rd <= Signed (A*B)	
MUL ADD:	Rd <= A* Rs + B	-- R15 can replace B
C MUL ADD:	Rd <= Cmplx(A* Rs + B)	-- R15 can replace B
MUL SUB:	Rd <= A* Rs - B	-- R15 can replace B
C MUL SUB:	Rd <= Cmplx(A* Rs - B)	-- R15 can replace B

3- Memory Operations:

LD IMM:	Rd <= Const;
LD MEM:	Rd0 <= Mem[Rs0], Rd1 <= Mem[Rs1], Rd2 <= Mem[Rs2];
LD X MEM:	Rd <= Mem[Rs + Rx]; Rx <= Rx + 1;
LD Rx :	Rx <= Rd;
LD DMA R:	Rdma <= Rs;
ST MEM:	Mem[Rd] <= Rs;

4- Miscellaneous Operations:

INCMT:	Rd <= A+1
DECMT :	Rd <= A-1
CNTLZ:	Rd <= # of leading zeros (if pos)/leading ones (if Neg) in A
CNTLZ32:	Rd <= # of leading zeros (if pos)/leading ones (if Neg) on Reg pair
WAKUP:	Compare Saved Label with Instruction Label, Wake up if match
PBRAN:	Sleep until WAKUP
SHIFT:	Rd<=Shift-Fun(A); Shift-Fun performs different types of shifts and rotations
A_SHIFT:	Rpd <= arithmetic shift (Rps)
ABS A:	Rd <= |A|
ADDSUB:	Rd <= A+B if Cond True, else A-B
ADDCC:	Rd <= A+B + Carry
SUBCC:	Rd <= A-B - Borrow

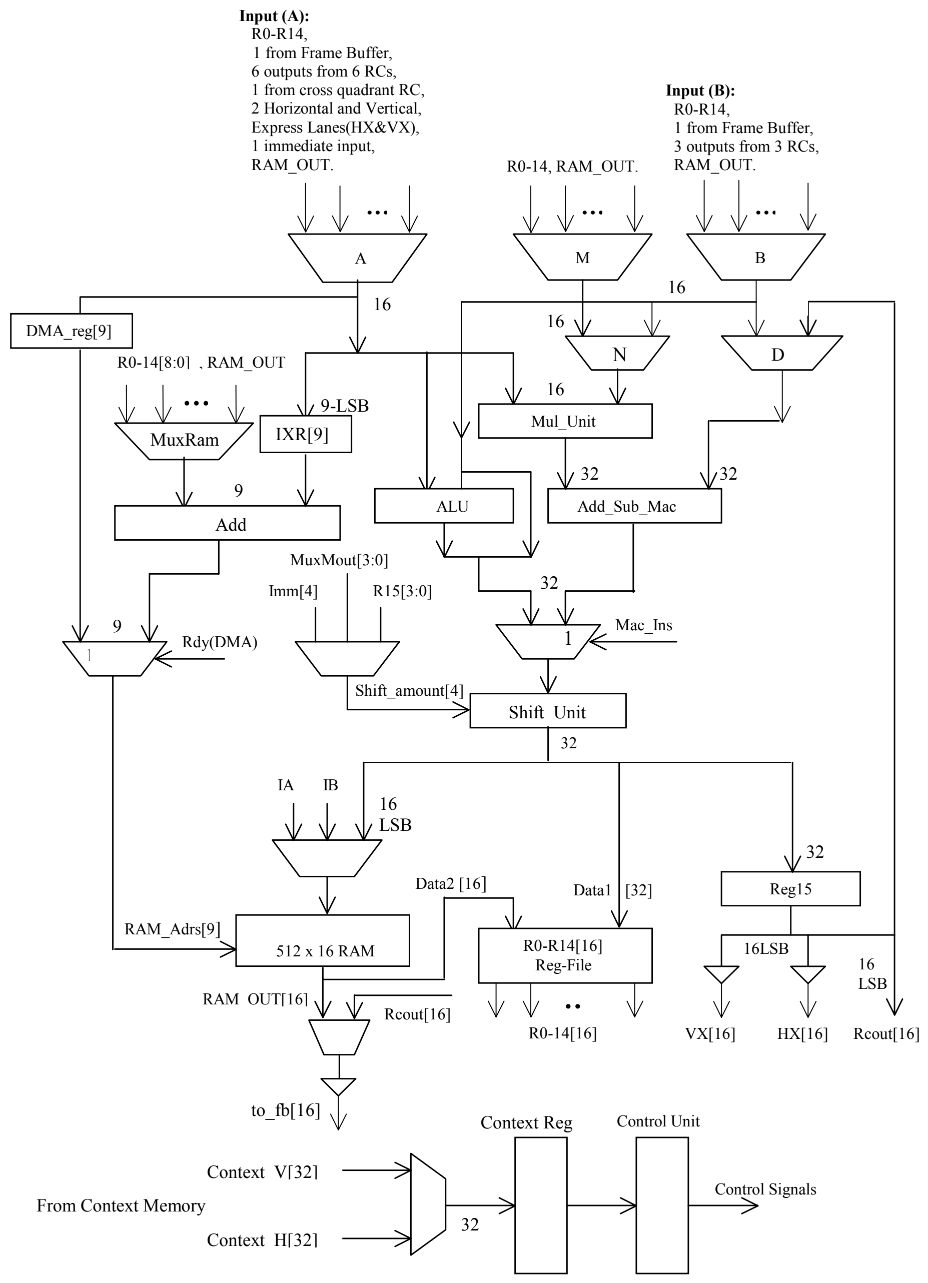


Figure 96. A Block Diagram of RC-M2

In the rest of this section, the architecture of major blocks in an RC is discussed.

5.13.1. Mul_Unit

There are four different types of multiplication to be performed in this module: signed/unsigned, real/complex. Both operands are 16 bits wide for real arithmetic, while complex operands have two 8-bit-wide real and imaginary parts. The complex signed/unsigned multiplication is expressed as follows:

M= A1*B1 - A0*B0 + j(A1*B0 + A0*B1), (4)

where A1 and A0 are the upper (real) and lower (imaginary) bytes, respectively, of the 16-bit signed/unsigned operand A. The multipliers are signed or unsigned if the operation is signed or unsigned respectively. A similar statement is true for operand B.

In order to be able to reuse signed multipliers for unsigned complex operands in equation (4), we have followed the following guidelines:

- Either sign extend or zero extend all A1, A0, B1, and B0 to 9 bits if the operation is signed or unsigned, respectively,
- Assuming that the operators are signed as well use equation (4) to work out the product.

Notice how unsigned complex multiplication is correctly carried out by two signed multipliers: considering that zeros are padded from the left, both operands now look positive to these signed multipliers.

To reuse the same operators for *real* unsigned multiplication, the product of two real unsigned operands may be expressed as follows:

M = A1*B1*2**4 + A0*B0 + (A1*B1 + A0*B1)*2**3. (5)

Notice that any 'n' bit negative number can be expressed as the sum of two 'm' bit wide M and 'k' bit wide K components, that is, N= M + K
where
n = m + k;
M= 'm' MSB of N & 'k' zeros; M is negative
K= 'm' zeros & 'k' LSB of N; K is positive
& is the concatenation operator.
Therefore, it can be shown that equation (5) remains valid for both signed and unsigned *real* multiplications if the following guidelines are adhered to:

1. Either sign extend or zero extend both A1 and B1 to 9 bits if the operation is signed or unsigned respectively.
2. Zero extend both A0 and B0 to 9 bits.
3. Assuming that the operators are signed as well, use equation (5) to work out the product.

Therefore, as shown in Figure 97 to implement all four types of multiplication, only four 9-bit signed multipliers are required, followed by three adders (Add_Sub_Mac) according to (4) and (5). Notice that sign extension for the inputs to the multipliers are not shown in this figure.

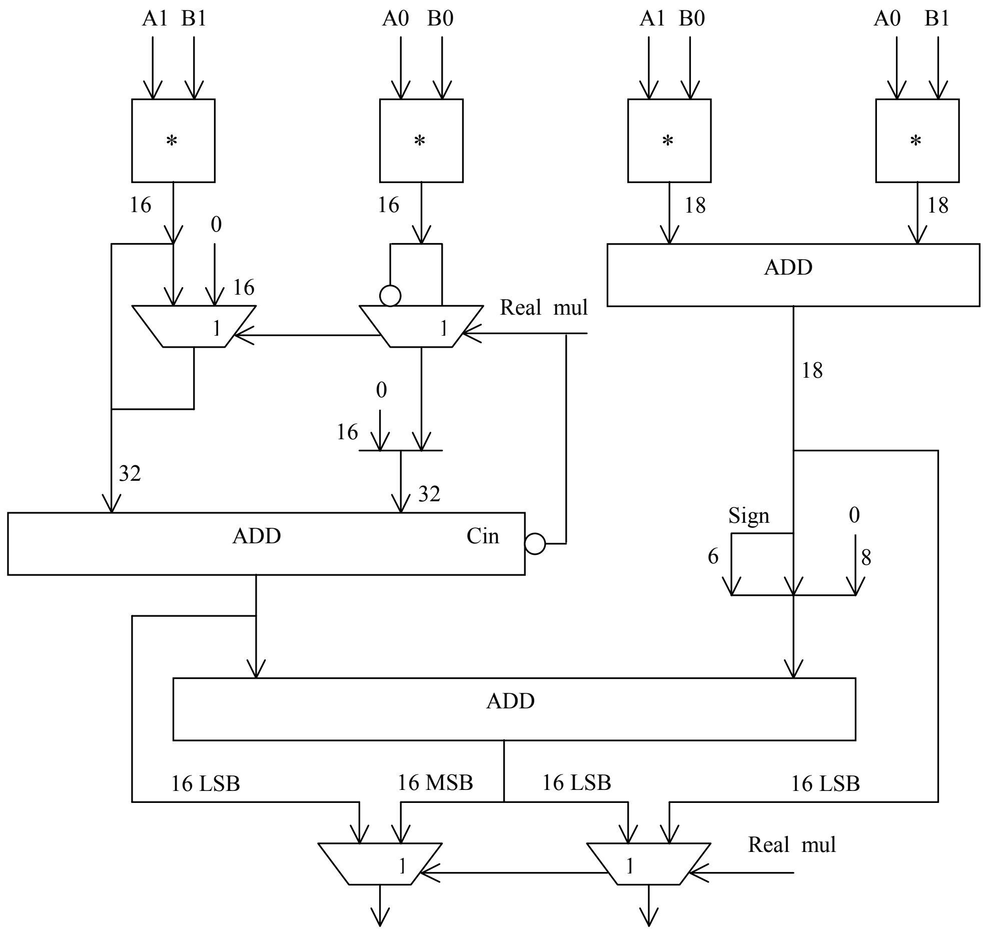


Figure 97. A Block Diagram of Mul_Unit

The structural description of adders in the above Figure 97 results in a more optimized netlist than that from the following behavioral and straightforward description based on equations (4) and (5):

If multiplication is real, then

32-bit product :=
(MUL11(15 : 0) << 16) + (MUL10(15 : 0) << 8) + (MUL01(15 : 0) << 8) + MUL00(15 : 0).

If the multiplication is complex, then
16 lower bits of product (imaginary part) := MUL10(15 : 0) + MUL01(15 : 0);
16 upper bits of product (real part) := MUL11(15 : 0) - MUL00(15 : 0),

where "MULij, i,j = 0,1 is the 18-bit product of the two 9-bit signed multipliers with 9-bit operands Ai and Bj, which have properly been sign or zero extended as explained above.

As shown in Figure 96, the output of Mul_Unit may be added to or subtracted from a third operand coming from the multiplexer MuxD, which may steer different values to Add_Sub_Mac, namely, 1) the content of Reg15, 2) the output of MuxB in two different shapes: zero extended to 32-bit real value, and zero extended to (16 , 16) complex value, and 3 all zero value. The latter is to bypass the Add_Sub_Mac unit, as discussed below.

5.13.2. Add_Sub_Mac Unit

Figure 98 shows a block diagram of Add_Sub_Mac unit. To use the adder circuit in both real and complex operations, two separate 16- and 17-bit adders have been utilized following the conditional inverting stages which are to realize subtraction as well. This figure depicts the exact structural model of the unit, which is more area efficient than what is achieved from behavioral descriptions. The lower adder is 17 bits wide to generate correct carryout bit when the two adders are to be concatenated for real arithmetic. Considering *multiplication only (no accumulation)* operation, the Add_Sub_Mac unit needs a bypass feature. This has been realized by forcing the operand coming from MuxD to zero, while the Add_Sub_Mac is doing an add operation. MuxD is able to supply *all-zero* output, as mentioned in the previous section.

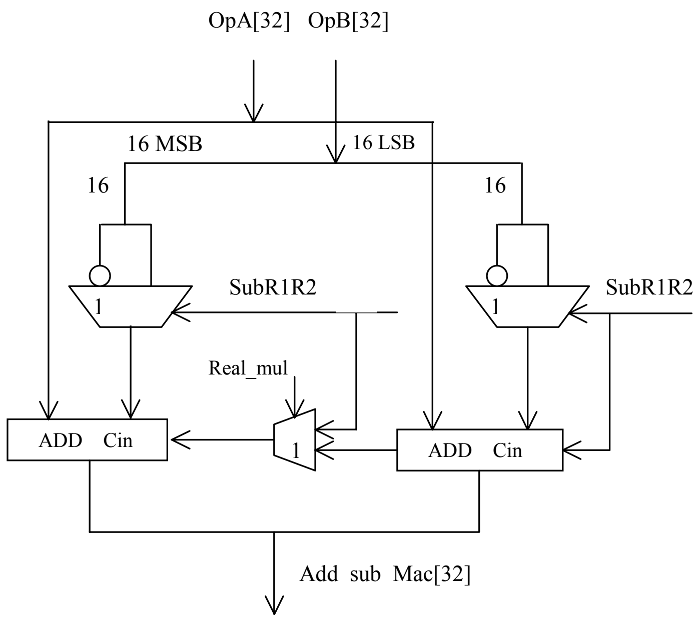


Figure 98. Add_Sub Unit following Mul_Unit

5.13.3. ALU

As illustrated in Figure 99, this unit is composed of three subunits—arithmetic unit, logic unit, and absolute value unit. The arithmetic unit is required to perform real/complex addition/subtraction with carry and no-carry options for real operands, and also work out the absolute value of either OpA or OpA - OpB. On the other hand, basic logic operations and counting leading zeros of both a 16- and-32 bit operands are the functions performed in the logic unit. The third block derives the absolute value of the input.

Arithmetic unit: As shown in Figure 99, and similar to Add_Sub_Mac unit, the adder here is also decomposed, but now into two 9-bit ones, to perform complex arithmetic as well. In real mode (i.e., real =`1`), the two adders are concatenated, while in complex mode, they become two isolated adders for the real and the imaginary parts of the input operands. The upper Mux provides the adders with either operand OpB or its complement in order to realize both addition and subtraction. Notice how *Add with carry* and *Subtract with borrow* have been implemented. For *Add with carry,* the output equation is straightforward:

Out := OpA + OpB + Cflag. (6)

For *Sub with borrow*, however, it is not that clear at first glance:

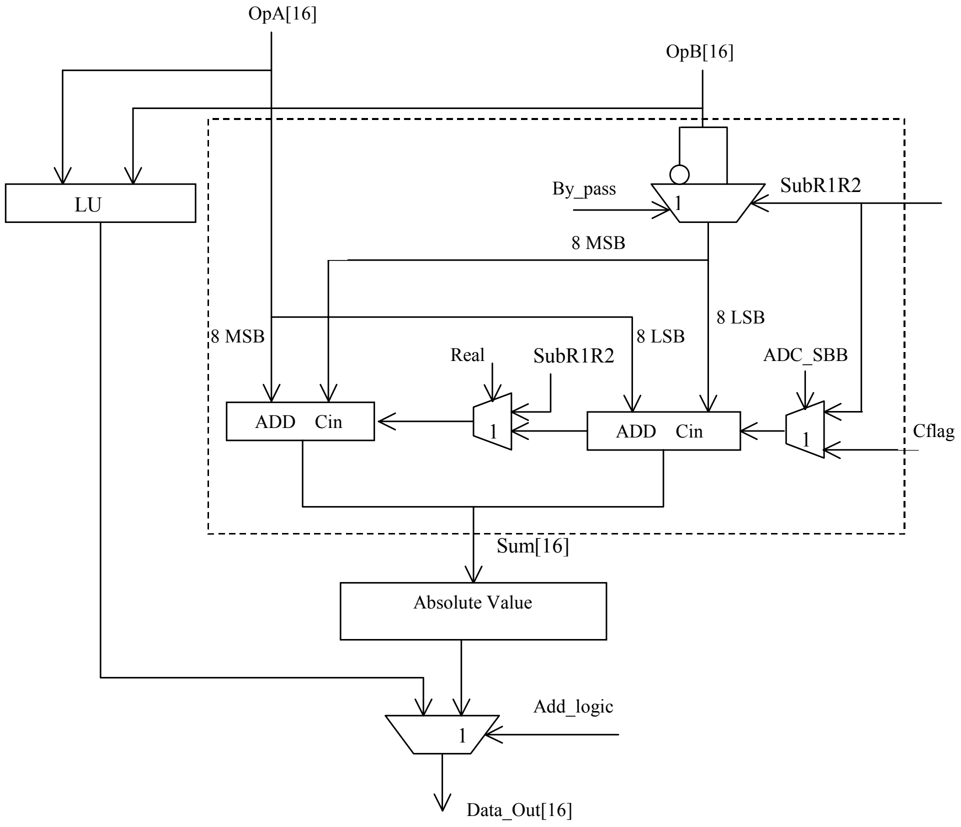


Figure 99. 3 Major Units of ALU: Add (dotted line), LU and Absolute

Out := OpA - OpB –Borrow-in; or
Out := OpA + Not(OpB) + 1 –Borrow-in. (7)

In our implementation, the borrow out of a subtraction is saved in the carry flag but in inverted form, that is, a set-to-one carry flag after a subtraction means no borrow and vice versa. Therefore, equation (7) may be written as follows:
Out := OpA + Not(OpB) + 1 – Not(Cflag); or
Out := OpA + Not(OpB) + Cflag. (8)

Equation (6) and (8) show that for both add with carry and sub with borrow operations, the carry-in bit to the adder has to be Cflag.

For add and sub operations without carry/borrow, the carry-in bit to the adder becomes 0 and 1, respectively. This is also the case with both adders in complex mode.

There is also a by_pass feature here to transfer OpA to the output of the block, which has again been realized by forcing the second operand to zero while an add operation is performed on OpA. This force to zero is carried out in the upper mux.

Logic unit: The major part in the logic unit is a 16/32-bit count leading zeroes/ones (CLZO) or the well-known priority encoder *(PE)*. We have decomposed it into two other blocks, namely, a priority detector (PD) followed by an encoder. By an n bit priority detector we mean a function with n inputs and n outputs in which at most one output at a time is asserted that corresponds to the input which has the highest priority among the set of all asserted inputs. A ripple PD using repetitive identical cells has been utilized in our design with no impact on the worst path. Figure 100 shows cell(i) of this circuit, where g(i) and g(i-1) are two grant signals from the higher priority block and to the lower priority block, respectively.

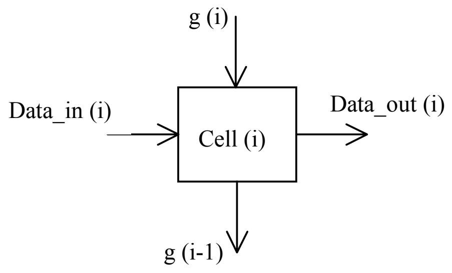


Figure 100. The ith Cell of the Priority Detector

Assuming active high signals the logic equations for the two outputs are derived as follows:

```
data_out(i) := data_in(i) if g(i) is asserted else
                '0';
g(i-1) := '0' if data_in is asserted else
          g(i);
```

To convert this 32-bit chain to a 16-bit one, we have simply de-asserted the MSB of the lower half of the 32-bit input operand.

The following functions located in package *pack* show the behavioral description of theses two blocks:

```
function pri (datain: Std_logic_vector) return Std_logic_vector is
variable g, dataout:   std_logic_vector (31 downto 0);

begin
   priority:
     for i in 31 downto 0 loop
      if i=31 then
        dataout(i) := datain(i);
        g(i-1) :=not datain(i);
     elsif (i<=30 and i>=1) then
           dataout(i) := datain(i) and g(i);
           g(i-1) := not datain(i) and g(i);
        elsif (i=0) then
             dataout(i) := datain(i) and g(i);
      end if;
   end loop;
   return dataout;
 end pri;
```

```
function encoder (datain: Std_logic_vector) return Std_logic_vector is
 variable dataout:   std_logic_vector (5 downto 0);

  begin
   encode:
     dataout := "000000";
     for i in 31 downto 0 loop
       if datain(i) = '1' then
          return dataout;
        else
          dataout:= dataout + '1';
        end if;
     end loop;
   return dataout;
 end encoder;
```

5.13.4. Shift unit

Shift operations are performed on either 16- or 32-bit operands, each of which may be carried out either through a shift-only instruction or within a complex instruction, such as mul, add and shift. The number of shifts is specified with an immediate operand for the embedded shift operations, while shift-only instructions use one of the general-purpose registers to identify the intended value. The embedded 32-bit shift, which is accompanied by either a mul or a mac instruction, is either signed or unsigned if the corresponding mul operation is signed or unsigned, respectively, but the 32-bit shift-only instruction is always signed. While 32-bit operands may be shifted only right or left, there are two more choices for 16-bit shift-only instructions, namely rotate left and right through the carry bit. Figure 101 illustrates a block diagram of the shift unit, and its major subunit shift16. Notice that block Shift32 uses the IEEE built in SHR and SHL Functions.

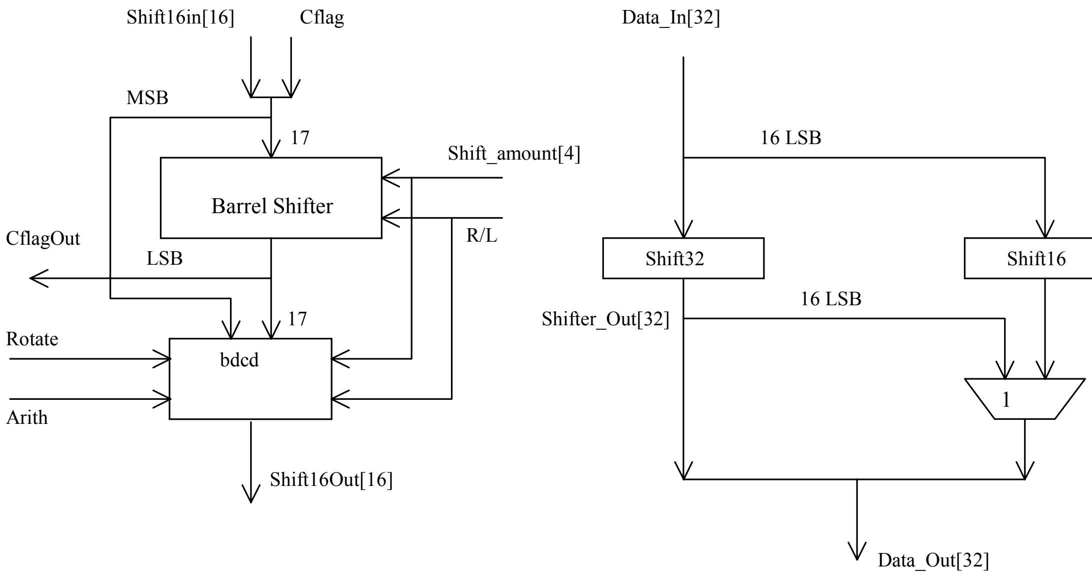


Figure 101. Block Diagrams of Shift16 (left) and Shift_Unit

Shift32 in Shift_unit is implemented using the IEEE built-in SHR and SHL functions, while shift16 is composed of two subunits, as shown in Figure 101. The barrel shifter rotates the 17-bit input vector (the 16-bit operand concatenated to the carry flag) by the specified number of bits, and then in the following block, bdcd (barrel_decoder) the rotated vector either remains unchanged or is converted to a shifted version (either arithmetic or logical) of the input. Notice that the carry flag also enters the barrel shifter to participate in a possible rotation. A new carry bit is then generated according to the value of the carry bit in the rotated vector.

