Fall 2003

Configurable computer systems can support dataflow computing

Anish Arvind Sathe
New Jersey Institute of Technology

ABSTRACT

CONFIGURABLE COMPUTER SYSTEMS CAN SUPPORT DATAFLOW COMPUTING

by
Anish Arvind Sathe

This work presents a practical implementation of a uni-processor system design. This design, named D^2-CPU, satisfies the pure *data-driven* paradigm, which is a radical alternative to the conventional von Neumann paradigm and exploits the instruction-level parallelism to its full extent. The D^2-CPU uses the natural flow of the program, *dataflow*, by minimizing redundant instructions like fetch, store, and write back. This leads to a design with the better performance, lower power consumption and efficient use of the on-chip resources. This extraordinary performance is the result of a simple, pipelined and superscalar architecture with a very wide data bus and a completely out of order execution of instructions. This creates a program counter less, distributed controlled system design with the realization of intelligent memories. Upon the availability of data, the instructions advance further in the memory hierarchy and ultimately to the execution units by themselves, instead of having the CPU fetch the required instructions from the memory as in controlled flow processors. This application (data) oriented execution process is in contrast to application ignorant CPUs in conventional machines. The D^2-CPU solves current architectural challenges and puts into practice a pure data-driven microprocessor. This work employs an FPGA implementation of the D^2-CPU to prove the practicability of the data-driven computer paradigm using configurable logic. A relative analysis at the end confirms its superiority in performance, resource utilization and ease of programming over conventional CPUs.

CONFIGURABLE COMPUTER SYSTEMS CAN SUPPORT DATAFLOW COMPUTING

by
Anish Arvind Sathe

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

January 2004

APPROVAL PAGE

CONFIGURABLE COMPUTER SYSTEMS CAN SUPPORT DATAFLOW COMPUTING

Anish Arvind Sathe

Dr. Sotirios Ziavras, Thesis Advisor — Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT.

Dr. Alex Gerbessiotis, Committee Member — Date
Assistant Professor of Department of Computer Science, NJIT.

BIOGRAPHICAL SKETCH

Author: Anish Arvind Sathe

Degree: Master of Science

Date: January 2004

Undergraduate and Graduate Education:

Master of Science in Electrical Engineering,
New Jersey Institute of Technology, Newark, NJ, 2004

Bachelor of Engineering in Electrical Engineering,
Government College of Engineering, Pune, India, 2001

Major: Electrical Engineering

Dedicated to my parents

ACKNOWLEDGEMENT

First of all, I would like to thank Dr. Ziavras for his guidance and support during the whole tenure of my study for this thesis. I think only with his timely advice and great encouragement, I am able to complete this thesis work.

At the same time, I am also thankful to Dr. Hou and Dr. Gerbessiotis for participating in the thesis committee and providing valuable suggestions for improvement. Also many thanks to the entire staff of the Electrical and Computer Engineering department at NJIT.

Also I would like to thank senior colleagues, Tirupati, Satchit, Zafrul, Sunil and Xizhen for their immense help during practical difficulties. And finally, although not in the least, I want to thank all my friends here in NJIT who are behind me in hard time.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Conventional von Neumann Architecture

Conventional computers are based on a control flow mechanism. The order of program execution is decided by the user and is stated in the program. The program counter (PC) is used at the hardware level to obey this order. This PC sequences the execution of instructions in a program. Ultimately, the PC leads to the sequential execution of a program on a *control-driven* architecture. These types of computers use shared memory to hold instructions and data objects. Instructions and data objects are stored differently. Many instructions can change shared memory. So, dependencies and control flow have to be followed carefully for the correct execution of programs. The central processing unit (CPU) fetches each instruction and its data and executes it. This makes the CPU the master of computer system, where as the shared memory remains "dumb". This creates a large amount of redundant operations and, in turn, low utilization of the resources directly related to the implementation of application algorithms.

Each fetch instruction directly or indirectly relates to memory use. For programming flexibility, these computer systems became very complex at the hardware level with large complex instruction sets. This has created many processor families, though the emphasis remains on the reducing the complexity, leading to RISC, or increasing operations per cycle leading to pipelining and super-pipelining, or increasing the parallel operations per cycle leading to superscalars and VLIW machines.

To increase the clock frequency and decrease the cycles per instruction (CPI) functional units are divided into smaller sets of logic. Different stages are created for each functional unit leading to *pipelining*. With more advanced technology the stages are further

divided to develop *super-pipelined* CPUs which ideally increase throughput in linear with the number of stages.