5.13.5. Local RAM:

A 512 by 16-bit synchronous RAM generated by the Artisan HS-RAM generator has been utilized as an internal memory for each RC. This is a synchronous RAM, meaning that both write and read operations are carried out by a positive edge of the clock signal. This clock edge requirement brings up a potential problem, as now the data read out from the RAM becomes valid only at the end of the clock cycle during which the address has been valid. Therefore, we need to wait until the end of next clock cycle to have the data written in a register. Consider the following piece of code:

```
#n     LD R3, MEM(R12);
#n+1 ADD R4, R3, R2.
```

At the end of cycle number n, the memory location pointed by R12 is read out but not transferred to R3 yet; therefore, the ADD instruction will take an incorrect value from the register if no provision is made to remove this data hazard. The problem may be solved in a couple of ways: 1) implement the register file in a way that it is first written in and then read out, say, by using negative edges of the clock signal to trigger the RAM if the register file is triggered by positive edges, and 2) use the well-known bypassing or forwarding technique widely utilized in pipelined architectures. While using different clock edges is very straightforward, it has a major drawback as well: assuming a 50 percent duty cycle, both read and write operations are performed in the middle of a clock cycle, unlike an asynchronous RAM which can be read out and written in early and late, respectively, in a clock cycle, leaving enough room for data manipulation to be carried out during the rest of the cycle. To make most of the cycle available for data manipulation in a write operation, we have used data forwarding to solve the above problem. This has facilitated the implementation of arithmetic operations with a destination in the RAM (CISC-type store instructions) and hence greatly enhanced the bit utilization of store instruction.

As illustrated in Figure 96 the RAM can be provided with an address by a couple of sources, namely a DMA register, and any register in the register file added or not added to an index register. The two special-purpose registers, DMA and index registers, can be loaded from any general-purpose register through the two corresponding instructions. A general-purpose register together with the index register realizes indexed mode (load only) and register indirect mode addressing features to access the memory. The index register is incremented after each indexed mode addressing to expedite array processing. The DMA register will be more discussed in the DMA section.

Bit utilization in load instructions has also been enhanced, but now by packing multiple loads (up to 3) in one instruction. The first load is carried out as usual, and the destination becomes valid at the beginning of the next clock cycle; however, the two pending loads have to wait for the next two cycles to be executed in parallel with the next two sequential instructions. This has made a dual port register file inevitable, as now up to two writes at a time may be requested.

5.13.6. Register file

There are 15 by 16-bit registers in the dual input port register file. The 16th register, the 32-bit Reg15, is although physically separate, logically addressable similar to the rest of the registers in some of the instructions. The result of all data processing enters the register file through port 1, while port 2 is used only by the local RAM. Notice that the RAM updates its output with every read and write operation. Furthermore, this output represents a valid content of a general-purpose register when a load instruction is executed and remains valid until either this value is written back to the specified register, or this register is updated through another instruction via port 1. In our implementation, the valid output is not saved in the destination register unless the memory output has to be updated by another load, a pending load, or a store instruction.

Support for DMA

The RC supports a cycle stealing direct data transfer between its local RAM and an external data RAM, FB, which is shared by all RCs. The special-purpose DMA register inside each RC is to be initialized with the starting address before the TinyRISC initiates a direct transfer. This register may be loaded from any of the registers in the register file by the corresponding load instruction, and is incremented after each direct data transfer to get updated for the next transfer cycle. Since the local RAM is single port, an arbitration mechanism has to be devised when two accesses to this memory are requested at the same time by the DMA controller and RC itself. In our design, RC has priority over the DMA controller, as this relieves RC of the overhead caused by the direct data transfer, resulting in a higher performance. In case the RCs instruction execution becomes DMA data dependent, it is the responsibility of the TinyRISC to make RC wait until the data transfer is completed.

Figure 102 shows the external data and context buses for the RC array. All of these buses have been explained throughout the course of this section. The intercell connections of the array and also the express buses are identical to Morpho M1, so they have not been addressed in this section.

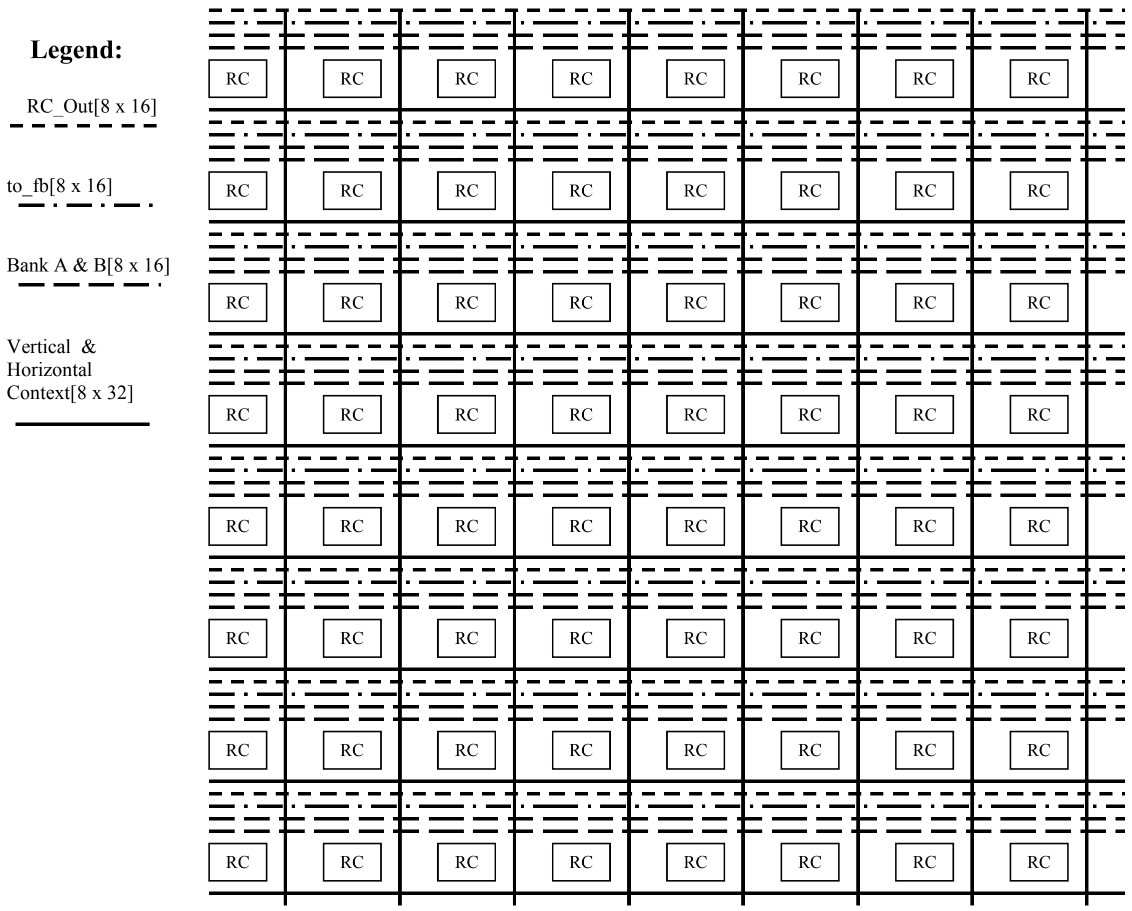


Figure 102. The RC Array External Context and Data Buses

6. Mapping of the Digital Video Broadcasting--Terrestrial (DVB-T) Application

Today, a new digital-to-software revolution in information technology is taking place after the analog-to-digital revolution 2 decades ago [11]. Software Defined Radio is one among the most famous revolutions. Compared with hardware solutions, software solutions can provide great flexibility in a much shorter market time. DVB-T, the European standard (by ETSI) on digital TV radio [12], is one of the highest speed wireless communications systems with the highest technical sophistication. It adopts the coded orthogonal frequency division modulation (COFDM) scheme. The OFDM symbol length is as large as 2 K/8 K with the highest data rate of 31.67 Mbps. The complicated base-band receiver is composed of an inner receiver (OFDM demodulator), an outer receiver (64-state Viterbi decoder and Reed-Solomon decoder), and a source decoder (MPEG-2 decoder). How to implement this computational-intensive receiver onto a single programmable device is an extremely challenging task. A multi-chip quasi-software solution exists [13,14], requiring no less than 10 chips in year 1999, while still resorting to FPGAs and ASICs. This section proposes a pure-software single-chip solution based on a reconfigurable architecture called MorphoSys. First, DVB-T receiver system design procedure is described and then a fixed-point simulation result conforming to the protocol is achieved. Second, we introduce the architecture and programming model of the MorphoSys processor. Last, we algorithm mapping procedure and the results will demonstrate the architecture having orders of magnitude higher computation capability than other commercial DSP chips.

6.1. DVB-T System Design

6.1.1. DVB-T Receiver

The whole DVB-T receiver consists of an RF front-end part and a base-band DSP part. In this section, we will focus on the design and implementation of the DSP. The DSP part receives digital base-band input and completes tasks such as OFDM demodulation, channel decoding, and MPEG-2 decoding. The corresponding units are called inner receiver, outer receiver, and source decoder. The diagram is shown in Figure 103.

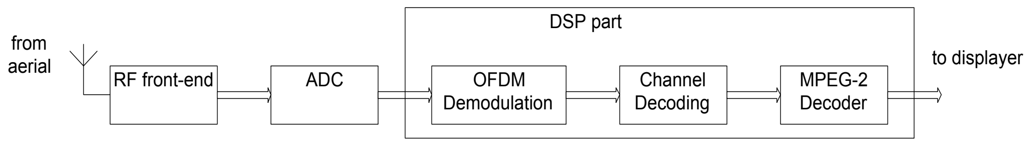


Figure 103. Diagram of DVB-T Receiver

6.1.2. Floating-point System Design and Simulation

We choose a computation-intensive system configuration of DVB-T as our design target: single frequency network (SFN), 8 MHz channels, 8-K mode, 64-QAM, quarter-guard interval, and half convolutional code rate, which has “a good compromise between bandwidth efficiency and robustness” [15]. The software solution has enough flexibility to accomplish other configurations, too. There are three stages in the floating-point system design: 1) algorithm selection, 2) channel simulator design, and 3) parameter optimization.

First is the algorithm selection. There are two modes of the receiver: acquisition mode and tracking mode. As soon as the receiver is turned on, it goes into acquisition mode. At this mode, the only active function is the inner receiver with 10 blocks: coarse timing acquisition, fractional carrier frequency acquisition, carrier frequency correction, FFT, integer carrier frequency acquisition, frequency tracking, timing and sampling frequency correction, frame synchronization, channel estimation, and fine timing acquisition block. The task flowchart is shown in Figure 104.

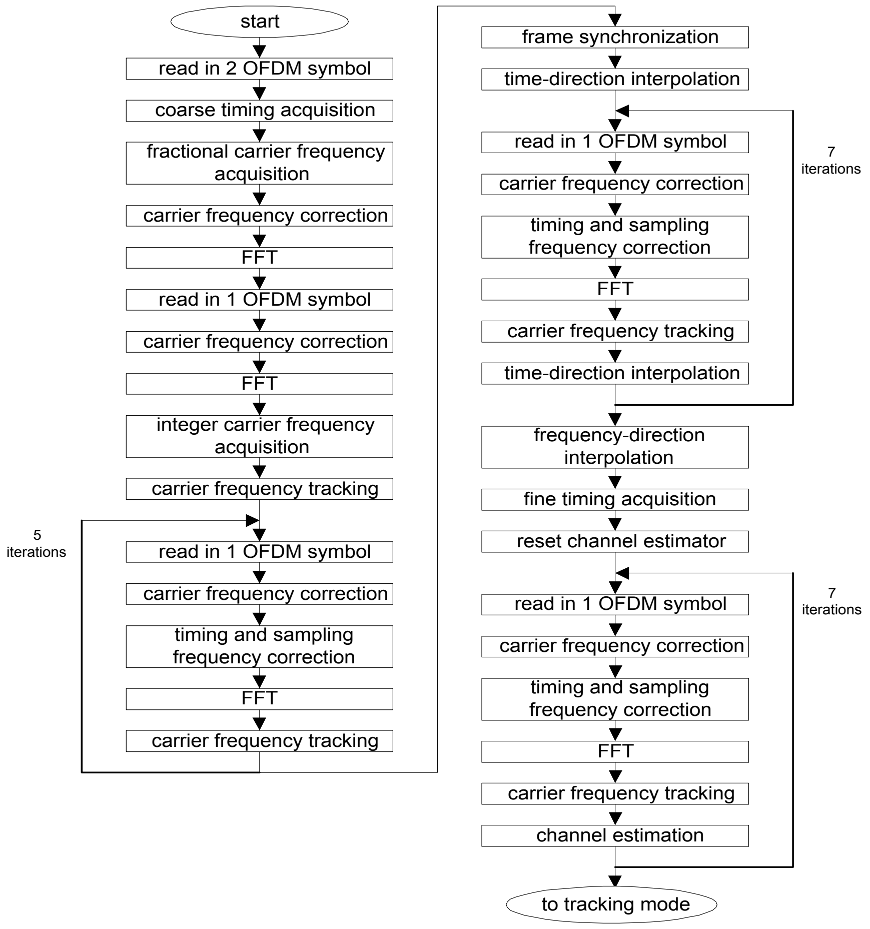

start
read in 2 OFDM symbol
coarse timing acquisition
fractional carrier frequency acquisition
carrier frequency correction
FFT
read in 1 OFDM symbol
carrier frequency correction
FFT
integer carrier frequency acquisition
carrier frequency tracking
5 iterations
read in 1 OFDM symbol
carrier frequency correction
timing and sampling frequency correction
FFT
carrier frequency tracking
frame synchronization
time-direction interpolation
7 iterations
read in 1 OFDM symbol
carrier frequency correction
timing and sampling frequency correction
FFT
carrier frequency tracking
time-direction interpolation
frequency-direction interpolation
fine timing acquisition
reset channel estimator
7 iterations
read in 1 OFDM symbol
carrier frequency correction
timing and sampling frequency correction
FFT
carrier frequency tracking
channel estimation
to tracking mode

Figure 104. Task Flowchart in Acquisition Mode

After timing and frequency acquisition is achieved in 22 OFDM symbols (24.6 ms), the receiver goes into tracking mode. In tracking mode, the inner receiver includes 5 blocks: offsets correction, FFT (8 K), frequency tracking, channel estimation and equalization, and fine timing synchronization block. The outer receiver contains seven blocks: de-mapping, inner deinterleaving, inner decoding (Viterbi decoding), outer deinterleaving, outer decoding (Reed-Solomon decoding), de-scrambling, and transmission parameter signaling (TPS) block. Most of the algorithms mentioned above are similar with those in references [15 through 21], etc. Source decoder consists of variable length decoding (VLD), de-quantization, inverse discrete cosine transforms (IDCT), and motion compensation block. The task flowchart is shown in Figure 105.

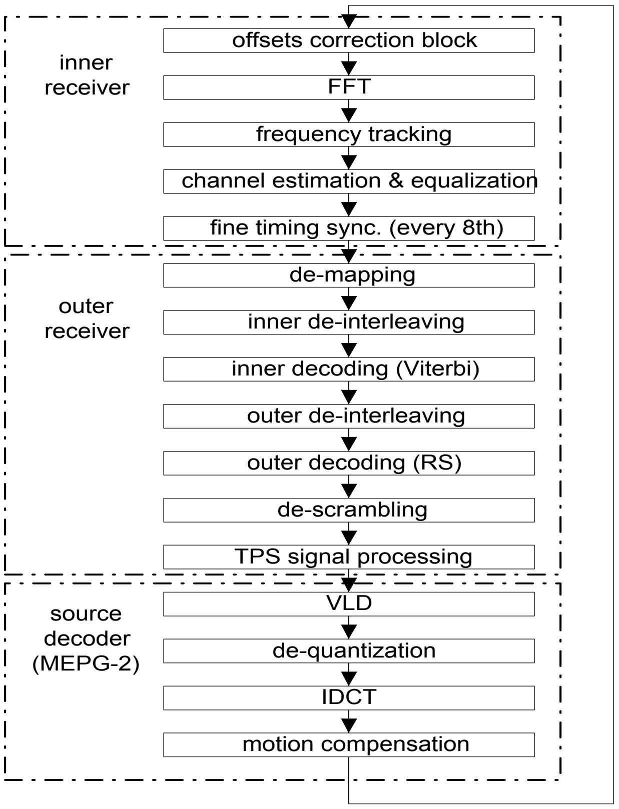


Figure 105. Task Flowchart in Tracking Mode

6.1.3. Channel Simulator

An accurate and fast channel simulator is very important for wireless communication system simulation. As a widely used accurate modal, a multipath Rayleigh fading, wide-sense-stationary uncorrelated-scattering (WSSUS) channel model is adopted [22,23]. Then a fast WSSUS channel simulator is written based on the one in reference [24]. As the most difficult scenario, “typical case for hilly terrain (HTx)” specified in 3GPP standard [25] is used. SFN is modeled as HT1, followed by HT2 with a delay of half-guard interval.

6.1.4. Parameter Optimization (SNR Budgeting)

Since the critical path lies in the tracking mode, parameter optimization focuses on the tracking mode. The objective of parameter optimization is to minimize the computation requirement while still meeting the Quasi-Error-Free (QEF) requirement (BER<2*10-4) after Viterbi. Thus, SNR budgeting starts from Viterbi decoder backward to the first block, the offsets correction block. The following figures show the optimized results based on careful simulation.
Viterbi: Since the soft decision Viterbi is 3.5 dB better than hard decision Viterbi (Figure 106), the soft decision algorithm is adopted and the input SNR needed is 13.2 dB.
Channel estimation and equalization block: A robust channel estimator using Weiner filter [19,20] with 3 sets of coefficients (12 each) is optimized. The estimator gain can be 4 to 5 dB.

Frequency tracking block: a 25-tap 4-fold interpolator followed by a linear interpolator is used to compensate sampling frequency offset.

There is no adjustable parameter in the outer receiver and source decoder. Simulation shows that in order to reach QEF, the input channel SNR is 20 dB for HT1 channel (Figure 107), and 25 dB for SFN (Figure 108). Figure 109 shows the receiver can also work in mobile condition even in the highly frequency-selective fading channel.

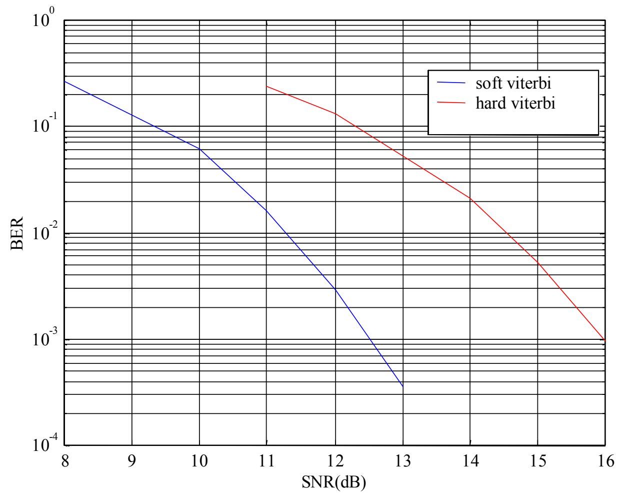


Figure 106. Viterbi Performance Comparison

6.1.5. Fixed-point System Design and Simulation (Signal Quantization-Noise Ratio (SQNR) Budgeting)

Fixed-point system design on a digital signal processor is a process of tradeoffs between dynamic range and quantization error. Careful optimization of the fixed-point system on the 16/32-bit MorphoSys architecture [26,27] results in a negligible performance degradation (see Figures 107 and 108). Operations such as cosine, sine, division, Galois

Field multiplication, are efficiently implemented using LUTs or small LUTs followed by linear interpolators.

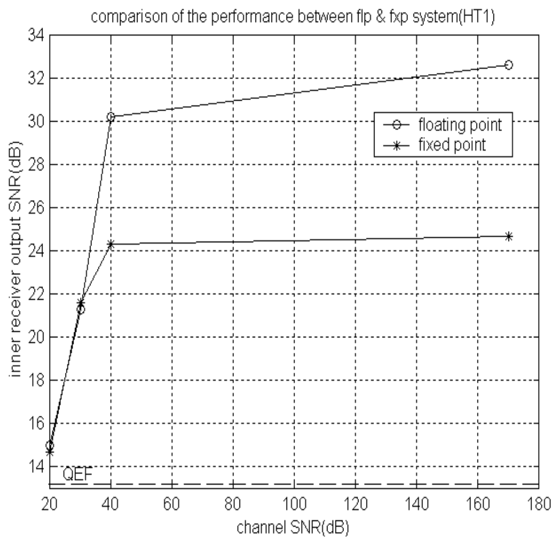


Figure 107. Inner Receiver Performance (HT1)

6.2. MorphoSys Architecture Revisited for DVB-T Application

MorphoSys is a domain-specific reconfigurable architecture targeting on data-parallel and computation-intensive applications, such as 3G/4G wireless communications, multimedia processing, high-quality graphics, etc. [26,27]. It can achieve high performance near ASICs while having the flexibility of DSPs. Moreover, its power consumption is lower than DSPs and FPGAs.

From a programming standpoint, the programming model is very simple. TinyRISC takes charge of the whole data control flow (DCF); RC array and FB are only triggered by TinyRISC and execute on their own configurations continuously for a given number of cycles specified by TinyRISC. Since the DCF is predefined, there is no data, control, or structure hazard in both TinyRISC and RC array with the help of static scheduling. Below is a piece of code for vector dot-product calculation.

```
TRISC assembly code:
loop 3, 100  #100 iterations, 3 cycles each
rcex all, -, RC-dot-product, 3  #issue contexts to RCs

Contexts for RC array:
RC-dot-product:
all ldid +H R1 R5 > R2  #R2=mem(R1);R1=R1+R5
all ldid +H R3 R5 > R4  #R4=mem(R3);R3=R3+R5
all muladd R2 R4 MACREG > MACREG
                        #MACREG=MACREG+R2*R4
```

Based on the design of the first version of the MorphoSys architecture called M1 (see Figure 110) and numerous mapping experiences thereafter, many modifications were made to design the improved version of the chip called M2 that is much faster, more memory efficient, and less power dissipating. Following is the description of these modifications.

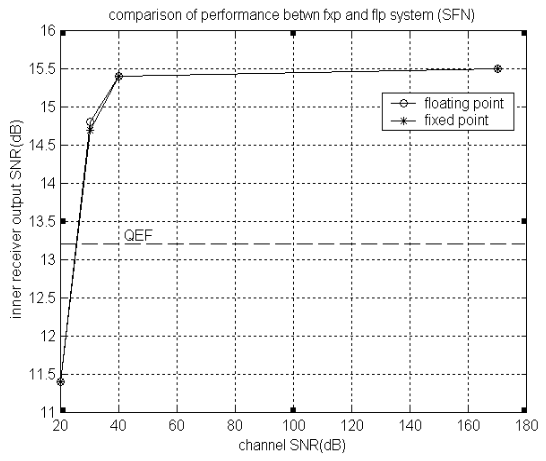


Figure 108. Inner Receiver Performance (SFN)

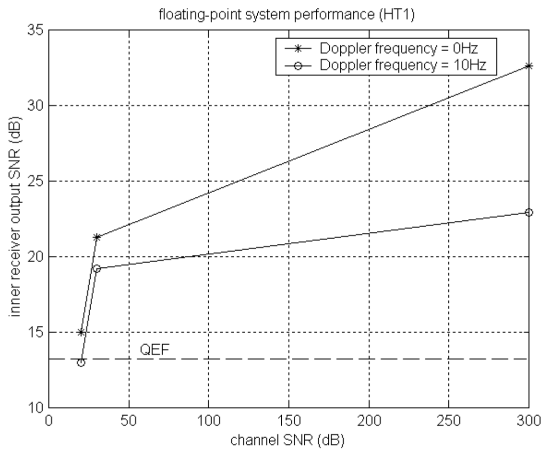


Figure 109. Inner Receiver Performance (Mobile Channel)

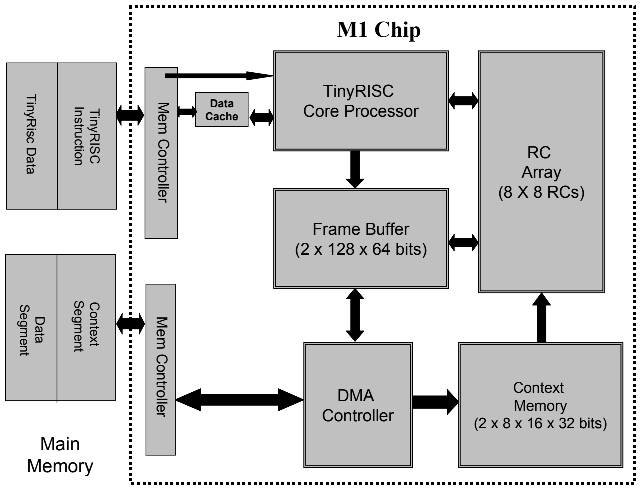


Figure 110. MorphoSys Architectural Model (M1)

6.2.1. TinyRISC modifications

Decoupled Execution of TinyRISC and RC Array

In M1, TinyRISC has to issue one instruction to specify the Context Memory address for each cycle when RC array is executing context. Through application mapping, we find out that most of the contexts are executed continuously for a certain number of cycles, varying from 2 to 150. Based on this observation, modifications have been made to void TinyRISC issuing an instruction to the RC array each cycle when the RC array is executing contexts continuously, which happens in most cases (refer to Figure 111). TinyRISC specifies the starting address of context memory and the number of contexts that need to be executed. After that, TinyRISC can continue to do other tasks unrelated to the RC array, and the RC array will increase the context memory address automatically. Besides, in order to support the decoupling, TinyRISC has an instruction to access FB or inner-RC memory in a single cycle. The hardware cost is a timer and a context memory address incrementor, and 16-bit buses between TinyRISC, FB and inner-RC memory.

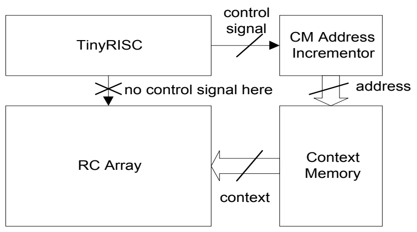


Figure 111. Decoupled Execution of TinyRISC and RC Array

There are many benefits for this decoupled execution of TinyRISC and RC array. Firstly, this will make possible the overlap of execution of parallel codes on RC array and sequential codes on TinyRISC. In large systems, there are often some irregular computations and/or naturally sequential tasks, e.g., zigzag scan in MPEG-2 encoder/decoder [28], trace back in Viterbi [29], etc., which can hardly be parallelized onto RC array. At the same time, execution of these tasks on TinyRISC sequentially may take a long time. Overlapping them with other parallel tasks on RC array by high-level pipelining will greatly reduce the total execution time. For example, Viterbi decoder performance is improved at 31 percent. Secondly, this decoupling can reduce the TinyRISC code size by a great amount, sometimes in orders of magnitude. For example, the number of TinyRISC instructions related to RC array drops from 150 to 1! Furthermore, it also saves power by reducing issued instructions.

Subroutine Support

For a large system, there must be many modules, which should be designed and tested individually. To support this modular programming capability, a low-overhead (one cycle) subroutine call instruction is added to TinyRISC instruction set (see Figure 112). The hardware cost is a PC stack in TinyRISC. Due to the specific domain and non-necessity of recursive algorithm support, the stack depth of 10 has been proven to be adequate.

Zero-overhead Loop

In communications and DSP applications, there are often many loops with a large number of iterations. In some cases, each iteration contains only a few instructions. For example, for data movement, there's often only one instruction in each iteration. However, loop overhead is three cycles. A zero-overhead loop instruction for TinyRISC can remove this costly overhead and boost the performance by 4 times. The hardware cost is only a timer in TinyRISC (see Figure 113).

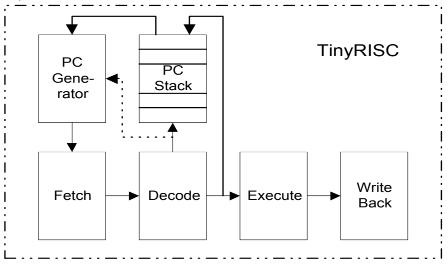


Figure 112. Subroutine Support

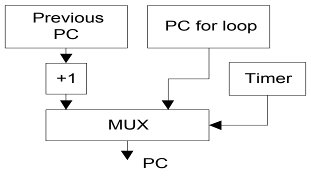


Figure 113. PC Generator for Zero-overhead Loop

Other Modifications

The instruction set is also modified according to above innovations as well as others that follow. New instructions, such as multiplication, sleep, etc., give the processor a better tradeoff between area, speed, and power. Interrupt unit is also added to deal with interrupts from timer, DMAs, and off-chip requests. Thus, all these modifications give TinyRISC more computation power, less CDF control overhead, smaller code size, and lower power dissipation with smaller hardware cost.

6.2.2. RC array modifications

Temporal Granularity: key of High-speed Architecture

As the operation profiling of the whole system indicates, MAC operations occupy only less than 10 percent of the total cycle number, while MAC unit contributes two-thirds of the critical-path delay. This causes a great mismatch between the temporal granularity of the architecture (RC array) and that of the algorithm. From this observation comes the solution: pipeline the MAC unit in three stages and shorten the critical path. Since the predefined data dependency can be scheduled statically, no hardware hazard detection unit is needed. Circuit simulation shows that the critical path is made up of the following parts with conservatively estimated delay (see Figure 114):

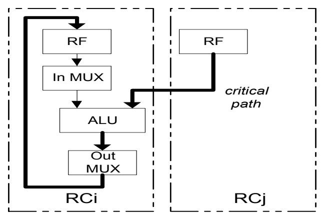


Figure 114. Critical Path

Register setup time and hold time, including clock skew: 0.2 ns
Inter-RC connections: 0.5~1 ns
Input mux for ALU (overlapped with inter-RC connections): 0.3 ns
ALU: 1.2 ns
Output mux for ALU: 0.3 ns

After optimization, Synopsis postsynthesis simulation using 0.13-μm technology typical library shows that the critical-path delay excluding inter-RC-connection delay is 1.3 ns. Thus the total critical-path delay turns out to be 1.8 to 2.3 ns (556 to 435 MHz). Compared with the former architecture of 8 ns delay (125 MHz), the performance without MAC operations is boosted up by 4 times or so. For applications with n percentage of MAC operations, the performance will drop by less than 2*n percent, for some continuous MAC operations can be pipelined to increase throughput. For DVB-T receiver, n is less than 10. Therefore, the speed of M2 is far beyond what the DVB-T receiver needs as 328 MHz (shown in the mapping section).

Inner-RC Memory

Inner-RC memory (4 Kb SRAM in each RC) is the main storage media in the architecture. According to the SIMD nature of the application domain, e.g., Finite Impulse Response (FIR) filter, data assessments are often locally within each RC. Thus, global data movement can be minimized by using inner-RC memory.

Enhanced Inter-RC Connection

For a large system, there is always some residual data movement between RCs even after careful minimization, such as butterflies in FFT and DCT, metrics movement in Viterbi, and rearrangement of edge data for FIR filters, etc. It will be very time consuming if it has to be done sequentially. Fortunately, these are regular data movement and can be dealt with in parallel with enhanced inter-RC connection, which can access all registers in the same column and same row [30].

RC Active Mask

It is not always the case that all of the RCs will be busy. This happens in many blocks, such as the Reed-Solomon decoder, the outer deinterleaver, and so on. Furthermore, sometimes it is even mandatory to idle some RCs, e.g., on metrics movement in Viterbi decoding. Two RC active masks were added to solve this problem, one for row and another for column. Thus, when contexts are issued, only the columns (/rows) specified by the column (/row) mask are active. This will also save the code size of contexts.

Context Set Modification

Based on the profiling of mapping results on communications and DSP systems, the context set is optimized to match the domain requirements tighter while still persisting enough generality. For example, immediate-valued-operand contexts are added to reduce both execution time and code size; a conditional OR instruction is added to meet the requirement of many bit-wise algorithms, e.g., writing trace-back information in Viterbi, and so on. To sum up, these modifications result in an RC array with maximized computation speed, reduced data movement overhead, smaller code size, and negligible area and power increment per cycle.

6.2.3. FB Modifications

Fast Connection Between FB and RC Array

The FB is constructed in 64 banks corresponding to 64 RCs, so that data communication will be as fast as 128 bytes per cycle. The FB is a two-port memory, which can overlap the time when data are buffered from external memory to inner-RC memory through FB. The FB assess time is less than 2 ns. Since it is often needed to copy one LUT into 64 RCs, the FB also has the capability to broadcast one datum to 64 RCs.

Fast Continuous Data Movement within FB

In large systems, data output from one code block often needs to be rearranged before it can be used as input to the next code block. This data rearrangement is a bottleneck of many parallel architectures. For example, the simple zero-padding operation of a vector in SIMD architecture will become very difficult. However, most of these rearrangements are either continuous data movement or movement with parallelism on RC arrays, or a combination of both types. In DVB-T receiver, except pseudorandom deinterleaver, all others fall into these three categories. For movement with parallelism on RC arrays, an inter-RC connection can deal with it well. For continuous data movement, a special mechanism in the FB is designed to accelerate the speed up to 32 bytes per cycle. Figure 115 gives out the block diagram. The hardware cost is 32 16-bit shifters and more sense amplifiers in the SRAM array.

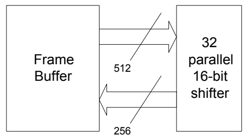


Figure 115. Fast Continuous Data Movement within FB

Shift Between Two Addressing Order

Due to the bank structure of the FB, it can be regarded as a two-dimensional memory cell (16-bit) array. There are two ways to order the address: one is addressing crossing banks (address = index_of_bank+64*index_within_bank); another is addressing along banks (address = index_within_bank+1K*index_of_bank). See Figure 116. Due to data locality, most of time data are continuous in the sense of the second order, while in some cases they will change to the first order, e.g., the order of data after FFT will change from the second order to the first order. Hence, a special 8-K-bit block of FB is designed to be capable of shifting data between these two orders. The hardware innovation is to route two sets of bit-lines and word-lines in orthogonal directions. The memory cell is a two-port cell as the same in the rest part of FB (see Figure 117).

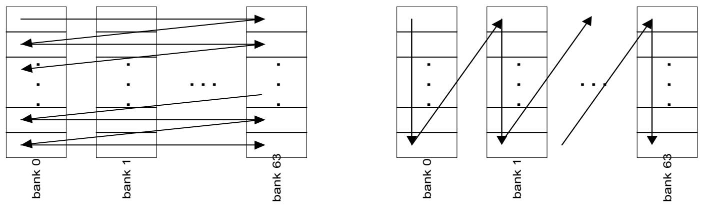


Figure 116. Two Addressing Order of FB

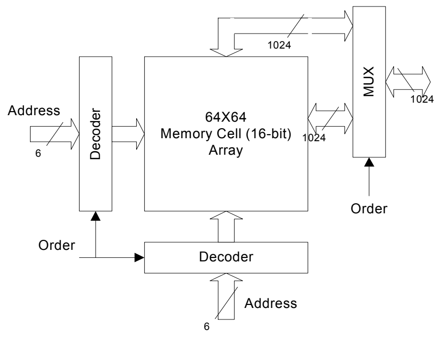


Figure 117. A Block Diagram of Reordering Buffer

Direct Memory Movement (DMM)

Besides regular types of movement above, there still may be some naturally irregular ones, e.g., pseudorandom deinterleaver, whose very purpose is to disorder the data in a pseudorandom manner. It is possible to do this sequentially by TinyRISC. However, the cycle number needed is costly—about 64 K. Thus, a DMM unit is introduced to remove the overhead and finally reduce the cycle number to 24 K. What the DMM does is to read source data and destination address in one cycle and write the data to destination address with a given bias in another cycle (Figure 118). An additional benefit of the DMM unit is to free TinyRISC as well.

With the new FB, numerous different types of data movement in the huge DVB-T and MEPG-2 system can be accelerated 8 to 64 times faster than sequential movement. Even the notoriously difficult pseudorandom deinterleaver can be accelerated and overlapped with other computations.

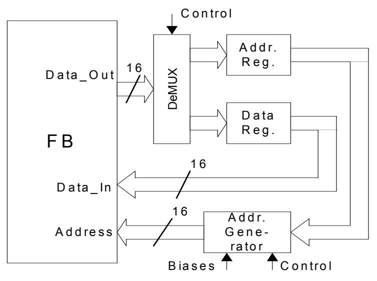


Figure 118. A Block Diagram of DMM

6.3. Mapping DVB-T onto MorphoSys Architecture

6.3.1. Mapping Methodology

Since the critical path of the system lies only in tracking mode, we only consider mapping of tracking mode. Thanks to the simple programming model the architecture adopts, the mapping methodology is not much more complex than that used for general-purpose processors. The whole CDF is controlled by TinyRISC. The RC array is simply executing block code without branches, behaving like a multicycle function unit. The FB can be viewed as another function unit. TinyRISC code is written in assembly language, which has some high-level language features such as loop, subroutine, etc., to ease the programming. Multifile modular programming is also supported.

The key of a good mapping is to extract maximum parallelism from algorithms. There are two types of parallelism suited for the architecture:

SIMD: use the parallelism of the RC array. It is always appreciated that all RCs are executing the same context in a given time. In DVB-T, most of the time this parallelism is extractable, e.g., FIR filters. Regular data movement between RCs can also be implemented in parallel, e.g., FFT and Viterbi.

MPMD: use the decoupling property of TinyRISC and the RC array. Sequential codes on TinyRISC and parallel codes on the RC array can be overlapped, e.g., trace back in Viterbi is accomplished in TinyRISC, while at the same time the RC array is doing ACS operations of the next segment of data. Besides, DMM also provides another MPMD between FB and other components.

Memory allocation is also simplified by using a single address space for all of the external memory and internal memory including FB, inner-RC memory, and context memory. Contexts can only address inner-RC memory within one RC. The data movement problem, which is essential for data-intensive parallel computation system, is also well addressed by enhanced FB and inter-RC connection.

6.3.2. Mapping Results

Buffer Address Calculation

This small block calculates the starting address of the new OFDM symbol based on the timing and frequency offset and then loads data from the FB to the RC array. This is done by purely sequential TinyRISC code.

Carrier Frequency Correction

This block corrects the phase error due to carrier frequency offset. Since all of the 8-K samples in one OFDM symbol can be corrected independently, the full 64-fold SIMD parallelism can be easily extracted by simply distributing these 8 K samples into 64 RCs, with 128 samples in each RC. Sine and cosine functions are implemented by a small 16-entrée LUT) followed by a linear interpolator (see Figure 119).

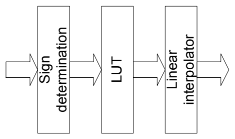


Figure 119. LUT Implementation

Timing and Sampling Frequency Offset Correction

This block compensates the timing and sampling frequency offset by re-sampling the received signal on the new corrected time spots. The re-sampler is a four-fold interpolator followed by a linear interpolator. Since the size of the output of the four-fold interpolator is 4 times larger than the input 8-K OFDM symbol (32 KB), i.e., totally 128 KB, the input symbol is truncated into four segments to reduce the temporary storage to one-fourth, i.e., 32 KB (see Figure 120).

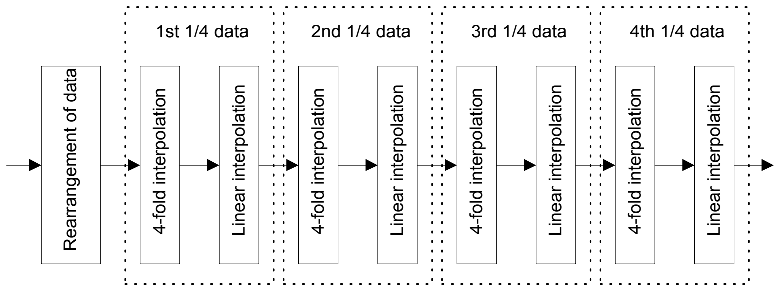


Figure 120. Mapping of Timing & Sampling Freq. Offset Correction

FFT

The Radix-2 method is adopted for this 8 K FFT. For all of the 13 stages, 64 SIMD operations can be easily achieved since every sample in every stage involves the same butterfly operation. Among them, seven stages work in a pure SIMD fashion with no data movements between RCs. The remaining six stages require some regular data movement using inter-RC connection; other parts of the mapping are done in the SIMD fashion. Presorting of input data is simple with the help of inter-RC connection, while the post-sorting is done by reordering the FB, which has 64 parallelisms (see Figure 121).

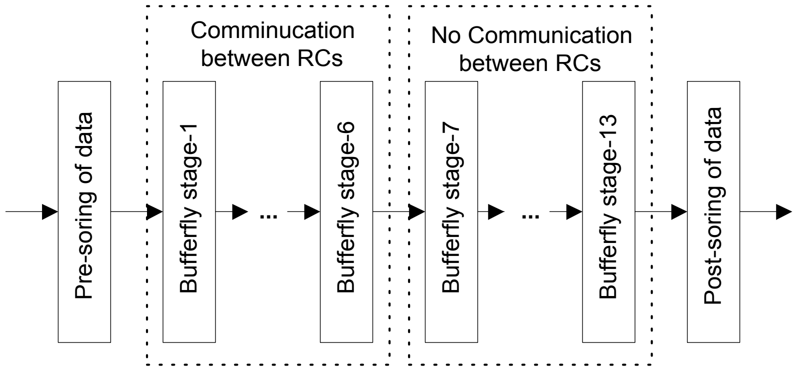


Figure 121. Implementation of 8K FFT

The first step in FFT is doing a bit-reverse presorting. This presorting alone takes more than 14,000 cycles on TI's TMS320C62x™ DSP. The FB is capable of doing this presorting utilizing 64 parallelism. In a real system delivered data to FFT subsystem are in the second order of FB. If data is in RC-SRAM, it will be in second order as well; so the presorting scheme that will be presented is general. In the latter case, data from SRAM should be transferred to FB which will take 256 cycles. Eventually, the organization of data in the FB is as shown in figure 122.

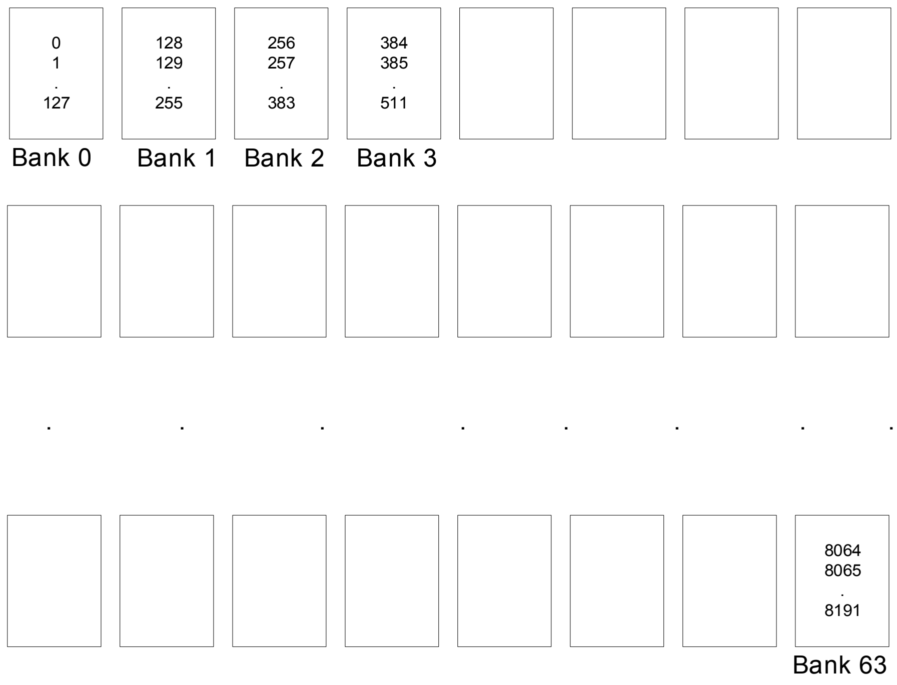


Figure 122. Data Layout in FB before Presorting

Now assume the following presorting algorithm:

1. Read data in row *i*
2. Write it back to row bit-reverse (*i*) while the FB crossbar switch is set in a bit-reversing order; which means to do realize the permutation of table 21.

Table 21. Presorting Permutation Table

	C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
Row 0	0	1	2	3	4	5	6	7
Row 1	8	9	10	11	12	13	14	15
Row 2	16	17	18	19	20	21	22	23
Row 3	24	25	26	27	28	29	30	31
Row 4	32	33	34	35	36	37	38	39
Row 5	40	41	42	43	44	45	46	47
Row 6	48	49	50	51	52	53	54	55
Row 7	56	57	58	59	60	61	62	63

Bit $\Rightarrow$ Reverse

C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
0	32	16	48	8	40	24	56
4	36	20	52	12	44	28	60
2	34	18	50	10	42	26	58
6	38	22	54	14	46	30	62
1	33	17	49	9	41	25	57
5	37	21	53	13	45	29	61
3	35	19	51	11	43	27	59
7	39	23	55	15	47	31	63

But this permutation is not possible by the quasi-crossbar configuration of the FB because, as can be observed, all the elements in one row should move to the same column. This permutation needs a complete crossbar; therefore, this permutation is not realizable in a single cycle. But we show that we can do such a permutation in two steps. At first step we send the data to a temporary row in the frame buffer (in first order), while the crossbar switch is set to allow the permutation shown in table 22.

Table 22. First Level Permutation to Realize Bit-reversed Permutation

	C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
Row 0	0	1	2	3	4	5	6	7
Row 1	8	9	10	11	12	13	14	15
Row 2	16	17	18	19	20	21	22	23
Row 3	24	25	26	27	28	29	30	31
Row 4	32	33	34	35	36	37	38	39
Row 5	40	41	42	43	44	45	46	47
Row 6	48	49	50	51	52	53	54	55
Row 7	56	57	58	59	60	61	62	63

1st $\Rightarrow$ Rel.*

C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
0	36	22	50	15	43	29	57
1	37	23	51	8	44	30	58
2	38	16	52	9	45	31	59
3	39	17	53	10	46	24	60
4	32	18	54	11	47	25	61
5	33	19	55	12	40	26	62
6	34	20	48	13	41	27	63
7	35	21	49	14	42	28	56

*: Rel. stands for relocation

Then on the second step we write the data in the temporary row to the desired row, which is the bit-reverse of the original row while the crossbar switch is programmed to perform permutation in table 23. Thus the presorting algorithm is accomplished following the mentioned steps. This algorithm is equal to decomposing the presorting of 13 bits into a 7-bit and a 6-bit (and then decomposing this 6-bit to two 3-bit) bit-reversed operation. This decomposition speeds up the operation. The overhead is associated with the crossbar configuration data, which for each permutation is 64*6=384 bits. M2 can cache 16 set of crossbar permutation data; however only three permutations are needed (one default crossbar plus two for realizing presorting 2-level permutations). This presorting can be done in:

2(one to write to temp)*2(real and imaginary)*128(8192/64=128)

which adds up to 512 cycles.

Table 23. Second Level Permutation to Realize Bit-reversed Permutation

	C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
Row 0	0	1	2	3	4	5	6	7
Row 1	8	9	10	11	12	13	14	15
Row 2	16	17	18	19	20	21	22	23
Row 3	24	25	26	27	28	29	30	31
Row 4	32	33	34	35	36	37	38	39
Row 5	40	41	42	43	44	45	46	47
Row 6	48	49	50	51	52	53	54	55
Row 7	56	57	58	59	60	61	62	63

2nd ⇨ Rel.

C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7
0	33	18	51	12	45	30	63
32	1	50	19	44	13	62	31
16	49	34	3	28	61	46	15
48	17	2	35	60	29	14	47
8	41	26	59	20	53	38	7
40	9	58	27	52	21	6	39
24	34	42	11	36	5	54	23
56	25	10	43	4	37	22	55

Table 24 illustrates the bit-reversed table for the 7-bit part that defines which row to write, the data read from row i. Note that presorting is relocation to a new space in FB, so it can't be performed in-place (it's not in form of swap operation, so data should not be overwritten.).

Table 24. 7-Bit Bit-reversed Pattern Table for 8K FFT Presorting

0	0	16	4	32	2	48	6	64	1	80	5	96	3	112	7
1	64	17	68	33	66	49	70	65	65	81	69	97	67	123	71
2	32	18	36	34	34	50	38	66	33	82	37	98	35	114	39
3	96	19	100	34	98	51	102	67	97	83	101	99	99	115	103
4	16	20	20	36	18	52	22	68	17	84	21	100	19	116	23
5	18	21	84	37	82	53	86	69	81	85	85	101	83	117	87
6	48	22	52	38	50	54	54	70	49	86	53	102	51	118	55
7	112	23	116	39	114	55	118	71	113	87	117	103	115	119	119
8	8	24	12	40	10	56	14	72	9	88	13	104	11	120	15
9	72	25	76	41	74	57	78	73	73	89	77	105	75	121	79
10	40	26	44	42	42	58	46	74	41	90	45	106	43	122	47
11	104	27	108	43	106	59	110	75	105	91	109	107	107	123	111
12	24	28	28	44	26	60	30	76	25	92	29	108	27	124	31
13	88	29	92	45	90	61	94	77	89	93	93	109	91	125	95
14	56	30	60	46	58	62	62	78	57	94	61	110	59	126	63
15	120	31	124	47	122	63	126	79	121	95	125	111	123	127	127

After presorting, the data that was in the second order of FB will be in the first order. The reason that we chose radix-2 for our mapping is that in radix-4, though there are fewer stages, each stage is more complicated. Radix-2 or radix-4 does not have any advantage from the point of view of SNR, but the number of multiplications is fewer in radix-4. The data communication overhead was the main reason for choosing radix-2. Another reason was the ease of programmability. Notice that 8192 can not be displayed in form of 4^N, therefore it couldn't be implemented in all radix-4 stages. So a combination of radix-2 and radix-4 is needed. To eliminate the need to have both algorithms and butterflies and eventually context planes, all radix-2 stages are chosen. This makes operations in each stage almost alike, which leads to a very short TinyRISC code. Another important aspect of our method is using redundant twiddle factors saved in the FB that when combined with FB features, make the data available for RCs in a single cycle. Instead of saving 4096 twiddle factors, we need to save 8448 twiddle factors. This redundancy pays off and this method is used in almost all of the FFT mappings, including standard DSPs. For each stage, we set the twiddles in the order we need them in frame buffer. The number of twiddles for each stage is shown in table 25.

Table 25. Redundant Twiddles

Stage	Twiddles needed	Stage	Twiddles needed
1	0	8	128
2	64 (including 1s)	9	256
3	64 (including 1s)	10	512
4	64 (including 1s)	11	1024
5	64 (including 1s)	12	2048
6	64 (including 1s)	13	4096
7	64		

The total number of twiddles adds up to:

$$64+64+64+64+64+64+128+256+512+1024+2048+4096 = 8,848$$

These twiddles are set in the FB in a way that using the default configuration of crossbar switch, it can feed RCs with the appropriate twiddle. This makes the addressing very easy as we just need to address the bank(0,0) of frame buffer.

Because of the number of data items and the nature of the FFT algorithm, which incorporates butterflies between points in various locations, data communication is inevitable in this algorithm. We planned to use the whole RC Array for the mapping, which obviously makes sense when we are talking about 8,192 points; thus in each RC we will have 8,192/64=128 points of data. The mapping scheme presented is to distribute the first 6 stages on all RCs and as FFT butterflies distance increase as a power of 2 (distance of the points in each RC is 64, which is power of 2), after the sixth stage, all butterflies will be inside each RC with the twiddles fed from the frame buffer in one cycle.

First, we load 128 points data from FB to RCs. Data in the FB is partitioned in two separate chunks, one for real parts and one for imaginary parts (this implies that our data format is 16-bit real; 16-bit imaginary). Data width is established because of the SNR criterion. As a rule of thumb, we need one bit for each stage to have an acceptable SNR, so for 8K FFT at least 13 bits are needed. So we have two fixed index and one offset to address the points. Using the first order of the FB and selecting the next line offset as 8, we can load 64 consecutive points in each cycle. We use registers R0 through R3 to allocate data. R0 and R1 are used for the first data (R0 for real, R1 for imaginary), and R2 and R3 are used for the second data. So, the data 0, for instance, will go to RC(0,0), and data 1 will go to RC(1,0) and … and data 64 will go to RC(0,0) and so on and so forth. Now it is possible to operate 128-point FFT on these points. The 128-point FFT diagram is shown in figure 123.

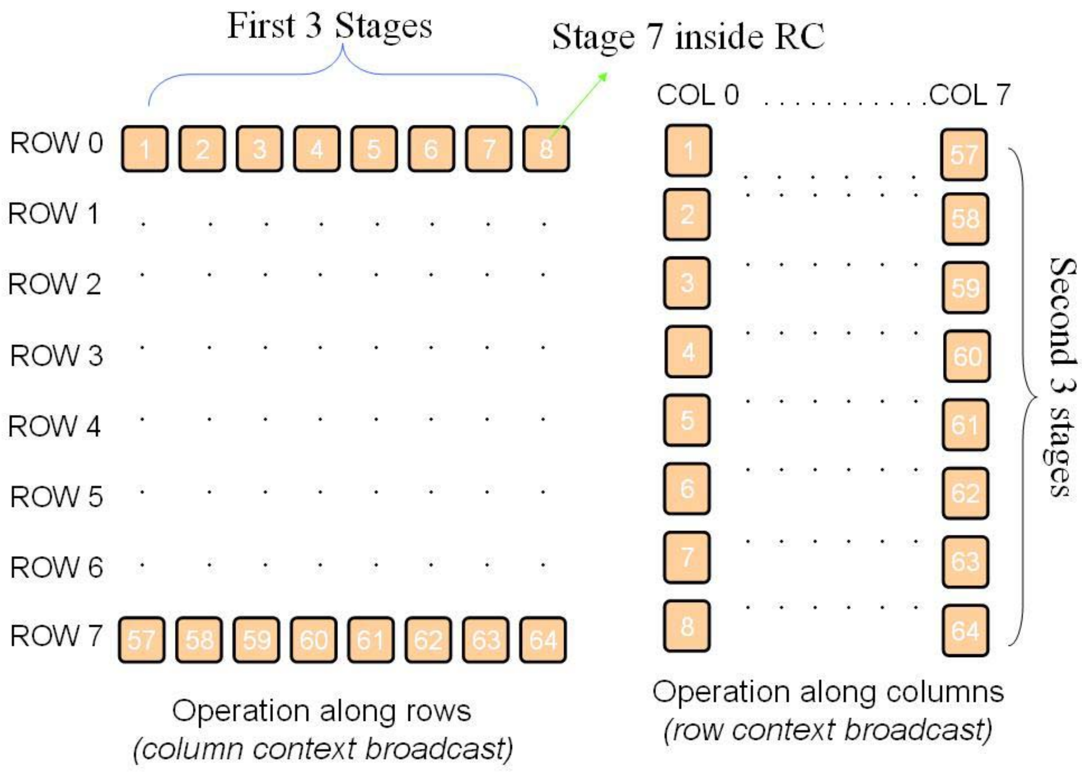


Figure 123. First Seven Stages of FFT

As stated in the diagram, the first three stages are operations on columns and contexts for butterfly operations are broadcasted along columns. For instance the first-stage butterfly is between data in RC(0,0) with RC(0,1), and RC(0,2) with RC(0,3), and so on. For the first stage, no twiddle is needed. For the second stage first we need a complex multiplication by twiddle factors, which MorphoSys is not capable of doing that in single instruction, when it is 16-bit. The reason is that complex multiplier needs four 16-bit multiplier which essentially will be slow and occupies a lot of area. So using twiddles fed from the FB each multiplication takes four cycles by using MAC operations. Shifting back to right is done in the end to avoid overflow during the same phase. Following equation show how a single complex multiplication is converted to four regular multiplications.

$$(a+bj)*(c+dj)=(ac-bd)+(ad+bc)j$$

Then the second-stage butterfly should be done for the data in RC(0,0) with RC(0,2), and RC(0,1) with RC(0,3), and so on and so forth. The butterflies for other stages are almost the same. First, twiddles are multiplied (the redundant twiddle in FB is padded with dummy ones so that the context can be broadcasted on all RCs); then the butterfly operations are done. It should be noted that stages 4 through 6 are done along rows, and context is broadcasted along rows with the same pattern as the first three stages, but on different rows. For the seventh stage, the butterfly should be done between the points that are 64 points apart, i.e., data 0 with 64, data 1 with 65 and so on, which they all will all happen to be in same RC. The result after the seventh stage is saved in RC SRAM. If we do this 128-point FFT operations 64 times, we have done the first seven stages of FFT on all data and they are in RC SRAM thereafter. The same relationship rules apply to the last 6 stages because the distance between butterfly points counts 64, so they will be all inside one RC. For instance, the eighth stage butterfly is between the data entries in SRAM address 0 and 2 with first twiddle plane, and between data in SRAM address 1 and 3 with second set of twiddles. Figures 124 and 125 elaborate more on the mentioned procedure.

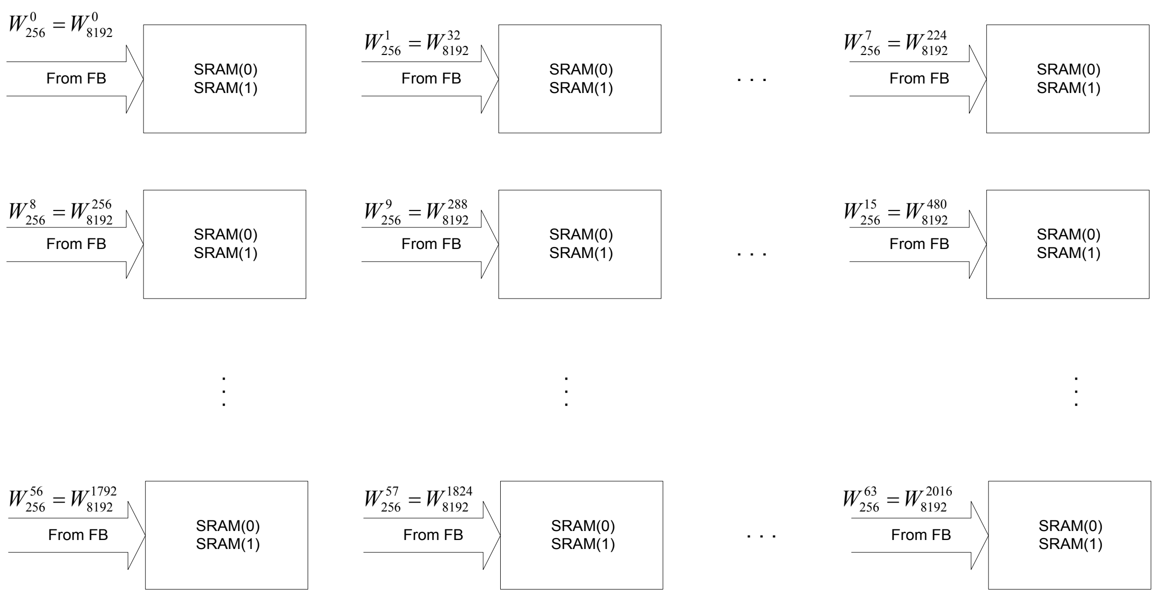
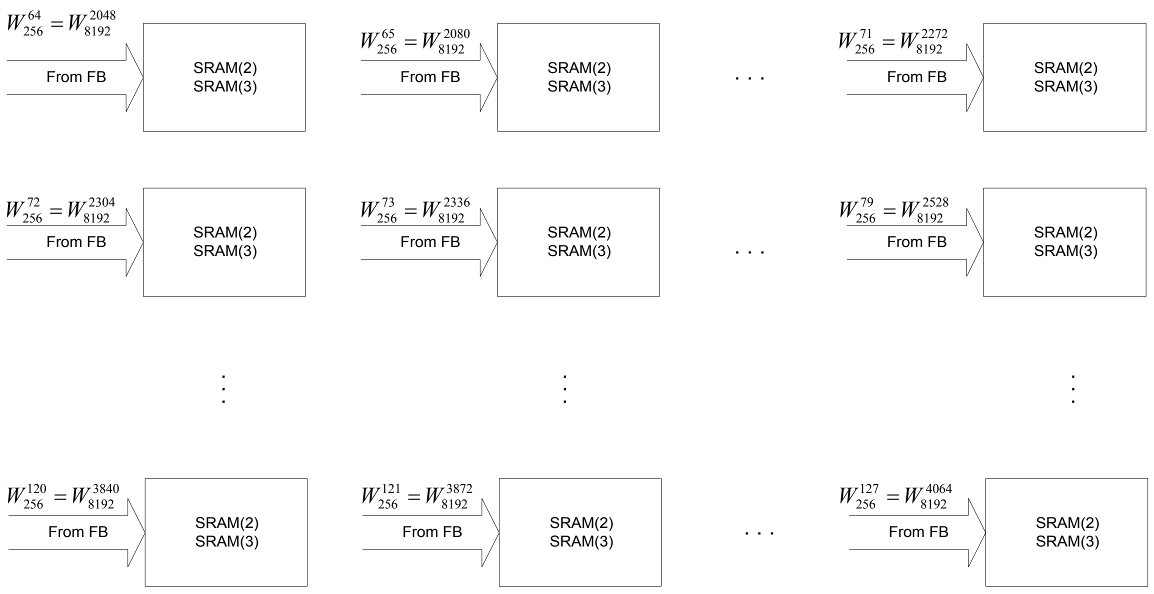

$W_{256}^{0}=W_{8192}^{0}$
From FB
SRAM(0)
SRAM(1)
$W_{256}^{1}=W_{8192}^{32}$
$W_{256}^{7}=W_{8192}^{224}$
$W_{256}^{8}=W_{8192}^{256}$
$W_{256}^{9}=W_{8192}^{288}$
$W_{256}^{15}=W_{8192}^{480}$
$W_{256}^{56}=W_{8192}^{1792}$
$W_{256}^{57}=W_{8192}^{1824}$
$W_{256}^{63}=W_{8192}^{2016}$

Figure 124. First step of 8th Stage with First Twiddle Set



$W_{256}^{64} = W_{8192}^{2048}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{65} = W_{8192}^{2080}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{71} = W_{8192}^{2272}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{72} = W_{8192}^{2304}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{73} = W_{8192}^{2336}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{79} = W_{8192}^{2528}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{120} = W_{8192}^{3840}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{121} = W_{8192}^{3872}$
From FB
SRAM(2) SRAM(3)
$W_{256}^{127} = W_{8192}^{4064}$
From FB
SRAM(2) SRAM(3)

Figure 125. Second Step of 8th Stage with Second Twiddle Set

As can be seen, data will be loaded from SRAM, then the second data is multiplied by twiddle coming from FB, then butterfly operation is done inside the RC, and then stored back to SRAM. The MorphoSys load and store indirect instructions have enough flexibility so that all these loads and stores are done automatically so the whole program can be wrapped in a two level nested loop. For the eighth stage, the mentioned operation should be done 32 (8,192/256=32) times. The following stages are almost the same, but the number of steps will be doubled in the next stage, and the number of times that loop should be executed will be half. This concludes are scheme for mapping 8,192 points FFT algorithm on MorphoSys.

The TinyRISC code and contexts for this mapping has written and the whole program with a set of test data has been executed on our cycle accurate MorphoSys simulator using out context assembler and out TinyRISC assembler. Figure 126 shows the mapping result from the number of cycles' point of view according to our simulation.

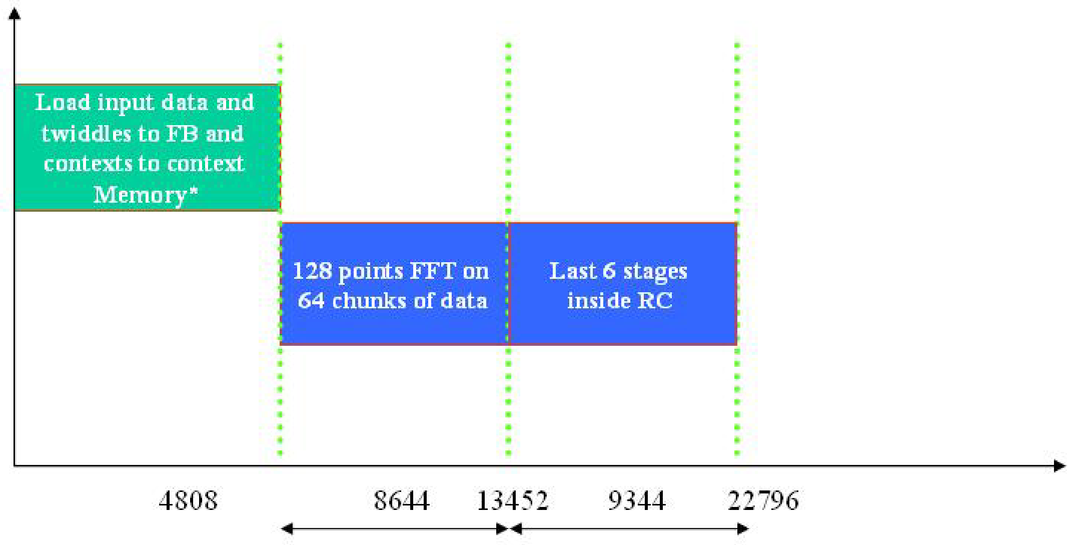


Figure 126. Cycle Statistics for Mapping 8,192 Points FFT

Notice that the twiddle factor loading and context loading is done just once in the beginning. Moreover, in real application data is already in the frame buffer, so loading data into forward should not be counted in total cycles. Therefore the FFT computation cycles will be counted as 17,988. Table 26 shows some specifications of the 8,192 points FFT function along with the exact number of cycles.

Table 26. Specifications of 8192 Points FFT Function

Function name	FFT8k
Input length	8192 points
Input type	Complex , signed
Input bit length	32 bits , 16 bit real and 16 bit imaginary
Coefficients (twiddle factors) bit length	16 bits
Output type	Complex , signed
Output bit length	32 bits 16 bit real and 16 bit imaginary
TinyRISC registers	R0-R10 used for addressing and making loops
RC registers	R0-R7 Used for computations R8-R13 Used for addressing inside RC SRAM
Contexts Broadcasting instructions	15109
Input bit precision	14 bits
Average RC parallelism	Theoretically 64 (out of 64) (There is no NOP in contexts)
Library function size	FFT8k_T : 441 lines (line = 32bits) R-Ctxts : 42 Planes (Plane = 8* 32Bits) C-Ctxts : 30 Planes (Plane = 8* 32Bits)

Frequency Offset Tracking

This small block first estimates phase error and then calculates the output of the Proportional-Integral (PI) controller for both carrier and sampling frequency offsets. Most of the computation is sequential code on TinyRISC.

Fine Timing Synchronization

This block is executed every 16 OFDM symbols to track small residual timing offset. Firstly, scattered pilots are zero-padded to 4,096 points using the fast continuous data movement capability of the FB. Secondly, 4,096 points Inverse Fast Fourier Transform (IFFT) is calculated, reusing most of the codes of 8192 points FFT. Then, the total windowed energy is calculated in parallel on RC array. At last, the window is shifted, and the peak energy is found by sequential codes on TinyRISC.

Channel Estimation, Equalization and Pseudorandom Deinterleaving

The purpose of interleaving/deinterleaving is to shuffle the data stream to avoid clustered errors. Pseudorandom deinterleaving is notoriously difficult for its nature of random data movement with no temporal locality (nonrepeated addresses) and little spatial locality (random addresses). Hence, it is dealt with sequentially by DMM, which reduces the cycle number from 64 K, if using TinyRISC sequential codes, to 24 K. Furthermore, since it only involves FB, overlapping it with other blocks running on RC array and/or TinyRISC can save more cycles. These blocks are channel estimation and equalization. However, there is data dependency among these blocks. A solution is to pipeline these blocks and utilize the Multiple Program Multiple Data (MPMD) parallelism between FB (in DMM mode) and other components (Figure 127). Thus, nearly 99 percent of the cycles for channel estimation and equalization blocks are hidden.

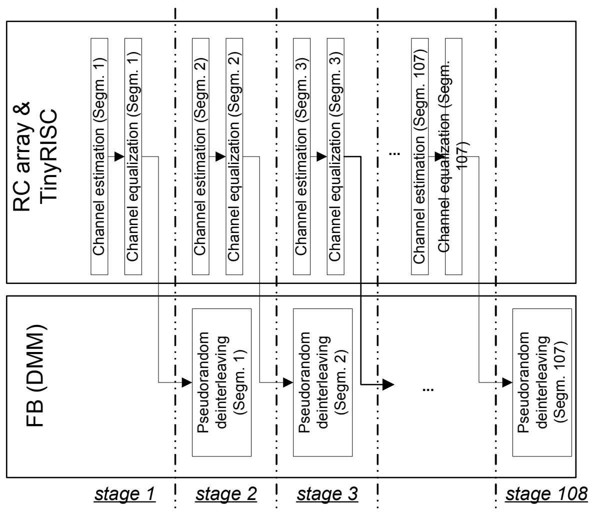

RC array & TinyRISC
FB (DMM)
Channel estimation (Segm. 1)
Channel equalization (Segm. 1)
Channel estimation (Segm. 2)
Channel equalization (Segm. 2)
Channel estimation (Segm. 3)
Channel equalization (Segm. 3)
...
Channel estimation (Segm. 107)
Channel equalization (Segm. 107)
Pseudorandom deinterleaving (Segm. 1)
Pseudorandom deinterleaving (Segm. 2)
...
Pseudorandom deinterleaving (Segm. 107)
stage 1
stage 2
stage 3
stage 108

Figure 127. Pipeline of Channel Estimation, Equalization and Deinterleaver

The channel equalization block needs a division of 10-bit precision. It is done by a reciprocal operation, followed by a multiplication. Similar to sine/cosine operation, the reciprocal operation is also implemented as a small 16-entrée LUT followed by a linear interpolator. The difference is that here the input is first normalized by a leading-zero-detect instruction (Figure 128).

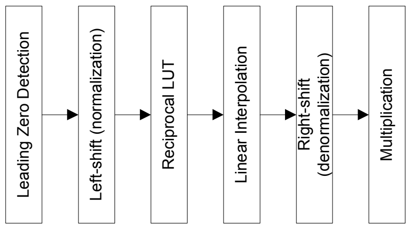


Figure 128. Division Implementation

Demapping, Bit-wise Deinterleaving, and Viterbi Decoder

The same as in timing and sampling frequency offset correction block, here the input data frame is also truncated to 18 segments, and the 3 blocks are pipelined to reduce the temporary storage of branch metrics from 145 KB to 8.1 KB.

Viterbi decoder is the most computation-intensive block. First, the input data segments are arranged to overlap at both edges to reduce the extra error caused by the frame truncation to a negligibly low level. Then, forward metrics computation in RC array is overlapped with sequential trace-back in TinyRISC (Figure 129). This MPMD parallelism between TinyRISC and RC array totally hides the trace-back subblock, resulting in 31 percent reduction of the cycle number. Careful optimization has achieved a very high speed of nine cycles per decoded bit, corresponding to 47 to 60 Mbps/chip under the current technology. This is near the ASIC performance, while no special add-compare-selection (ACS) instruction is added, unlike the other DSPs do [31].

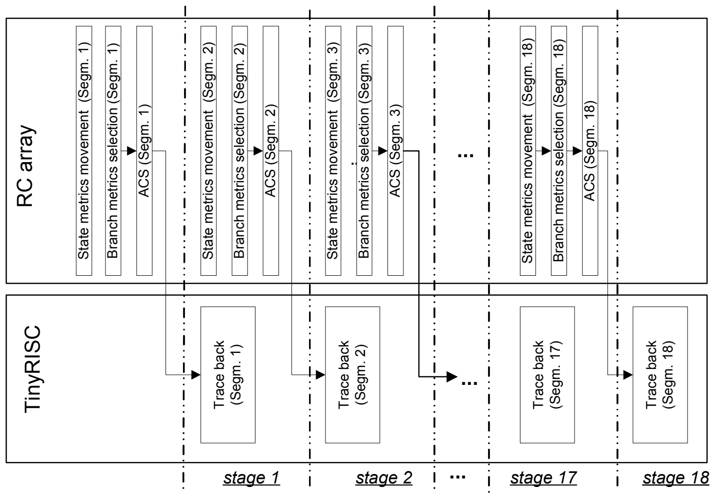

RC array
TinyRISC
State metrics movement (Segm. 1)
Branch metrics selection (Segm. 1)
ACS (Segm. 1)
State metrics movement (Segm. 2)
Branch metrics selection (Segm. 2)
ACS (Segm. 2)
State metrics movement (Segm. 3)
Branch metrics selection (Segm. 3)
ACS (Segm. 3)
...
State metrics movement (Segm. 18)
Branch metrics selection (Segm. 18)
ACS (Segm. 18)
Trace back (Segm. 1)
Trace back (Segm. 2)
...
Trace back (Segm. 17)
Trace back (Segm. 18)
stage 1
stage 2
...
stage 17
stage 18

Figure 129. Pipeline Implementation of Viterbi Decoder

Outer Deinterleaving

There can be 11 outer deinterleavers executing in parallel.

Descrambler

Descrambling is done independently on each sample, so 64-fold SIMD parallelism can easily be achieved.

MPEG-2 Decoder

The estimation is based on the information provided for mapping of MPEG-2 encoder [28].

Summary

Table 27 gives an estimate of the mapping result of the whole system.

Table 27. Mapping Estimate of DVB-T onto MorphoSys

Units	Cycle #[1] (K)	Permanent storage[2] (KB)	Tempo-rary storage (KB)	Context size (KB)	TRISC inst. Size (KB)
Buffer add. calculation	1	32	0.0	0.0	0.2
Carrier freq. correction	11	32	0.1	0.3	0.1
Other offset correction	17	0.0	34	0.3	0.1
FFT	24	0.0	8	0.9	1.2
Freq. offset track	1	0.0	0.5	0.1	0.4
Fine timing sync.	1	0.0	16	0.1	1.3
Channel estimation[3]	17	0.0	3	0.1	0.2
Channel equalization[3]	5	96	8	0.1	0.1
Random deinterleaving[3]	24	0.0	13	0.0	0.1
Demapping[4]	4	0.0	8	0.2	0.1
Bit-wise deinterleaving[4]	7	0.0	0.0	1.0	0.3
Viterbi decoder[4]	175	0.0	8	3	0.5
Outer deinterleaving	5	0.0	5	0.1	0.1
Reed-Solomon decoder	5	0.0	48	3.2	0.4
Descrambler	1	0.0	5	0.0	0.0
MPEG-2 decoder[5]	179	0.0[6]	64	4.0	7.0
Total or maximum[7]	455	160	64	13.4	12.1

1 The OFDM symbol duration is 1.12 ms.
2 The permanent memory doesn't count in what has been required by former units.
3 These units are pipelined and overlapped so that the total cycle number is 24 K.
4 These three units are pipelined to reduce the temporary storage.
5 MPEG-2 decoder (MP@ML) mapping estimate is mainly made based on the previous mapping of parallel kernels [28], except for that sequential code on TinyRISC for VLD is estimated according to many public references. Note that overlapping of the sequential VLD with parallel code for channel Demodulation and Decoding subsystem could further reduce the cycle number much lower. Only video decoding is considered.
6 Reference Pictures are stored in External Memory.
7 It is assumed that the loading of all the temporary data, context, and TRISC code from External Memory is 100 percent overlapped with the execution of other units. This assumption will cause cycle number estimation error within 30 K, most probably.

7. Research Summary

Recongurable computing has opened new frontiers in the field of computer architecture and poses unique problems for the development of programming software. This work demontrated MorphoSys, a new model for reconfigurable computing systems, developed to investigate the effectiveness of combining multiple reconfigurable processing elements with a general-purpose processor for word-level, computation-intensive tasks such as multimedia applications.

MorphoSys is a coarse-grain, integrated reconfigurable system-on-chip targeted at high throughput and data-parallel applications such as video compression. It comprises an array of reconfigurable cells (RC Array), a modifieded RISC processor core (TinyRISC) and an efficient memory interface unit. We first described the MorphoSys-M1 architecture, including the RC array, the TinyRISC, and data and configuration memories. The innovative features of M1 are clarified with emphasis on the SIMD nature of the RC Array. The extensive functional and interconnect reconfigurability is described along with the illustration of its utility in efficiently executing a set of target applications.

We then presented the detailed design implementation and the various aspects of physical layout of different sub-blocks of MorphoSys. The first-generation MorphoSys, M1, was implemented using CMOS 0.35 μm technology. It was designed to operate at 100 MHz of clock frequency. We provided simulation results of the M1 for some typical data-parallel applications (e.g. video compression and automatic target recognition). The results indicate that the MorphoSys system can achieve significantly better performance for most of these applications in comparison with other systems.

In Chapter 4 comprehensive software environment including simulators with graphical user interface, compiler and CAD tools, were developed for supporting the mapping of applications to Morphosys-M1. Several applications from the target application domain, such as video compression, data encryption and target recognition were mapped to the M1 architecture. Performance evaluation of these applications indicates improvements of up to an order of magnitude in comparison with other systems.

Version M2 of MorphoSys, with some major enhancements over M1, was introduced in Chapter 5. A local memory in each RC, a DMA controller to establish a direct path between this memory and the global data memory, guarded instructions and pseudo branches are some of the advanced features added to the new version.

In Chapter 6 a pure-software single-chip solution based on a modified MorphoSys was proposed. We first described DVB-T receiver system design and then presented a fixed-point simulation result conforming to the protocol. Then, we introduced the architecture and programming model of the modified MorphoSys processor. Lastly, algorithm mapping procedure and the results with orders of magnitude higher computation capability than other commercial DSP chips were demonstrated.

8. Future Work

Meta-MorphoSys

Design of MorphoSys has shown great potential for data intensive applications such as MPEG-2, encryption/decryption and ATR. Although MorphoSys can achieve an order of magnitude improvement in terms of clock cycles over current popular general-purpose processors, and other contemporary reconfigurable computing systems, its performance is limited by the number of the reconfigurable cells and external memory bandwidth. In this chapter we introduce Meta-MorphoSys, a scalable MorphoSys, which overcomes the limitations of MorphoSys and possibly improves performance by another order of magnitude for data intensive applications by applying the following two enhancements to the current design:

1) The number of Morpho units on the chip is scaled to a level that renders an order of magnitude speedup. Each Morpho units is a MorphoSys-like reconfigurable system.
2) The memory bandwidth is increased by a factor of ten by using embedded DRAM technology to keep the Morpho units busy.

Meta-MorphoSys architecture is a scalable dynamically reconfigurable computing architecture, consisting of a myriad of identical MorphoSys-like reconfigurable blocks, called Morpho units. The number of Morpho units will be scaled based on the target applications. Meta-MorphoSys includes new features in addition to the characteristics of MorphoSys architecture introduced in the previous chapters. Among those new features is the embedded DRAM, which is called eM-CDRAM (embedded Multiple-Cache DRAM). Meta-MorphoSys is a new class of microchips that exploits the data parallel features of data intensive applications with high bandwidth capabilities of embedded DRAM technology on dynamically reconfigurable architecture.

The centerpiece of the architecture are multiple Morpho units, which can be scaled based on the performance requirements of applications. In order to provide the necessary memory bandwidth, each Morpho unit is directly connected to a dedicated embedded DRAM (eDRAM) block. The main motivation to integrate a large piece of DRAM closer to processors is to remove some of the constraints that slow down the memory access. For instance, bus widths may be increased from the more conventional pin-limited 32 bits up to 512 or even more. Besides the apparent increase in raw data rate, off-chip drivers are eliminated and their removal can significantly reduce the power dissipated and electrical noise generated. Embedded DRAM will also make it possible to re-evaluate the feasibility of a return to the simplest interface: separate input bus and output bus, and full address bus (not row, then column, multiplexing of the address—so called RAS/CAS in commodity DRAMs). This simple architecture eliminates the overhead of front-end logic on the memory and the logic associated with memory controllers to create the extremely sensitive DRAM timings. Using the above features, the embedded DRAM design on the same chip as the processor will improve the performance of the system.

Since each Morpho unit has its own TinyRISC, and can fetch and execute different codes as well as data, it will increase the flexibility of mapping algorithms to Meta-MorphoSys architecture. In addition, this architecture will lead to much more efficient utilization of the chip by using different Morpho units for different sections of codes as well as for different data sets. This will allow three different models of computation for Meta-MorphoSys architecture:

•Massive SIMD computation model: It is a simplest computation model. At any clock cycle, Every Morpho unit will perform the same operations but different sets of data as shown in Figure 9-1 a).
•Multithreaded SIMD computation model: In this case, different Morpho units in the scalable system run either different application kernels or different applications as shown in Figure 9-1 b).
•Pipelined SIMD computation model: Many of non-block data applications, such as transposed FIR with a large number of taps, require more than one Morpho unit in order to keep sustained and high throughput. Therefore, parts of the operations can be mapped to one Morpho unit, while the rest are mapped to other Morpho units. The data transfer between Morpho units will be based on direct connectivity of each Morpho unit with its four nearest neighboring units. Even though many of block data applications, such as MPEG-2, could be mapped into pipelined SIMD model if the wide data bus existed between neighboring Morpho units. Under this hypothetic assumption, each Morpho unit would work on a certain kernel. After a while, the data would be moved to next Morpho unit to perform the next operation. However, the performance of the pipelined SIMD model for block data applications will be inferior to that of the Massive/Multithread SIMD model at current system configuration with embedded DRAM. In fact, the essential point is which approach is more time-consuming to transfer data or reload contexts to Context Memory with fixed-number of context planes. It can be shown that the data transfer among Morpho units will consume more clock cycles than the context reloading.

Therefore, we can avoid the pipelined SIMD model for block data applications by transforming the pipelined SIMD model to massive/multithread SIMD model, and only keep pipelined SIMD model for non-block data application. This simplification will also incur much easier mesh connection among Morpho units with much less bitwidth. For the Meta-MorphoSys, it is straightforward to realize the massive and multithreaded computation paradigms. In both cases, the data will not be exchanged among different Morpho units. Each Morpho unit will perform a segment of the code, which is independent of the others. This is a pure extension of previous MorphoSys architecture and the operations will be performed in parallel. It can use the same approach to reconfigure or schedule the application kernels as that in MorphoSys. However, for the pipelined SIMD model of the block data applications, the decision is dependent on data flow graph (DFG) of the application and system configuration, such as the bit-width for context and for data transfer among Morpho units.

Meta-MorphoSys architecture exposes several new challenges and design issues that can revolutionize the research direction of current computer architecture. It is distinct from traditional microprocessor architecture in two different aspects.

System-on-chip architecture: scalable reconfigurable SIMD units, embedded DRAM and general-purpose processors are to be integrated on a single chip. This requires a significant redesign of the pin interface of the traditional microprocessor architecture.
Multi-Level Parallelism. Meta-MorphoSys exploits massive, pipelined and multithreaded parallelism on top of SIMD parallelism by reconfiguring the contexts and data streams of the processors. This will lead to higher resource utilization and more flexibility in exploiting parallelism in data-intensive applications as compared to traditional SIMD processing paradigm.

9. References

1. A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor and N. Bagherzadeh, "Design and Implementation of TinyRISC microprocessor," in *Microprocessors and Microsystems*, Vol. 16, No. 4, 1992.

2. Ebeling, D. Cronquist, and P. Franklin "Configure Computing: The Catalyst for High-performance Architectures," *Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 364-372, July 1997.

3. E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," in *Proceedings of IEEE Symposium on Field-programmable Custom Computing Machines*, April 1996.

4. T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

5. J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agrawal, "The RAW Benchmark Suite: computation structures for general-purpose computing," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM 97, pp. 134-143, 1997.

6. Hartej Singh, "Reconfigurable Architectures for Multimedia and Data-Parallel Application Domains," Ph.D. thesis, Department of Electrical and Computer Engineering, University of California, Irvine

7. T. Arai, I. Kuroda, K. Nadehara and K. Suzuki, "V830R/AV: Embedded Multimedia Superscalar RISC Processor," *IEEE MICRO*, Mar/Apr 1998, pp. 36-47.

8. http://www.darpa.mil/ito/research/acs/challenges.html/

9. Rencher and B.L. Hutchings, "Automated Target Recognition on SPLASH 2," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machine,* April 1997.

10. Villasenor, B. Schoner, K. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable Computing Solutions for Automatic Target Recognition," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machine*, April 1996.

11. D. Wood, "Bits 'R' Us—new economics and approaches for digital broadcasting," *EBU Technical Review*, January 2002.

12. "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television," *ETSI EN 300 744 v1.4.1*, January 2001.

13. F. Frescura, et al., "DSP based OFDM demodulator and equalizer for professional DVB-T receivers", IEEE Trans. Broadcasting, Vol. 45, No. 3, pp. 323-332, Sept. 1999.

14. F. Frescura, et al., "C6000 based DVB-T receiver," *Digilab2000*, University of Perugia, Italy.

15. M. Speth, S. Fechtel, G. Fock, and H. Meyr, "Optimum Receiver Design for OFDM-Based Broadband Transmission—Part II: A Case Study," IEEE Trans. Commun., Vol. 49, No. 4, pp. 571-578, April 2001.

16. M. Speth, S. Fechtel, G. Fock, and H. Meyr, "Optimum Receiver Design for Wireless Broadband Systems Using OFDM—Part I," *IEEE Trans. Commun.*, Vol. 47, No. 11, pp. 1668-1677, Nov. 1999.

17. T. Schmidl and D. Cox, "Robust Frequency and Timing Synchronization for OFDM," *IEEE Trans. Commun.*, Vol. 45, No. 12, pp. 1613-1621, December 1997.

18. S. Fechtel, "OFDM carrier and sampling frequency synchronization and its performance on stationary and mobile channels," *IEEE Trans. Consumer Electronics*, Vol. 46, No. 3, pp. 438-441, Aug. 2000.

19. Ye Li, "Pilot-Symbol-Aided Channel Estimation for OFDM in Wireless Systems," *IEEE Trans. Vehicular Tech.*, Vol. 49, No. 4, pp. 1207-1215, July 2000.

20. P. Hoeher, S. Kaiser, and P. Robertson, "Two-dimensional pilot-symbol-aided channel estimation by Weiner filtering," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, pp. 1845-1848, 1997.

21. J. Hsu and C. Wang, "An Area-efficient Pipelined VLSI Architecture for Decoding of Reed-Solomon Codes Based on a Time-Domain Algorithm," *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 7, No. 6, pp. 864-871, December 1997.

22. P. Hoeher, "A statistical Discrete-Time Model for the WSSUS Multipath Channel", *IEEE Trans. Vehicular Technology*, Vol. 41, No. 4, pp. 461-468, November 1992.

23. T. Wang, V. Dubey, and J. Ong, "Generation of Scattering Functions for Mobile Communication Channel: A Computer Simulation Approach", *International Journal of Wireless Information Networks*, Vol. 4, No. 3, pp. 187-204, 1997.

24. A. Anastasopoulos and K. Chugg, "An efficient method for simulation of frequency selective isotropic Rayleigh fading," *IEEE 47th Vehicular Technology Conference*, Vol. 3, pp. 2084-2088, May 1997.

25. "Digital cellular telecommunications system (Phase 2+); Radio transmission and reception (3GPP TS 05.05 version 8.9.0 Release 1999)," ETSI TS 100 910 v8.9.0, April 2001.

26. H. Singh, el al., “MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation –Intensive Applications,” *IEEE Trans. Computers*, Vol. 49, No. 5, pp. 465-481, May 2000.

27. M. Lee, el al., “Design and Implementation of the MorphoSys Reconfigurable Computing Processor,” *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, Vol. 24, No. 2-3, Kluwer Academic Publishers, pp. 147-164, March 2000.

28. M. Lee, “Design and Implementation of the High-Performance Low-Power MorphoSys Reconfigurable Processor,” PHD thesis, University of California, at Irvine, 2000.

29. A. Niktash, “Parallel Viterbi Decoder for Wireless Communication Using Regonfigurable Architecture,” accepted paper, 2002.

30. H. Parizi, “MorphoSys: A Coarse Grain Reconfigurable Architecture for MultiMedia Applications,” accepted paper, 2002.

31. T. Kumura, et al., “VLIW DSP for mobile applications,” *IEEE Signal Processing Magazine*, Vol. 19, No. 4, pp.10-21, July 2002.

10.List of ACRONYMS

ACRONYM	DESCRIPTION
3GPP	third generation partnership project
ACS	add-compare selection
ADC	analog to digital converter
AGU	address generator unit
ALU	arithmetic and logic unit
ASIC	application specific integrated circuit
ATR	automatic target recognition
BER	bit error rate
BFU	basic unit
BTM	binary template matching
CISC	complex instruction set computer
CLZO	count leading zeroes/ones
CM	context memory
CMOS	complementary MOS
COFDM	coded orthogonal frequency division modulation
CPL	complementary pass-transistor logic
CPU	central processing unit
CSA	carry-save adder
DCD	decoder
DCF	data control flow
DCT	discrete cosine transforms
DMA	direct memory access
DMAC	DMA controller
DMM	direct memory movement
DNRZ	delayed non-return-to-zero
DSP	digital signal processor
DRU	data register unit
DSP	digital signal processing
dst	destination (register)
DUT	device under test
DVB-T	digital video broadcasting-terrestrial
EDIF	electronic data interchange format
ETSI	European telecommunications standards institute

ACRONYM	DESCRIPTION
FA	full adder
FB	frame buffer
FFT	fast Fourier transform
FI	final identification
FIR	finite impulse response
FOA	focus of attention
FPGA	field programmable gate array
FSBM	full search block matching
GND	ground
GOPS	giga operations per second
HDTV	high definition television
HE	horizontal express lane
HT	Hilly Terrain
IBBP	I = intracoded picture, P = forward predicted picture, and B = bidirectionally predicted picture
IC	integrated circuit
IDCT	inverse discrete cosine transforms
IFFT	inverse FFT
IMS	integrated measurement systems
I/O	input/output
ISA	instruction set architecture
LSB	least significant bit
LUT	lookup table
MAC	multiplier-accumulator
MAD	mean absolute difference
MIPS	million instruction per second
MOS	metal oxide semiconductor
MPEG	moving picture experts group
MPMD	multiple programs multiple data
MSB	most significant bit
NMOS	n-channel MOS
NRZ	non-return-to-zero
OFDM	orthogonal frequency division modulation
OMSE	overall mean square error
OPA	operand A
OPB	operand B

ACRONYM	**DESCRIPTION**
PC	program counter
P/G	power and ground
PD	priority detector
PE	priority encoder
PI	proportional integral
PMOS	p-channel MOS
QAM	quadrate (quaternary) amplitude modulation
QEF	quasi-error-free
RAM	random access memory
RC	reconfigurable cell
RC	return –to-complement
RF	radio frequency
RISC	reduced instruction set computer
RZ	return-to-zero
RZI	return-to-zero-inverted
SAR	synthetic aperture radar
SLD	second level of detection
SIMD	single instruction multiple data
SFN	single-frequency network
SNR	signal to noise ratio
SQNR	signal quantization-noise ratio
SRAM	static RAM
TI	Texas Instruments
TPS	transmission parameter signaling
VE	vertical express lane
VLC	variable length encoding
VLD	variable length decoding
VLIW	very long instruction word
VLSI	very large scale integration
WCDMA	wideband code division multiple access
WSSUS	wide-sense-stationary uncorrelated-scattering
XQ	cross quadrant RC

Appendix A

TinyRISC ISA[1]-M1

A.1 Instruction Formats

In the TinyRISC ISA, the instructions assume one of the two formats shown below:

31 - 25	24	23 - 20	19 -16	15 - 12	11 - 0
OpCode	Immb	Sr1	Sr2	Dr	Unused

31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
OpCode	Immb	Sr1	Dr	Imm	

- OpCode: the 7-bit instruction opcode;
- Immb: the immediate bit. If Immb = 0, the second operand is stored in a data register file. If Immb = 1, the second operand is a 16-bit immediate value (Imm) extended to 32 bits;
- Sr1: the register id of the first operand;
- Dr: the id of the destination register;
- Sr2: the register id of the second operand;
- Imm: the 16-bit immediate value (if Immb = 1).
- UnsImm: unsigned immediate value.
- SignImm: signed immediate value.

A.2 Instruction Codes

The following subsections describe the instructions in each category: arithmetic, logical, shift, comparison, load immediate, memory access, control transfer, and MorphoSys instructions.

[1] TinyRISC ISA (except the MorphoSys instructions) was developed by Professor Eliseu M. Chaves Filho from COPPE/Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil.

A.2.1 Arithmetic Instructions

ADD Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000100	0	sr1	sr2	dr	unused

Description: This instruction adds the two unsigned values in registers sr1 and sr2 and writes the result into register dr.

ADDI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000100	1	sr1	dr	imm	

Description: This instruction adds the unsigned value in register sr1 to the zero-extended imm value and writes the result into register dr.

SUB Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000101	0	sr1	sr2	dr	unused

Description: This instruction subtracts the unsigned value in register sr2 from the unsigned value in register sr1 and writes the result into register dr.

SUBI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000101	1	sr1	dr	imm	

Description: This instruction subtracts the zero-extended imm value from the unsigned value in register sr1 and writes the result into register dr.

A.2.2 Logical Instructions

AND Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000000	0	sr1	sr2	dr	unused

Description: This instruction performs a bit-wise and of the values in registers sr1 and sr2 and writes the result into register dr.

ANDI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000000	1	sr1	dr	imm	

Description: This instruction performs a bit-wise and of the value in register sr1 and the zero-extended imm value and writes the result into register dr.

OR Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000001	0	sr1	sr2	dr	unused

Description: This instruction performs a bit-wise or of the values in registers sr1 and sr2 and writes the result into register dr.

ORI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000001	1	sr1	dr	imm	

Description: This instruction performs a bit-wise or of the value in register sr1 and the zero-extended imm value and writes the result into register dr.

XOR Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000010	0	sr1	sr2	dr	unused

Description: This instruction performs a bit-wise exclusive-or of the values in registers sr1 and sr2 and writes the result into register dr.

XORI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000010	1	sr1	dr	imm	

Description: This instruction performs a bit-wise exclusive-or of the value in register sr1 and the zero-extended imm value and writes the result into register dr.

XNOR Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000011	0	sr1	sr2	dr	unused

Description: This instruction performs a bit-wise exclusive-or followed by a not of the values in registers sr1 and sr2 and writes the result into register dr.

XNORI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0000011	1	sr1	dr	imm	

Description: This instruction performs a bit-wise exclusive-or followed by a not of the value in register sr1 and the zero-extended imm value and writes the result into register dr.

A.2.3 Shift Instructions

LSL Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010001	0	sr1	sr2	dr	unused

Description: This instruction shifts to the left the contents of sr1 by the amount indicated in sr2, inserting zeros on the right. The result is written into register dr.

LSLI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010001	1	sr1	dr	imm	

Description: This instruction shifts to the left the contents of sr1 by the amount indicated in imm, inserting zeros on the right. The result is written into register dr.

LSR Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010010	0	sr1	sr2	dr	unused

Description: This instruction shifts to the right the contents of sr1 by the amount indicated in sr2, inserting zeros on the left. The result is written into register dr.

LSRI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010010	1	sr1	dr	imm	

Description: This instruction shifts to the right the contents of sr1 by the amount indicated in imm, inserting zeros on the left. The result is written into register dr.

ASR Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010011	0	sr1	sr2	dr	unused

Description: This instruction shifts to the right the contents of sr1 by the amount indicated in sr2, replicating the most signi_cant bit. The result is written into register dr.

ASRI Dr, Sr1, UnsImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010011	1	sr1	dr	imm	

Description: This instruction shifts the contents of sr1 to the right by the amount indicated in imm, replicating the most signi_cant bit. The result is written into register dr.

A.2.4 Comparison Instructions

SLT Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001000	0	sr1	sr2	dr	unused

Description: This instruction signed compares the values in registers sr1 and sr2 and writes the value 0x00000001 into dr if [sr1] < [sr2] or the value 0x00000000 otherwise.

SLTI Dr, Sr1, SignImm					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001000	1	sr1	dr	imm	

Description: This instruction signed compares the value in register sr1 and the sign-extended value imm. It writes the value 0x00000001 into dr if [sr1] < [imm] or the value 0x00000000 otherwise.

SLTU Dr, Sr1, Sr2					
31 - 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001001	0	sr1	sr2	dr	unused

Description: This instruction unsigned compares the values in registers sr1 and sr2 and writes the value 0x00000001 into dr if [sr1] < [sr2] or the value 0x00000000 otherwise.

SLTUI Dr, Sr1, UnsImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001001	1	sr1	dr	imm	

Description: This instruction unsigned compares the value in register sr1 and the zero-extended value imm. It writes the value 0x00000001 into dr if [sr1] < [imm] or the value 0x00000000 otherwise.

SGE Dr, Sr1, Sr2					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001010	0	sr1	sr2	dr	unused

Description: This instruction signed compares the values in registers sr1 and sr2 and writes the value 0x00000001 into dr if [sr1] > = [sr2] or the value 0x00000000 otherwise.

SGEI Dr, Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001010	1	sr1	dr	imm	

Description: This instruction signed compares the value in register sr1 and the sign-extended value imm. It writes the value 0x00000001 into dr if [sr1] > = [imm] or the value 0x00000000 otherwise.

SGEU Dr, Sr1, Sr2					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001011	0	sr1	sr2	dr	unused

Description: This instruction unsigned compares the values in registers sr1 and sr2 and writes the value 0x00000001 into dr if [sr1] > = [sr2] or the value 0x00000000 otherwise.

SGEUI Dr, Sr1, UnsImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001011	1	sr1	dr	imm	

Description: This instruction unsigned compares the value in register sr1 and the zero-extended value imm. It writes the value 0x00000001 into dr if [sr1] > = [imm] or the value 0x00000000 otherwise.

SEQ Dr, Sr1, Sr2					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001100	0	sr1	sr2	dr	unused

Description: This instruction signed compares the values in registers sr1 and sr2 and writes the value 0x00000001 into dr if [sr1] = [sr2] or the value 0x00000000 otherwise.

SEQI Dr, Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0001100	1	sr1	dr	imm	

Description: This instruction signed compares the value in register sr1 and the sign-extended value imm. It writes the value 0x00000001 into dr if [sr1] = [imm] or the value 0x00000000 otherwise.

A.2.5 Load-Immediate Instructions

LDLI Dr, UnsImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0011100	1	unused	dr	imm	

Description: This instruction loads the immediate value into the lower 16 bits of the dr register, zeroing the upper 16 bits.

LDUI Dr, UnsImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0011101	1	unused	dr	imm	

Description: This instruction loads the immediate value into the upper 16 bits of the dr register, zeroing the lower 16 bits.

A.2.6 Memory Access Instructions

LDW Dr, RegSrc1					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010100	0	sr1	unused	dr	unused

Description: This instruction loads into register dr the value from the memory location which address is in register sr1.

STW Sr1, Sr2					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0010101	0	sr1	sr2	unused	unused

Description: This instruction stores the value in register sr2 into the memory location which address is in register sr1.

A.2.7 Control Transfer Instructions

BRT Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0011011	1	sr1	unused	imm	

Description: This instruction tests the value in register sr1 and jumps if it has the value 0x00000001 with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

BRF Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0011010	1	sr1	unused	imm	

Description: This instruction tests the value in register sr1 and jumps if it has the value 0x00000000 with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

BRLT Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0100000	1	sr1	dr	imm	

Description: This instruction signed compares the values in registers sr1 and dr and jumps if [sr1] < [dr] with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

BRLE Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0100001	1	sr1	dr	imm	

Description: This instruction signed compares the values in registers sr1 and dr and jumps if [sr1] ≤ [dr] with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

BREQ Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0100010	1	sr1	dr	imm	

Description: This instruction unsigned compares the values in registers sr1 and dr and jumps if [sr1] > [dr] with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

BRNE Sr1, SignImm					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0100011	1	sr1	dr	imm	

Description: This instruction unsigned compares the values in registers sr1 and dr and jumps if [sr1] ≠ [dr] with a one-instruction delay. The address of the target instruction is calculated by adding the sign-extended imm offset to the instruction's address.

JAL Dr, Sr1					
31 – 25	24	23 - 20	19 - 16	15 - 12	11 - 0
0011000	0	sr1	unused	dr	unused

Description: This instruction unconditionally jumps with a one-instruction delay to the target address in register sr1. The instruction's address plus 2 is saved into register dr.

A.2.8 MorphoSys Instructions

LDCTXT SreReg1, r/c#, r/c, context#, #contexts to be loaded									
31-26	25	24	23-20	19	18-16	15	14-11	10-8	7-0
6	1	1	4	1	3	1	4	3	8
100000	-	-	Sr1	-	r/c#	r/c	context#	---	# contexts to be loaded

- Sr1 : The starting address of external memory where the context configuration is stored (32-bit address)
- r/c# : used to control the starting cell in the context memory (0 to 7 in the horizontal direction)
- r/c : column context or row context (0—column, 1 – row)
- context# : starting context (0 to 15)
- --- : don't care bits.
- #contexts to be loaded: specify how many contexts to be loaded through DMA.

Description: Load Context
NOTE: During DMAC loading context, it increase r/c# first, then increase the context#.

LDFB Sr1, bank, set#, #words							
31-26	25	24	23-20	19-11	10	9	8-0
6	1	1	4	9	1	1	9
100010	-	-	Sr1	---------	Bank	Set#	#words number

- Sr1 : The starting address of external memory where the data is stored.(32bit address);
- bank : which bank. 0—bank A. 1—bank B;

- set#		: set number 0/1;
- #words number		: 1 word = 32bits

Description: load Frame Buffer.

NOTE: When load/store data to/from frame buffer, it always starts from the beginning. It is different from the mechanism of context memory, which can start anywhere. One bank has 64 rows, each row has 2 words (64 bits).

STFB Sr1, bank, set#, #words							
31-26	25	24	23-20	19-11	10	9	8-0
6	1	1	4	9	1	1	9
100011	-	-	Sr1	---------	Bank	Set#	#words number

- Sr1		: the starting address where the data want to be stored
- bank		: the same as LDFB
- set#		: the same as LDFB
- #words number		: the same as LDFB.

Description: store Frame Buffer data to memory.

SBCB b_all, b_row_col, r/c, context#, bank, set#, bank_addr									
31-26	25-21	20	19	18-16	15	14-11	10	9	8-0
6	5	1	1	3	1	4	1	1	9
110100	-----	b_all	-	b_row_col	r/c	Context #	bank	Set#	bank_addr

- b_all		: specify whether the whole RC Array (8 by 8) is active or only one row/column RCs are active; 1= ALL of the RCs are active. 0-- only one row/column RCs are active.
- B_row_col	: If the b_all =0, then this field specifies which row/column is active;
- r/c		: specify the context memory perform row context broadcast or column context broadcast: 1 for row context broadcast and 0 for column context broadcast
- Bank		: specify which bank in Frame buffer
- Set		: specify which set
- Bank_addr	: provide the direct frame buffer address (9 bits).

Description: Single Bank Context Broadcast. Fetch only one operand from one of the Frame buffer banks.

NOTE: Since each bank in the Frame buffer has the capacity 64 by 8 bytes, 6 bits to specify which row (2^6) and the other 3 bits to specify the starting word in that line. The important feature of the frame buffer is that it always fetches the consecutive eight words even though sometimes it will wrap around to the next row.

DBCBC Sr1, bank_B_addr_base, b_all, r/c#, context#, set#, bank_A_addr							
31-26	25	24	23-20	19-16	15-12	11-9	8-0
6	1	1	4	4	4	3	9
111100	set	b_all	Sr1	base_bankB	Context#	r/c#	bank_A_addr

- Set : Specify frame buffer set 0 or set 1
- b_all : the same as SBCB
- Sr1 : specify the register number
- Base_bankB : directly provide the base address for bank B. This 4 bit, together with the 5 bits from Sr1, provide the Bank B address (9 bits)
- Context # : the same meaning as SBCB
- r/c# : if b_all =0, then specify which column RCs is active.

Description: Double bank ***column-wise*** context broadcast. To one column RC (8 RCs), 8 bytes will be from bank A, and the other 8 bytes will be from Bank B. This 16-byte data is interleaved as AB. The 128-bit data bus is split into 8 segments. Each segment bus is connected to all of the RCs in one row. When during column context broadcast, all of the RCs in one column perform the same functionality. The same scheme is applied to the row context broadcast.

DBCBR Sr1, bank_B_addr_base, b_all, r/c#, context#, set#, bank_A_addr							
31-26	25	24	23-20	19-16	15-12	11-9	8-0
6	1	1	4	4	4	3	9
111101	set	b_all	Sr1	base_bankB	Context#	r/c#	bank_A_addr

- Set : Specify frame buffer set 0 or set 1;
- b_all : the same as SBCB;
- Sr1 : specify the register number.
- Base_bankB : directly provide the base address for bank B. This 4 bits, together with the 5 bits from Sr1, provide the Bank B address (9 bits).
- Context # : the same meaning as SBCB;
- r/c# : if b_all =0, then specify which column RCs is active.

Description: Double bank ***row-wise*** context broadcast. It has the same meaning as DBCBC. All of the RCs in one row will perform the same functionality. Why do we have these two separate instructions with almost the same meaning? It is because we don't have enough bits if we don't use op-code field.

CBCAST b_all, b_row_col, r/c, context#							
31-26	25-21	20	19	18-16	15	14-11	10-0
6	5	1	1	3	1	4	11
111000	-----	b_all	-	b_row_col	r/c	Context#	-----------

- b_all : 1= all of the RC is active. 0= only one row/column RC is active
- b_row_col : if B_all =0, then b_row_col specify which row/column is active
- r/c : 0= column context broadcast. 1= row context broadcast.

Description: Context broadcast using internal data only.

WFBI r/c#, r/c, bank, set#, bank_addr							
31-26	25-19	18-16	15	14-11	10	9	8-0
6	7	3	1	4	1	1	9
101000	-------	r/c#	r/c	----	bank	set#	bank_addr

- r/c # : specify the data in column number (r/c #) will be write back to Frame buffer
- r/c : you can ignore it
- Bank : specify which bank in Frame Buffer
- Set : specify which set
- Bank_addr : directly frame buffer address (9 bits).

Description: Write data to Frame Buffer with the immediate Frame Buffer address.

WFB Sr1, r/c#, r/c, bank, set#									
31-26	25-24	23-20	19	18-16	15	14-11	10	9	8-0
6	2	4	1	3	1	4	1	1	9
101001	--	Sr1	-	r/c#	r/c	----	bank	set#	---------

- Sr1 : specify the register number. It is the Frame Buffer address.
- r/c # : specify the data in column number (r/c #) to write back to Frame buffer.
- r/c : you can ignore it.
- Bank : specify which bank in Frame Buffer;
- Set# : specify which set;

Description: Write data to Frame Buffer with the address coming from the register file.

RCRISC Dest, col#				
31-26	25-19	18-16	15-12	11-0
6	7	3	4	12
100100	-------	col#	Dest	------------

- col# : specify which RC (0 to 7) in the first row will write data to TinyRISC.
- Dest : the destination register.

Description: Write one of the data in the first row of the RC Array back to the TinyRISC.

Appendix B

mLoad

The following map.pl is the lookup table used in the mLoad to convert the symbolic context configuration into binary context configuration.

B.1 Map.pl

```
# MAP FIELDS:
# field | 7  | 6  | 5    | 4    | 3    |   2   |  1    |    0    |
# -----------------------------------------------------------------------
# bit # | 31 | 30 |29-28 | 27   | 26-23| 22-19 | 18-16 |   15-0  |
# -----------------------------------------------------------------------
# define|WE  | WR | RF   |shift | SH#  |  muxa | muxb  | alu_constant |

# We:  Write Express Lane
# WR:  Write Register File.
# RF:  Register File I.D.
# shift:  Whether Right or Left Shift.
# SH#: Right or Left Shift Number.
# muxa: For MUXA.
# muxb: For MUXB.
# alu_constant:  alu control signal and constant.

print "Loading " . __FILE__ . "\n";
$numOfRCs  = 16;  # Number of context memory CELL.
$numOfctxs = 16;  # Number of contexts
$numRegs   = 4;         # Number of RX registers
$sp = '';         # Output format spacer.

# For the MUX involving operand A:
%amux = (
 'I',       '0000',           # Data Bus
 'L',       '0001',           # Left
 'M',       '0010',           # Middle
 'R',       '0011',           # Right
 'T',       '0100',           # Top
 'C',       '0101',           # Center
 'B',       '0110',           # Bottom
 'VE',      '0111',           # From Vertical Express Lane
 'HE',      '1000',           # From Horizontal Express Lane
 'XQ',      '1001',           # Cross Quadrant

 'r0',      '1100',           # Register read access
 'r1',      '1101',           # Register read access
 'r2',      '1110',           # Register read access
 'r3',      '1111',           # Register read access
 'def',     '0000',           # The default value;
);
```

```
# For the MUX involving operand B:
%bmux = (
 'I',       '000',            # Data Bus
 'U',       '001',            # Up
 'D',       '010',            # Down
 'L',       '011',            # Left
 'r0',      '100',            # Register access
 'r1',      '101',            # Register access
 'r2',      '110',            # Register access
 'r3',      '111',            # Register access
 'const',   'err',            # Currently no const possible here
 'def',     '000',            # The default value;
);


# For the shifter
%shift = (
'default',  '0',  # No Shift
'NS',       '0',
'LSL',      '0',  # Shift Left      'LSL'
'LSR',      '1',  # Shift Right     'LSR'
);

# For the ALU Operations
# We have 3 operands which we will call <A>, <B>, <C>
# <Out> means the output of the ALU unit

%functions = (
# Note 'C' as the first letter will hint program to expect a constant
#     right after the mnemonic name.

# USing Constant

'CLOAD',       '0000',   # <out> = SIGN_EXT(Constant) For Constant Load
'COR',         '0001',   # <out> = <A> OR  Constant
'CAND',        '0010',   # <out> = <A> AND Constant
'CXOR',        '0011',   # <out> = <A> XOR Constant
'CADD',        '0100',   # <out> = <A>  +  Constant
'CSUB',        '0110',   # <out> = <A>  -  Constant
'CMUL',        '1001',   # <out> = <A>  *  Constant
'CMULBADD',    '1101',   # <out(t+1)> = <A> * Constant + B For FIR
'CMULOADD',    '1100',   # <out(t+1)> = <A> * Constant + <out> For DCT
'CMULSUB',     '1110',   # <out(t+1)> = <A> * Constant - <out>

# Don't use constant

'BYPASS',      '11110000',      # Out = <A> and set flag based on Sign.
'OR',          '11110001',      # Out = <A> OR <B>
'AND',         '11110010',      # Out = <A> AND <B>
'XOR',         '11110011',      # Out = <A> XOR <B>
'ADD',         '11110100',      # Out = <A> + <B>
'SUB',         '11110110',      # Out = <A> - <B>
'SUBBA',       '11110111',      # Out = <B> - <A>

'ADDSUBF',     '11110101',      # add or sub is based on the Flag
                                # IF Flag = 0 then A + B
                                # ELSE  A - B
'BTM',         '11111010',      # AND the Right 8 bits and
                                #  Count 1 in the result.

'ABSD',        '11111110',      # Out = |A-B|+Previous
'KEEP',        '11111000',      # <Out>(t+1) = <Out>(t) FOR ALU Idle
'ROUND',       '11111100',      # <Out> = MSB 16 bit + round.
                                # For Rounding, needs righ shift 12.
'RESET',       '11111111',      # RESET the output register.
'default',     '0000',          # OR is default operation.
);

#----------------------------------------------------------------------

# Defaults and initial values
sub resetDefaults{
$const        =  0;     # The default constant value
$bmuxCtx      =  $bmux{'def'};
$amuxCtx      =  $amux{'def'};
$writeRF_EN   = '0';
$writeRF_ID   = '00';
$sftCtlBit    = '0';    # Direction of left shift
$sftValBits   = '0000'; # Amount of port shift
$aluBits      = $functions{'default'};
$writexpress  = '0';
$alu_domain   = '0000000000000000';
}

#----------------------------------------------------------------------
#
# This routing is used to set the order and content of the output
# context string.

sub outStr { # String context output string in correct order
 local($s);
 $s =   $writexpress                    . $sp .
        $writeRF_EN                     . $sp .
        $writeRF_ID                     . $sp .
        $sftCtlBit . $sftValBits        . $sp .
        $amuxCtx                        . $sp .
        $bmuxCtx                        . $sp .
        $alu_domain;
 return $s;
}
```

Appendix C

DCT and Motion Estimation

This appendix provides the context configurations and the assembly codes for 2-Dimensional DCT and Motion Estimation.

C.1 DCT Context Configuration

```
commentOutput;

set 0,0 BYPASS I I >0;
set 1,0 BYPASS I I >0;
set 2,0 BYPASS I I >0;
set 3,0 BYPASS I I >0;
set 4,0 BYPASS I I >0;
set 5,0 BYPASS I I >0;
set 6,0 BYPASS I I >0;
set 7,0 BYPASS I I >0;

set 0,1 KEEP I I ;
set 1,1 CLOAD!0x004 def def >2 ;
set 2,1 ADD HE r0 LSL 2 >0 WE;
set 3,1 ADD XQ r0 LSL 2 >0;
set 4,1 SUB XQ r0 LSL 2 >0;
set 5,1 SUB HE r0 LSL 2 >0 WE;
set 6,1 CLOAD!0x004 def def >2 ;
set 7,1 KEEP I I ;

set 0,2 ADD HE r0 LSL 2 >0 WE;
set 1,2 BYPASS r0 def >0;
set 2,2 CLOAD!0x004 def def >2 ;
set 3,2 CLOAD!0x004 def def >2 ;
set 4,2 CLOAD!0x004 def def >2 ;
set 5,2 CLOAD!0x004 def def >2 ;
set 6,2 BYPASS r0 def >0;
set 7,2 SUB HE r0 LSL 2 >0 WE;

set 0,3 KEEP I I >3;
set 1,3 ADD HE r0 LSL 2 >0 WE;
set 2,3 BYPASS r0 def >0;
set 3,3 BYPASS r0 def >0;
set 4,3 BYPASS r0 def >0;
set 5,3 BYPASS r0 def >0;
set 6,3 SUB HE r0  LSL 2 >0 WE;
set 7,3 KEEP I I >3;

set 0,4 ADD R r0 >0;
set 1,4 ADD M r0 >0;
set 2,4 SUB M r0 >0;
set 3,4 SUB L r0 >0;
set 4,4 KEEP I I  >3;
set 5,4 SUB M r0 >0;
set 6,4 ADD M r0 >0;
set 7,4 KEEP I I  >0;

set 0,5 ADD L r0 >0 ;
set 1,5 SUB L r0 >0;
set 2,5 CMUL!0x1D9 R def >1;
set 3,5 CMUL!0x1D9 R def >1;
set 4,5 KEEP I I >3;
set 5,5 CMUL!0x16A r0 def LSR 9 >1;
set 6,5 CMUL!0x16A r0 def LSR 9 >1;
set 7,5 KEEP I I >3;

set 0,6 CMUL!0x16A r0 def LSR 9 >0;
set 1,6 CMUL!0x16A r0 def LSR 9 >0;
set 2,6 CMULOADD!0x0C3 r0 def LSR 9 >0;
set 3,6 CMULSUB!0x0C3 r0 def LSR 9 >0;
set 4,6 ADD L r0 >0;
set 5,6 SUB L r1 >0;
set 6,6 SUB R r1 >0;
set 7,6 ADD R r0 >0;

set 0,7 KEEP I I >3;
set 1,7 KEEP I I >3;
set 2,7 KEEP I I >3;
set 3,7 KEEP I I >3;
set 4,7 CMUL!0x1F6 R def >1;
set 5,7 BYPASS M def >1;
set 6,7 CMUL!0x1AA M def >1;
set 7,7 CMUL!0x1F6 L def >1;


set 0,8 KEEP I I >3;
set 1,8 KEEP I I >3;
set 2,8 KEEP I I >3;
set 3,8 KEEP I I >3;
set 4,8 CMULOADD!0x64 r0 def LSR 9 >0;
set 5,8 CMUL!0x11C r0 def >0;
set 6,8 CMULOADD!0x11C r0 def LSR 9 >0;
set 7,8 CMULSUB!0x64 r0 def LSR 9 >0;

set 0,9 CLOAD!0x004 def def >2 ;
set 1,9 BYPASS HE def >0 WE;
set 2,9 KEEP I I >3;
set 3,9 KEEP I I >3;
set 4,9 BYPASS HE def >0 WE;
set 5,9 CMULSUB!0x1AA r1 def LSR 9 >0;
set 6,9 KEEP I I >3;
set 7,9 CLOAD!0x004 def def >2 ;

set 0,10 BYPASS r0 def >0;
set 1,10 KEEP I I >3 ;
set 2,10 KEEP I I >3 ;
set 3,10 BYPASS HE def >0 WE;
set 4,10 KEEP I I >3 ;
set 5,10 BYPASS M def >0 WE;
set 6,10 BYPASS HE def >0;
set 7,10 BYPASS r0 def >0;


set 8,0 ADD VE r0 LSR 1 >0 WE;
set 9,0 KEEP I I >3;
set 10,0 KEEP I I >3;
set 11,0 ADD r0 D LSR 1 >0;
set 12,0 SUBBA r0 U LSR 1 >0;
set 13,0 KEEP I I >3;
set 14,0 KEEP I I >3;
set 15,0 SUB VE r0 LSR 1 >0 WE;

set 8,1 KEEP I I >3;
set 9,1 ADD VE r0 LSR 1 >0 WE;
set 10,1 KEEP I I >3;
set 11,1 KEEP I I >3;
set 12,1 KEEP I I >3;
set 13,1 KEEP I I >3;
set 14,1 SUB VE r0 LSR 1 >0 WE;
set 15,1 KEEP I I >3;

set 8,2 KEEP I I >3;
set 9,2 KEEP I I >3;
set 10,2 ADD VE r0 LSR 1 >0 WE;
set 11,2 KEEP I I >3;
set 12,2 KEEP I I >3;
set 13,2 SUB VE r0 LSR 1 >0 WE;
set 14,2 KEEP I I >3;
set 15,2 KEEP I I >3;

set 8,3 ADD B r0 >0;
set 9,3 ADD C r0 >0;
set 10,3 SUB C r0 >0;
set 11,3 SUB T r0 >0;
set 12,3 KEEP I I >3;
set 13,3 SUB C r0 >0;
set 14,3 ADD C r0 >0;
set 15,3 KEEP I I >3;

set 8,4 ADD T r0 >0 ;
set 9,4 SUB T r0 >0;
set 10,4 CMUL!0x1D9 B def >1;
set 11,4 CMUL!0x1D9 B def >1;
set 12,4 KEEP I I >3;
set 13,4 CMUL!0x16A r0 def LSR 9 >1;
set 14,4 CMUL!0x16A r0 def LSR 9 >1;
set 15,4 KEEP I I;

set 8,5 CMUL!0x16A r0 def LSR 9 >0 ;
set 9,5 CMUL!0x16A r0 def LSR 9 >0;
set 10,5 CMULOADD!0x0C3 r0 def LSR 9 >0;
set 11,5 CMULSUB!0x0C3 r0 def LSR 9 >0;
set 12,5 ADD T r0 >0;
set 13,5 SUB T r1 >0;
set 14,5 SUB B r1 >0;
set 15,5 ADD B r0 >0;
set 8,6 BYPASS r0 def >0;
set 9,6 BYPASS r0 def >0;
set 10,6 BYPASS r0 def >0;
set 11,6 BYPASS r0 def >0;
set 12,6 CMUL!0x1F6  B def >1;
set 13,6 BYPASS C def >1;
set 14,6 CMUL!0x1AA C def >1;
set 15,6 CMUL!0x1F6 T def >1;

set 8,7 ADDSUBF r0 r2 LSR 3 >0;
set 9,7 ADDSUBF r0 r2 LSR 3 >0;
set 10,7 ADDSUBF r0 r2 LSR 3 >0;
set 11,7 ADDSUBF r0 r2 LSR 3 >0;
set 12,7 CMULOADD!0x64 r0 def LSR 9 >0;
set 13,7 CMUL!0x11C r0 def >0;
set 14,7 CMULOADD!0x11C r0 def LSR 9 >0;
set 15,7 CMULSUB!0x64 r0 def LSR 9 >0 ;

set 8,8 KEEP I I >3;
set 9,8 BYPASS VE def >0 WE;
set 10,8 KEEP I I >3;
set 11,8 KEEP I I >3;
set 12,8 BYPASS VE def >0 WE;
set 13,8 CMULSUB!0x1AA r1 def LSR 9 >0;
set 14,8 KEEP I I >3;
set 15,8 BYPASS r0 def >0;

set 8,9 KEEP I I >3 ;
set 9,9 ADDSUBF r0 r2 LSR 3 >0;
set 10,9 KEEP I I >3 ;
set 11,9 BYPASS VE def >0 WE;
set 12,9 KEEP I I >3 ;
set 13,9 BYPASS C def >0 WE;
set 14,9 BYPASS VE def >0;
set 15,9 ADDSUBF r0 r2 LSR 3 >0;

set 8,10 KEEP I I >3 ;
set 9,10 KEEP I I >3 ;
set 10,10 KEEP I I >3 ;
set 11,10 ADDSUBF r0 r2 LSR 3 >0;
set 12,10 KEEP I I >3 ;
set 13,10 ADDSUBF r0 r2 LSR 3 >0;
set 14,10 KEEP I I >3 ;
set 15,10 KEEP I I >3 ;

writeFile "dct_out.ctx";
```

C.2 DCT Assembly Code

```
        .set    noreorder
        .set    noat
        .file   2 "me.c"
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 23
        .ent    main
main:
      nop
      lui $1, 0x0004
      nop

# Load Frame buffer at Bank A, set 0.
      ldfb $1, 0, 0, 16
      ldi  $2, 0x0000
      ldi  $3, 0x0080
      ldi  $10, 16
Delay1.DONE:
      subi $10, $10, 4
      nop
      brle $0, $10, Delay1.DONE
      nop

# Load Col context
      ldctxt $2, 0, 0, 0, 128
      ldi    $10, 128
Delay2.DONE:
      subi $10, $10, 4
      nop
      brle $0, $10, Delay2.DONE
      nop

      nop
      nop
      nop

# Load Row context
      ldctxt $3, 0, 1, 0, 128
      ldi  $10, 128
Delay3.DONE:
      subi $10, $10, 4
      nop
      brle $0, $10, Delay3.DONE
      nop

      nop
      nop
      nop
      nop

# Now begin to 2-D DCT operation.
#            b_all b_r_c r/c ctx BK set bank_addr
#            b_all = 1 all, b_all=0 mask
      sbcb 0,    0,    0,  0,  0, 0,  0
      sbcb 0,    1,    0,  0,  0, 0,  8
      sbcb 0,    2,    0,  0,  0, 0,  16
      sbcb 0,    3,    0,  0,  0, 0,  24
      sbcb 0,    4,    0,  0,  0, 0,  32
      sbcb 0,    5,    0,  0,  0, 0,  40
      sbcb 0,    6,    0,  0,  0, 0,  48
      sbcb 0,    7,    0,  0,  0, 0,  56

#  COL Broadcasting
#              b_all b_r_c r/c ctx
      cbcast 1,    0,    0,   1
      cbcast 1,    0,    0,   2
      cbcast 1,    0,    0,   3
      cbcast 1,    0,    0,   4
      cbcast 1,    0,    0,   5
      cbcast 1,    0,    0,   6
      cbcast 1,    0,    0,   7
      cbcast 1,    0,    0,   8
      cbcast 1,    0,    0,   9
      cbcast 1,    0,    0,   10
#     cbcast 1,    0,    0,   11

#  ROW Broadcasting
      cbcast 1,    0,    1,   0
      cbcast 1,    0,    1,   1
      cbcast 1,    0,    1,   2
      cbcast 1,    0,    1,   3
      cbcast 1,    0,    1,   4
      cbcast 1,    0,    1,   5
      cbcast 1,    0,    1,   6
      cbcast 1,    0,    1,   7
      cbcast 1,    0,    1,   8
      cbcast 1,    0,    1,   9
      cbcast 1,    0,    1,   10

# Nop is necessary before writing out
      nop
# Write DATA back to FB
#         r/c #, r/c, bank, set, bank_addr

      wfbi 0,     0,    0,    0, 0
      wfbi 1,     0,    0,    0, 8
      wfbi 2,     0,    0,    0, 16
      wfbi 3,     0,    0,    0, 24
      wfbi 4,     0,    0,    0, 32
      wfbi 5,     0,    0,    0, 40
      wfbi 6,     0,    0,    0, 48
      wfbi 7,     0,    0,    0, 56

#   store the data back to sram memory
#           sr1,   bank, set#, words#
# Load Frame buffer at Bank A, set 0.
      stfb  $1,     0,    0,   16
      nop
      nop
      ldi  $10, 16
Delay4.DONE:
      subi $10, $10, 4
      nop
      brle $0, $10, Delay4.DONE
      nop
```

```
#   FINISH
      nop
      nop
      .end main
```

C.3 Motion Estimation Context Configuration

```
#commentOutput;

set 0,0 ABSD  I I LSL 0;
set 1,0 ADD  r0 D LSL 0;
set 2,0 ADD  r0 D LSL 0;
set 3,0 KEEP  I I LSL 0;
set 4,0 BYPASS I I LSL 0;
set 5,0 BYPASS L I LSL 0;
set 6,0 RESET  I I  LSL 0;
set 7,0 BYPASS  R I LSL 0>0;


set 0,1 ABSD  I I LSL 0;
set 1,1 ADD  r0 D LSL 0;
set 2,1 ADD  r0 D LSL 0;
set 3,1 ABSD  XQ I LSL 0;
set 4,1 BYPASS I I LSL 0;
set 5,1 BYPASS L I LSL 0;
set 6,1 KEEP I I  LSL 0;
set 7,1 ADD  r0 D LSL 0;

set 0,2 ABSD  I I LSL 0;
set 1,2 ADD  r0 D LSL 0;
set 2,2 ADD  r0 D LSL 0;
set 3,2 ABSD  XQ I LSL 0;
set 4,2 BYPASS I I LSL 0;
set 5,2 BYPASS L I LSL 0;
set 6,2 ABSD  M I LSL 0;
set 7,2 ADD  r0 D LSL 0;

set 0,3 KEEP I I  LSL 0;
set 1,3 ADD  r0 D LSL 0;
set 2,3 ADD  r0 D LSL 0;
set 3,3 ABSD  XQ I LSL 0;
set 4,3 BYPASS I I LSL 0;
set 5,3 BYPASS L I LSL 0;
set 6,3 ABSD  M I LSL 0;
set 7,3 ADD  r0 D LSL 0;

set 0,4 KEEP I I  LSL 0;
set 1,4 ADD  r0 D LSL 0;
set 2,4 ADD  r0 D LSL 0;
set 3,4 KEEP I I  LSL 0;
set 4,4 BYPASS I I LSL 0;
set 5,4 BYPASS L I LSL 0;
set 6,4 ABSD  M I LSL 0;
set 7,4 ADD  r0 D LSL 0;
set 0,5 RESET I I LSL 0;
set 1,5 BYPASS  L I LSL 0 >0;
set 2,5 ADD  r0 D LSL 0;
set 3,5 ABSD XQ I  LSL 0;
set 4,5 BYPASS I I LSL 0;
set 5,5 BYPASS L I LSL 0;
set 6,5 ABSD  M I LSL 0;
set 7,5 ADD  r0 D LSL 0;

set 0,6 KEEP I I  LSL 0;
set 1,6 ADD  r0 D LSL 0;
set 2,6 BYPASS R I LSL 0 > 0;
set 3,6 RESET I I LSL 0;
set 4,6 BYPASS I I LSL 0;
set 5,6 BYPASS L I LSL 0;
set 6,6 ABSD  M I LSL 0;
set 7,6 ADD  r0 D LSL 0;

set 0,7 ABSD  I I LSL 0;
set 1,7 ADD  r0 D LSL 0;
set 2,7 ADD  r0 D LSL 0;
set 3,7 KEEP I I  LSL 0;
set 4,7 BYPASS I I LSL 0;
set 5,7 BYPASS L I LSL 0;
set 6,7 KEEP I I LSL 0;
set 7,7 ADD  r0 D LSL 0;

set 0,8 RESET def def;
set 1,8 RESET def def;
set 2,8 RESET def def;
set 3,8 RESET def def;
set 4,8 RESET def def;
set 5,8 RESET def def;
set 6,8 RESET def def;
set 7,8 RESET def def;

writeFile "me_out.ctx";
```

C.4 Part of Motion Estimation Assembly Code

```
# pseudo-code for the motion estimation
#   The definition of the register.
#   $0 = 0;
#   $1 = H;
#   $2 = V;   it is not used any more. Because, it is unrolled.
#   $3 = mini_value;
#   $4 = motion vector X;
#   $5 = motion vector Y;
#   $6 = First income data;
#   $7 = Second income data;
#   $8 = Third income data;
#   $9 = 15;
#   $10 = V + 8;
# 16x16 current block is in bank A, 32x32 search area is in bank B.
# For 16X16 block, the two dimension is [15:0-15:0].
# For 32X32 search area, the two dimension is [31:0-31:0]
# intial $3 = 7FFF-FFFF( the biggest positive integer).
      .set  noreorder
      .set  noat
      .file 2 "me.c"
      .text
      .align      4
      .align      4
      .globl      main
      .loc  2 23
      .ent  main
main:
      nop
# Main memory address size is 0-7FFFF, word addressable. top half is
for
# Image data, low half for context
        lui $1, 0x0004
      nop
# Load current data to Bank A set 0
      ldfb $1, 0, 0, 64
        lui  $2, 0x0006
      addi  $3, $2, 0x0080
        ldi  $10, 64
Delay1.DONE:
        subi $10, $10, 4
        nop
        brle $0, $10, Delay1.DONE
        nop

# Load current data to Bank A set 1
      ldfb $1, 0, 1, 64
        ldi  $10, 64
Delay2.DONE:
        subi $10, $10, 4
        nop
        brle $0, $10, Delay2.DONE
        nop

# Load reference data to Bank B, set 0
      ldfb $2, 1, 0, 128
        ldi  $10, 128
Delay3.DONE:
        subi $10, $10, 4
        nop
        brle $0, $10, Delay3.DONE
        nop

# Load reference data to Bank B, set 1
      ldfb $3, 1, 1, 128
        ldi  $10, 128
Delay4.DONE:
        subi $10, $10, 4
        nop
        brle $0, $10, Delay4.DONE
        nop
```

```
# Load context to Context memory.
      ldctxt $0, 0, 0, 0, 128
        ldi  $10, 128
Delay5.DONE:
        subi $10, $10, 4
        nop
        brle $0, $10, Delay5.DONE
        nop

# FINISH LOAD NECESSAY DATA AND CONTEXT OT FB AND CONTEXT MEMORY
# THE FOLLOWING INSTRUCTION BEGIN TO WORK.

# CLEAR DATA.
#              b_all b_r_c r/c ctx
        cbcast 1,    0,    0,   8

        add $1, $0, $0
        ldi $9, 16
        lui $3, 0x7FFF
      ori $3, $3, 0xFFFF
#          $3 = 0x7FFFFFFF biggest positive number.
main.OUT:
      brle $9, $1, main.DONE
        addi $10, $1,  8
# INNER loop 1
#             reg,set,all,base,ctx,r/c,BB
      dbcbc $1,  0,  1,   0,  0,  0, 0
      dbcbc $1,  0,  1,   1,  1,  0, 16
      dbcbc $1,  0,  1,   2,  2,  0, 32
      dbcbc $1,  0,  1,   3,  2,  0, 48
      dbcbc $1,  0,  1,   4,  2,  0, 64
      dbcbc $1,  0,  1,   5,  2,  0, 80
      dbcbc $1,  0,  1,   6,  2,  0, 96
      dbcbc $1,  0,  1,   7,  2,  0, 112
      dbcbc $1,  0,  1,   8,  2,  0, 128
      dbcbc $1,  0,  1,   9,  2,  0, 144
      dbcbc $1,  0,  1,   10,  2,  0, 160
      dbcbc $1,  0,  1,   11,  2,  0, 176
      dbcbc $1,  0,  1,   12,  2,  0, 192
      dbcbc $1,  0,  1,   13,  2,  0, 208
      dbcbc $1,  0,  1,   14,  2,  0, 224
      dbcbc $1,  0,  1,   15,  2,  0, 240
      dbcbc $1,  1,  1,   0,  3,  0, 0
      dbcbc $1,  1,  1,   1,  4,  0, 0
   bre $1, $0, main.L1.EXIT
        nop
      rcrisc $6, 1
        rcrisc $7, 2
        nop
      brle $7, $6, main.L1.ELSE1
              nop
              brle $3, $6, main.L1.EXIT
                  nop
                  add $3, $6, $0
# it is the Motion Vector of Loop 6 in  the previous Iteration.
                  subi $4, $1, 1
                  addi $5, $0, 15
                  brf $0, main.L1.EXIT
                  nop
main.L1.ELSE1:
              brle $3, $7, main.L1.EXIT
                  nop
                  add $3, $7,  $0
                  subi  $4, $1, 1
                  addi $5, $0, 16
main.L1.EXIT:
      dbcbc $10,  0,  1,   0,  7,  0, 8
      dbcbc $10,  0,  1,   1,  1,  0, 24
      dbcbc $10,  0,  1,   2,  2,  0, 40
      dbcbc $10,  0,  1,   3,  2,  0, 56
      dbcbc $10,  0,  1,   4,  2,  0, 72
      dbcbc $10,  0,  1,   5,  2,  0, 88
      dbcbc $10,  0,  1,   6,  2,  0, 104
      dbcbc $10,  0,  1,   7,  2,  0, 120
      dbcbc $10,  0,  1,   8,  2,  0, 136
      dbcbc $10,  0,  1,   9,  2,  0, 152
      dbcbc $10,  0,  1,   10,  2,  0, 168
      dbcbc $10,  0,  1,   11,  2,  0, 184
      dbcbc $10,  0,  1,   12,  2,  0, 200
      dbcbc $10,  0,  1,   13,  2,  0, 216
      dbcbc $10,  0,  1,   14,  2,  0, 232
      dbcbc $10,  0,  1,   15,  2,  0, 248
      dbcbc $10,  1,  1,   0,  5,  0, 0
      dbcbc $10,  1,  1,   1,  6,  0, 0
# INNER loop 2
      dbcbc $1,  0,  1,   3,  0,  0, 0
      dbcbc $1,  0,  1,   4,  1,  0, 16
      dbcbc $1,  0,  1,   5,  2,  0, 32
      dbcbc $1,  0,  1,   6,  2,  0, 48
      dbcbc $1,  0,  1,   7,  2,  0, 64
      dbcbc $1,  0,  1,   8,  2,  0, 80
      dbcbc $1,  0,  1,   9,  2,  0, 96
      dbcbc $1,  0,  1,   10,  2,  0, 112
      dbcbc $1,  0,  1,   11,  2,  0, 128
      dbcbc $1,  0,  1,   12,  2,  0, 144
      dbcbc $1,  0,  1,   13,  2,  0, 160
      dbcbc $1,  0,  1,   14,  2,  0, 176
      dbcbc $1,  0,  1,   15,  2,  0, 192
      dbcbc $1,  1,  1,   0,  2,  0, 208
      dbcbc $1,  1,  1,   1,  2,  0, 224
      dbcbc $1,  1,  1,   2,  2,  0, 240
      dbcbc $1,  1,  1,   3,  3,  0, 0
      dbcbc $1,  1,  1,   4,  4,  0, 0
      rcrisc $6, 1
        rcrisc $7, 2
        rcrisc $8, 7
        brle $7, $6, main.L2.ELSE1
              nop
              brle, $8, $6, main.L2.ELSE2
                    nop
                     brle $3, $6, main.L2.EXIT
                         nop
                          add $3, $6, $0
                         add $4, $1, $0
                         addi $5, $0, 0
                         brf $0, main.L2.EXIT
                         nop
main.L2.ELSE2:
                  brle $3, $8, main.L2.EXIT
                          nop
                          add $3, $8,  $0
                          add  $4, $1,  $0
                          addi $5, $0,  2
                          brf $0, main.L2.EXIT
                          nop
main.L2.ELSE1:
           brle $8, $7, main.L2.ELSE3
                      nop
                       brle $3, $7, main.L2.EXIT
                            nop
                             add $3, $7,  $0
                             add  $4, $1,  $0
                             addi $5, $0,  1
                             brf $0, main.L2.EXIT
                             nop
main.L2.ELSE3:
                  brle $3, $8, main.L2.EXIT
                          nop
                           add $3, $8, $0
                           add  $4, $1, $0
                           addi $5, $0,  2
main.L2.EXIT:
      dbcbc $10,  0,  1,   3,  7,  0, 8
      dbcbc $10,  0,  1,   4,  1,  0, 24
      dbcbc $10,  0,  1,   5,  2,  0, 40
      dbcbc $10,  0,  1,   6,  2,  0, 56
      dbcbc $10,  0,  1,   7,  2,  0, 72
      dbcbc $10,  0,  1,   8,  2,  0, 88
      dbcbc $10,  0,  1,   9,  2,  0, 104
      dbcbc $10,  0,  1,   10,  2,  0, 120
      dbcbc $10,  0,  1,   11,  2,  0, 136
      dbcbc $10,  0,  1,   12,  2,  0, 152
      dbcbc $10,  0,  1,   13,  2,  0, 168
      dbcbc $10,  0,  1,   14,  2,  0, 184
      dbcbc $10,  0,  1,   15,  2,  0, 200
      dbcbc $10,  1,  1,   0,  2,  0, 216
      dbcbc $10,  1,  1,   1,  2,  0, 232
      dbcbc $10,  1,  1,   2,  2,  0, 248
      dbcbc $10,  1,  1,   3,  5,  0, 0
      dbcbc $10,  1,  1,   4,  6,  0, 0

      ...
      ...

      nop
      .end  main
```

Appendix D

Test Vector Format of ProTest

The ProTest format is described next. This information can be used to translate the test vectors to another format. In ProTest, the test vector file begins with a specification of the external signals for the DUT. Each entry in the signal specification section has the following format:

```
  signal_name  low_bus_index   high_bus_index {DUT-inp | DUT-out | DUT-
                                   bidir}
```

where

- `signal_name` is the signal bus label (all external signals are considered to be buses).
- `low_bus_index` and `high_bus_index` indicate the size of a signal bus, by specifying the lowest and the highest signal index, respectively. In particular, a single signal is specified as "1 1"
- `DUT-inp`, `DUT-out` and `DUT-bidir` indicate, respectively, whether the signal bus is an input, output or bi-directional.

The test vectors appear right after the signal specifications. Test vectors are listed on a cycle-by-cycle basis. The general format of a test vector is as follows:

```
C# DUTin … [{I DUT_in_bidir} | X]… {O DUTout}… [{X | O
DUT_out_bidir}] …
```

where

- `C#` is the clock cycle number, numbered from 1.
- `DUTin` specify a logic value for an input signal. The input values should appear in the same order as the input signals were specified in the first section of the file.
- `DUT_in_bidir` specify the input logic value for a bidirectional signal used as an input in the current cycle. The specified value should be preceeded by an "I," indicating that the bidirectional signal is an input. If the bidirectional signal is not an input in a given cycle, then the letter "X" should be used with no value following. Again, values for bidirectional signals should appear in the same order as the signals were listed in the first section.
- `DUTout` specifies the expected logic value from an output signal. Each value should be preceeded by an "O." The input values should appear in the same order as the input signals were specified in the first section of the file.

- `DUT_out_bidir` specifies the input logic value for a bidirectional signal used as an output in the current cycle. The specified value should be preceeded by an "O", indicating that the bidirectional signal is an output. If the bidirectional signal is not an output in a given cycle, then the letter "X" should be used with no value following. Again, values for bidirectional signals should appear in the same order as the signals were listed in the first section.

An example of a test vector file in the ProTest format is provided below.

```
reset 1 1 DUT-inp
stall 1 1 DUT-inp
cache 1 1 DUT-inp
ftest_si 1 1 DUT-inp
dtest_si 1 1 DUT-inp
etest_si 1 1 DUT-inp
rtest_si 1 1 DUT-inp
stest_si 1 1 DUT-inp
test_se 1 1 DUT-inp
addr_TINY 3 0 DUT-out
nCS_TINY 1 1 DUT-out
nWE_TINY 1 1 DUT-out
nOE_TINY 1 1 DUT-out
addr_DMA 3 0 DUT-out
nCS_DMA 1 1 DUT-out
nWE_DMA 1 1 DUT-out
nOE_DMA 1 1 DUT-out
ftest_so 1 1 DUT-out
dtest_so 1 1 DUT-out
etest_so 1 1 DUT-out
rtest_so 1 1 DUT-out
stest_so 1 1 DUT-out
data_TINY 7 0 DUT-bidir
data_DMA 7 0 DUT-bidir
1 1 0 0 0 0 0 0 0 0 I 00000000 I 00000000 O 0000 O 0 O 0 O 0 O 0000 O 1
O 1 O 1 O 0 O 0 O 0 O 1 O 0 X X
2 1 0 0 0 0 0 0 0 0 I 00000000 I 00000000 O 0000 O 0 O 0 O 0 O 0000 O 1
O 1 O 1 O 0 O 0 O 0 O 1 O 0 X X
```

Appendix E

Conversion Script for IMS Format

This following script IMS_tvgen is used to convert the ProTest test vector to IMS test vector.

```
#!/dcs/bin/perl
#!/usr/local/bin/perl
#######################################################################
######
#
# Perl Script: IMS_tvgen for IMS tester in LSIM format
#
# Author:  Guangming Lu : glu@ece.uci.edu
#
# Copyright (c) 2000   All rights reserved.
#
#######################################################################
######
print "\n\n\n";
print "########################################\n";
print "Perl Script:  IMS_tvgen \n";
print "Author:  Guangming Lu\n";
print "Copyright (c) 2000   All rights reserved.\n";
print "If having any bugs, ";
print "please report to Guangming Lu at glu\@ece\.uci\.edu\n\n";
print "########################################\n";
$usage =
   "\n"
 . "Usage:  IMS_tvgen [-t <cycle time in ns>] <InputFile> \n"
 . "    <InputFile> ... --file that need to be converted. \n"
 . "        The default file extension is *.tv instead of *.vec, which
is \n"
 . "        the original file extension. \n"
 . "    -t <cycle time in ns> --specify a cycle time here, the default
is \n"
 . "        10 ns. the unit is ns \n"
 . "\n";

require "getopts.pl";

&Getopts('t:') || die $usage;
die $usage unless (@ARGV >= 1);

$InputFile   = shift( @ARGV );
$OutputFile  = $InputFile;

# change the file name from *.vec to *.tv
$OutputFile  =~ s/.vec\b/.tv/;

if( defined( $opt_t ) )  {
   $period = $opt_t;
} else {
   # No argument was given so use the default
   $period = 10;
   }

open( INFILE, "< $InputFile")
    || die "Could not open the file!\n";
open( OUTFILE, "> $OutputFile" )
         || die "Could not open the file to write. !!! \n";

open(HEADFILE, "<header")
    ||  die "Could not open the header file!\n";

while ( <HEADFILE>) {
    print OUTFILE;
}

$counter = 0;
$next_Input =
"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZ";

while ( <INFILE> ) {
    $Line = $_;
    @field = split(/ /);
    $LineNum = $field[0];
    if($LineNum =~ /^\d+/) {
       $Line =~ s/${LineNum} //;  # Only remove the first column;
       $Line =~ s/ //g;                  # remove the empty space
       $Line =~ s/I//g;
       $Line =~ s/X/ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ/g;

# Change Z to . in the input field
       @field = split(/O/, $Line);
       $old = $field[0];
       $field[0] =~ s/Z/./g;
       $new = $field[0];
       $Line =~ s/${old}/${new}/;

       $TimeNow = $counter * $period;
       @FirstHalf = split(/O/, $Line);    # fetch the input part;
       $FirstLine = "\@${TimeNow} <1".$FirstHalf[0]."
>ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;\n";
# don't compare the output value.
                                          # First half clock cycle = 1;

       $TimeNow = $TimeNow + 0.5 * $period;
       $Line =~ s/O/ >/;                   # change the first O to >;
       $Line =~ s/O//g;                   # remove the rest O;
       $Line =~ s/\n/\;\n/;               # Change \n to ;\n
       if($counter == 1) {
        $Line =~ s/1/0/;                # change the reset to low at
second
                                        # half clock cycle at cycle
number 1
```

```
        }
        $SecondLine = "\@${TimeNow} <0".$Line;
        print OUTFILE $FirstLine;
        print OUTFILE $SecondLine;
        $counter++;
    }
}
print OUTFILE "END\n";
close(INFILE);
close(HEADFILE);
close(OUTFILE);
```

Appendix F

Two New A-type Instructions in M2

F.1 Read Status dr

Where dr is a register in the register file.

31 - 25	24	23 - 20	19 -16	15 - 12	11 - 0
0111100	0	X	X	dr	X

F.2 Read RC specified by Col# from top row in dr

Where dr is a register in the register file.

31 - 25	24	23 - 20	19 -16	15 - 12	11 3	2-0
0111101	0	X	X	dr	X	Col#

Notice that in M1 the above instruction was in subgroup B3.

Appendix G

MorphoSys-M2 Instructions

LDCTXT Initialize the DMA controller to Load Context Memory

LDCTXT SreReg1, r/c#, r/c, context#, #contexts to be loaded						
31-26	25	24	23-20	19-16	15-8	8-0
6	1	1	4	4	8	9
100000	R/c	X	Sr1	Rg_context#	X	# of context planes

- Sr1 : Pointer to SRAM
- r/c : column or row broadcast (0—column, 1 – row)
- Rg_context# : Pointer to context memory.

LDFB Initialize the DMA controller to Load Frame Buffer from SRAM

LDFB Sr1, bank, set#, #words							
31-26	25	23-20	19-16	15-11	10	9	8-0
6	1	4	4	5	1	1	9
100010	X	Sr1	Rg_FB	X	Bank	Set#	# of data planes

- Sr1 : Pointer to SRAM
- Rg_FB : Pointer to FB
- bank : 0: B, 1: A
- set# : set number: 0/1.

STFB Initialize the DMA controller to Store Frame Buffer in SRAM

STFB Sr1, bank, set#, #words							
31-26	25	23-20	19-16	15-11	10	9	8-0
6	1	4	4	5	1	1	9
100011	X	Sr1	Rg_FB	X	Bank	Set#	# of data planes

- Sr1 : Pointer to SRAM
- Rg_FB : Pointer to FB
- bank : 0: B, 1: A
- set# : set number 0/1.

RCRAM2FB Initialize the DMA controller to Load FB from Local RAM

RCRAM2FB							
31-26	25-20	19-16	15-14	13-11	10	9	8-0
6	6	4	2	3	1	1	9
100100	X	Rg_FB	X	Col #	Bank	set	# word

- Col # : Active column
- Rg_FB : Pointer to FB
- Bank : Bank in FB
- # words : Number of words to be transferred.

FB2RCRAM Initialize the DMA controller to Store FB into Local RAM

FB2RCRAM							
31-26	25-20	19-16	15-14	13-11	10	9	8-0
6	6	4	2	3	1	1	9
100101	-----	Rg_FB	--	B_row_col	Bank	set	#words

- Col # : Active column
- Rg_FB : Pointer to FB
- Bank : Bank in FB
- Set# : specify which set
- #words : Number of words to be transferred.

CBCAST Context broadcast with no data

CBCAST b_all, b_row_col, r/c, context#							
31-26	25	24	23-16	15	14-12	11-3	2-0
6	1	1	8	1	3	9	3
111000	-	b_all	--------	r/c	alignment	context#	b_row_col

- b_all : 1: all RC's are active. 0= only one row/column is active
- b_row_col : if b_all =0, then b_row_col specify which row/column is active.
- r/c : 0= column context broadcast; 1= row context broadcast.
- context# : context plane number
- alignment : alignment field

SBCB Context Broadcast with single bank data

SBCB b_all, b_row_col, r/c, context#, bank, set#, bank_addr								
31-26	25	24	23-20	19-17	16	15	14-3	2-0
6	1	1	4	3	1	1	12	3
110100	set	b_all	Reg_Context#	---	bank	R/c	Bank_addr	b_row_col

• b_all	: 1: All RC's active, 0: one col/row active
• B_row_col	: Active col/row
• r/c	: 1: row broadcast, 0: column broadcast
• Bank	: Bank in FB
• Bank_addr	: Direct FB address (9 bits)
• Reg_context#	: Pointer to CM.

DBCBC Column Context Broadcast with double bank data

DBCBC Sr1, bank_B_addr_base, b_all, r/c#, context#, set#, bank_A_addr							
31-26	25	24	23-20	19-16	15-12	11-3	2-0
6	1	1	4	4	4	9	3
111100	set	b_all	Sr1	Sr2_BankA	Base_bankB	Context#	B_row_col

- Set : Specify frame buffer set 0 or set 1
- b_all : the same as SBCB
- Sr1 : Register which has address offset of Bank B
- Base_bankB : Direct base address for bank B. These 4 bits, together with the 5 bits from Sr1, provide the Bank B address (9 bits)
- Context # : See SBCB
- B_row_col : Active column
- Sr2_BankA : Pointer to Bank A.

DBCBR Row Context Broadcast with double bank data

DBCBR Sr1, bank_B_addr_base, b_all, r/c#, context#, set#, bank_A_addr							
31-26	25	24	23-20	19-16	15-12	11-3	2-0
6	1	1	4	4	4	9	3
111101	set	b_all	Sr1	Sr2_BankA	Base_bankB	Context#	B_row_col

- The fields have the meaning as the above instruction.

WFB Write to FB from an RC column

WFB Sr1, r/c#, r/c, bank, set#								
31-26	25	24	23-20	19-18	17	16	15-3	2-0
6	1	-	4	2	1	1	13	3
101000	set	X	Sr1	X	X	bank	X	Col #

- Sr1 : Pointer to FB;
- Col # : Active column.
- Bank : Bank in FB;

WFBI Write to FB from an RC column with immediate address

WFB Sr1, r/c#, r/c, bank, set#								
31-26	25	24	23-18	19-18	16	15	14-3	2-0
6	1	-	4	2	1	1	12	3
101001	set	X	X	X	bank	X	Bank Adrs	Col #

- Bank Adrs : Direct Adrs to FB;
- Col # : Active column.
- Bank : Bank in FB;

Appendix H

RC Instruction Set Architecture - M2

RC instructions for Graphics Applications (M2-G) are described in this section. Compared with the M1 version of the instruction set, the M2-G instruction set includes pseudobranch, shift instructions of different types, and index mode memory access instructions. Most instructions are executed only when the specified conditionals are true, and most instructions can have the computed results shifted. There are 16 instruction formats for RC instructions. Each format applies to a group of instructions. These formats are given in Table H-1.

Table H-1. MorphoSys RC Instruction Set Version for Graphics Applications

31-26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**INST_GRP**
OP	**Wex**	**DST**				**RL_SFT**	**X**	**#SFT**				**MUXA(SRC1)**					**MUXB(SRC2)**					**S/N**	**COND**				**ALU**
Operation: First [SRC1] op [SRC2], then shift the result in #SFT bits, and write result into DST. *See notes for other explanations.*																											
OP	**WEX**	**DST**				**RL_SFT**	**X**	**#SFT**				**MUXA(SRC1)**					**MUXB(SRC2)**					**SRC_MAC**				**Rg32**	**MAC**
Operation: First [SRC1] * [MAC] + [SRC2], then shift the result in #SFT bits, and write result into DST. *See notes for other explanations.*																											
OP	**Wex**	**DST**				**RL_SFT**	**X**	**#SFT**				**MUXA(SRC1)**					**MUXB(SRC2)**					**XXXXX**				**Rg32**	**MUL**
Operation: First [SRC1] * [SRC2], then shift the result in #SFT bits, and write result into DST. *See notes for other explanations.*																											
OP	**WEX**	**DST**				**XXXXX**					**IMMEDIATE**																**LDIMM**
Operation: load immediate value into DST. *See notes for other explanations.*																											
OP	**Wex**	**DST**				**XXXXXX**						**XXXXXX**				**xxxxxx**						**MEM_ADDR**				**X**	**LDMEM**
Operation: load [MEM_ADDR] into DST. *See notes for other explanations.*																											
OP	**Wex**	**SRC1**				**RL_SFT**	**X**	**#SFT**				**X**	**SUB_OPCODE**				**SRC2**					**DST**				**X**	**STMEM**

Table H-1. MorphoSys RC Instruction Set Version for Graphics Applications "(Continued)"

Operation: CISC type store. SUB_OPCODE specifies what operations to do on SRC1 and SRC2, and the result is written into MEM [DST]. SUB_OPCODE is defined as follows: 1:OR; 2:AND; 3: XOR; 4:ADD; 5:SUBAB; 6:SUBBA; 9:ABSUB; 0:STORE. For {0} STORE, the value in SRC1 is written into MEM [DST]. Other fields are identical to the corresponding fields in the ALU instructions. *See notes for other explanations.*										**(CISC)**
Op	**Wex**	**SRC**	**XXXXXXXXXXXXXX**					**XXXX**	**X**	**LDMAR**
Operation: Load [SRC] into DMA Register. *See notes for other explanations.*										
OP	**Wex**	**DST**	**RL_SFT**	**X**	**SFT_TYPE**	**MUXA(SRC)**	**MUXB(Reg_#SFT)**	**S/N**	**COND**	**SHIFT**
Operation: 16-bit operation. Shift [SRC] in [Reg_#SFT] amount and write the result into DST. *See notes for other explanations.*										
OP	**Wex**	**DST (even)**	**RL_SFT**	**X**	**SFT_TYPE**	**MUXA(SRC_even)**	**MUXB(Reg_#SFT)**	**S/N**	**COND**	**SFT32**
Operation: shift ([SRC_EVEN+1]<<16|[SRC_EVEN]<<16) in [Reg_#SFT] amount and write result into register pair (DST+1):(DST). *See notes for other explanations.*										
OP	**Wex**	**DST**	**X**	**X**	**XXXX**	**MUXA(SRC_MSB)**	**MUXB(SRC_LSB)**	**S/N**	**COND**	**CLZ32**
Operation: 32-bit operation. Return the leading zeros of [SRC_MSB]<<16|[SRC_LSB] to DST. *See notes for other explanations.*										
OP	**Wex**	**XXXX**	**X**	**X**	**XXXX**	**MUXA(SRC)**	**XXXX**	**S/N**	**COND**	**LDINX**
Operation: Load index register with [SRC] ***See notes for other explanations.***										
OP	**Wex**	**DST**	**AutoInc**	**X**	**XXXX**	**MUXA(Base_Reg)**	**XXXX**	**S/N**	**COND**	**LDBIX**
Operation: load data from [Base_reg+Index_reg] into DST, auto-increment index register if AutoInc is set. *See notes for other explanations.*										

OP		**DST0**	**DST2**	**SRC2**	**DST1**	**SRC1**	**SRC0**	**X**	**MLOAD**
		DST0					**SRC0**		
		DST0	**DST2**	**SRC2**			**SRC0**		
		DST0	**DST2**	**SRC2**	**DST1**	**SRC1**	**SRC0**		

Table H-1. MorphoSys RC Instruction Set Version for Graphics Applications "(Continued)"

Operation: Encapsulate multiple load instructions into one instruction. Up to one, two, or three memory loads can be included in one instruction. If there is more than one memory load, in the first cycle [SRC0] is loaded into DST0, then in the next cycle, [SRC1] is loaded into DST1 automatically without new command.										
OP	**XXXXXXXXXXX**					**Label**		**S/N**	**COND**	**BRANC**
Operation: For PBRANC, if conditional is true, load "label" into index register and make RC into sleep mode. For PTARG, if "label" matches the Label register value, wake up RC. The PTARG uses the same opcode as KEEPP. Reserve label=0 for KEEPP instruction. *See notes for other explanations.*										**PTARG** **KEEPP**
OP	**XXXXXXXXXXXXXXXXXXXXXX**							**S/N**	**COND**	**RESET**
Operation: reset RC registers *See notes for other explanations.*										
OP	**Wex**	**DST**	**RL_SFT**	**X**	**#SFT**	**MUXA(SRC)**	**XXXXX**	**S/N**	**COND**	**BYPAS**
Operation: For BYPAS, move [SRC] to DST, and shift it if #SFT is not zero. For CNTLZ, return the leading zeros of [SRC] to DST. *See notes for other explanations.*										**CNTLZ**

Notes:

1. The fields marked as "XXX" are unused fields
2. The meaning of each label is as follows.
   a. OP: opcode
   b. Wex: Write Express lanes
   c. DST: default destination field
   d. RL_SFT: the bit indicating leaf or right shift
   e. #SFT: Shift amount
   f. MUXA, MUXB: input s from multiplexer A or B
   g. SRC: source register
   h. S/N: the bit indicating flag is set or not set in this instruction
   i. COND: conditional flags
   j. MAC: register used in MAC operation

k. Reg_#SFT: the register that have the shift amount
l. Base_Reg: Base address register
m. Rg32: the bit to indicate to keep the 32-bit result. When rg32=1, only even indexed registers are used: r0=(r0, r1), r2=(r2, r3). R4=(r4, r5), r6=(r6, r7), …, r12=(r12, r13), r14 is forbidden. R15 is always written.
n. For MAC and Memory instructions and ADDSUB, there is no conditional execution field, and all are executed unconditionally.

3. If you do not want to write to the destination register in the ALU instruction, you can specify it in a format like " OP R15 A B," since R15 is always used as a garbage register. This can be used for comparing two values.
4. For MAC instructions, in which you may want to write the 32-bit result, you can specify 32 as the suffix.
5. For shift32 instruction, the amount of shift is specified in SRC2 by using one register, SRC1 is the source of the shifting, it has to be an even register.

The conditional flags are defined in Table 2.

Table H-2. Conditional Flags Specification

Code	Suffix[cond]	Flags	Description
0000	EQ	Z set	Equal
0001	NE	Z clear	Not equal
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	overflow
0111	VC	C clear	No overflow
1000	HI	C set or Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N equals V	Greater or equal
1011	LT	N not equal to V	Less than
1100	GT	Z clear and (N equals V)	Greater than
1101	LE	Z set or (N not equals V)	Less than or equal
1111	AL	DoeS/N't test any flag	PSEUDOBRANCH ALWAYS (DEFAULT)

The followings are the instruction syntax definition for all of the instructions.

Table H-3. Instruction Syntax Summary

Mnemonic	OPCODE	Inst. Format Group	Syntax	Action
BYPAS	000000	BYPAS	BYPAS [cond][S/N] Dst Src	Dst=Src
ORRAB	000001	ALU	ORRAB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src1 OR Src2
ANDAB	000010	ALU	ANDAB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src1AND Src2
XORAB	000011	ALU	XORAB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src1 XOR Src2
ADDAB	000100	ALU	ADDAB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src1+Src2
SUBAB	000101	ALU	SUBAB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src1-Src2
SUBBA	000110	ALU	SUBBA[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=Src2-Src1
ANDCT	000111	ALU	ANDCT[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=left shift (Src1and Src2) until '1'
ABSUB	001001	ALU	ABSUB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Dst=|Src1-Src2|
CXADD	001101	ALU	CXADD[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Reg=complx(A+B)
CXSUB	001110	ALU	CXSUB[cond][S/N] Dst Src1 Src2 [RLS] [#SFT][WEX]	Reg=complx(A-B)
KEEPP/PTARG	001010	KEEPP	KEEPP[cond][S/N]; PTARG[cond][S/N] label	
UCMUL	010000	MUL	UCMUL DST src1 src2 [RLS] {#SFT] [WEX]	Dst=unsigned complx(Src1*src2)
CMUL	010001	MUL	CMUL DST src1 src2 [RLS] {#SFT] [WEX]	Dst=signed complx(Src1*src2)
MULADD	010100	MAC	MULADD Dst Src1Mac Src2 [RLS][#SFT][WEX]	Dst=Src1*Mac+Src2
MULSUB	011000	MAC	MULSUB Dst Src1Mac Src2 [RLS][#SFT][WEX]	Dst=Src1*Mac-Src2
UMUL	010010	MUL	UMUL Dst src1 src2 [RLS] {#SFT] [WEX]	Dst=unsigned(Src1*Src2)
MUL	010011	MUL	MUL Dst src1 src2 [RLS] {#SFT] [WEX]	Dst=signed(Src1*Src2)
LDIMM	100000	LDIMM	LDIMM Dst Immediate_value [WEX]	Dst=immediate
LDMEM	100001	LDMEM	LDMEM Dst Mem_addr_reg [WEX]	Dst=Mem[Mem_addr_reg]
STMEM	100010	STMEM	STMEM SRC Mem_addr_reg [WEX]	Mem[Mem_addr_reg] =src
LDMAR	100011	LDMAR	LDMAR Addr_reg	Dma_reg= addr_reg
LINDX	100100	LINDX	LINDX Src	Index_reg = Src
LDBIX	100101	LDBIX	LDBIX Dst (Rb)+	Dst=Mem[Rb+index_reg]+
INCMT	110000	BYPAS	INCMT[cond][S/N] Dst Src [RLS][#SFT][Wex]	Dst=Src+1

Table H-3. Instruction Syntax Summary “(Continued)”

Mnemonic	OPCODE	Inst. Format Group	Syntax	Action
DECMT	110001	BYPAS	DECMT[cond][S/N] Dst Src[RLS][#SFT] [Wex]	Dst=Src-1
CNTLZ	110010	CNTLZ	CNTLZ[cond][S/N] Dst Src [RLS][#SFT] [WEX]	Dst = #leading 0 of Src
PBRAN	110100	PBRAN	PBRAN[cond][S/N] Label	If cond=true, label reg=label, rc into sleep mode
SHIFT	110101	SHIFT	SHIFT[type][cond][S/N] Dst Src #sft [WEX]	Dst = shift (Src) as type by #sft
ABBSS	110110	BYPAS	ABBSS[cond][S/N] Dst Src [WEX]	Dst=|src|
ADSUB	110111	ALU	ADSUB[cond][S/N] Dst Src1 src2 [WEX]	If (cond=1)Dst=Src1+Src2, else Dst = Src1-Src2
ADDCC	111000	ALU	ADDCC[cond][S/N] Dst Src1 src2 [WEX]	Dst=Src1+Src2+C
SUBCC	111001	ALU	ADDCC[cond][S/N] Dst Src1 src2 [WEX]	Dst=Src1-Src2-C
CLZ32	111010	CLZ32	CLZ32 Dst Src1 src2	Dst=leading ‘0’s of (Src2<<16|Src1)
SFT32	111011	SFT32	SFT32 [type][cond][S/N] Dst Src #sft [WEX]	Dst=(shift (Src+1)<<16|(src)) as type by #sft