Through many years of research, it was realized that only 25% of the instructions of complex instruction sets are used often. This fact implies 75% of the hardware-supporting other instructions are not used frequently. So, putting such instructions into software saved valuable chip space leading to RISC processors; the latter space can now be used for large register sets, local caches, and , sometimes, for floating point units. Instruction and data caches can then be made separate. This modification gives higher clock rate, fewer cycles per instruction (CPI). They are often called *scalar-RISC* processors.

With the advent of VLSI technology more on-chip area is available and that can be used to increase in resources and, sometimes, for resource duplications. This leads to *super-scalar* processors with more than one functional unit. Pipelining and super-scalar principles are used to create super-scalar, super-pipelined processors, which result in less than one CPI. Very large instructions (VLIW) were introduced to reduce memory latencies.

Unfortunately, the trend in CPU design has been to take advantage of increases in transistor densities to include additional features. Today's processors are nothing else but combinations of past research and that can implement wide instruction issue (VLIW), out-of-order instruction execution (data-flow after instruction issue), aggressive speculation, and in-order retirement of instructions.

1.2 Dataflow Architectures

In a dataflow computer, the execution of an instruction is driven by the data availability instead of being guided by the program counter. Ideally, in the pure dataflow computation model an instruction is executed as soon as its operands become available. The instructions in dataflow programs are not ordered in any way and there is no need of a program counter.

Instructions under this model carry their own data, i.e. operands, with them. As soon as an instruction produces a result, the result is broadcasted to all needy instructions. Again, as soon as any instruction gets all its operands, it is ready to execute and it moves to the CPU in the dataflow paradigm. Thus, the CPU doesn't fetch any instruction from memory. In short, the CPU is deprived of its Master right in the dataflow model, becoming a PU and memories become intelligent. This gives distributed control in the computer system.

The dataflow model has the potential to exploit all the parallelism available in a program. Since the execution is driven only by the availability of operands at the inputs to the functional units, its parallelism is limited only by the actual data dependencies in the application program rather control dependencies that become problematic in the conventional von Neumann model. The dataflow execution follows precisely dataflow graphs, which have embedded inherent parallelism. Thus, dataflow architectures represent a radical alternative to von Neumann architectures. There is a lot of work already done for the dataflow paradigm and there are also other architectures available which use both dataflow and von Neumann architectures to exploit, the inherent parallelism in dataflow and the ease of control flow in von Neumann respectively.

1.3 Related Work with the Dataflow Paradigm

The dataflow computation paradigm till now has been primarily employed in the implementation of parallel computers, where this paradigm is basically applied among instructions running on different PC-driven processors. The majority of dataflow multiprocessors and multi-computers used COTS (Commercial Off The Shelf) processors, which gave them the advantage of fast designing however they still remain PC driven at individual processor level. In contrast, a data-driven processor was introduced in [9] that utilizes a self-timed pipeline scheme to achieve distributed control. This design is based on

the observation that the *data-driven* paradigm can accommodate very long pipelines that are controlled independently, since packets flowing through them always contain enough information and data on the operations to be applied. However, this processor design also suffers from several constraints imposed by current design practices. Several data-driven architectures have been introduced for the design of high performance ASIC devices [10, 11]. In addition, several techniques have been developed for the implementation of ASICs in VLSI when the dataflow graphs of application algorithms are given. However, these techniques employ straightforward, one-to-one mapping of nodes from the dataflow graph onto distinct functional units in the chip. An exception is the recently proposed implementation of dataflow computation on FPGAs [12].

Multithreading is another widely used principle in CPU design. For multithreaded processors, each program is partitioned into a collection of instructions. Such a collection is called a thread. Instructions in a thread are issued according to the conventional von-Neumann model of computation, i.e. they are sequential. Similar to the dataflow model, instructions are run based on data availability [5]. A large degree of thread-level parallelism is derived through a combination of programmer, compiler, and hardware efforts. Similar to the above case COTS processors can be used for this purpose. Data dependencies are taken care of by the compilers and split-phase techniques guarantee no trouble without extra memory-access delay. Multithreading supports the dataflow execution among threads. The *Tera Multi-threaded architecture* (MTA) is an example of a distributed shared memory parallel machine with multithreaded computational processors and interleaved memory modules connected via a packet-switched interconnection network [7]. Similarly *Efficient Architecture for Running Threads* (EARTH) is a multiprocessor that contains multithreaded nodes [6]. Again each node contains a COTS RISC processor for executing threads sequentially and an ASIC synchronization unit that supports dataflow like thread

synchronizations and scheduling. A thread is activated when all its input data become available and then they can spawn or create many other threads. This principle directly relates to the dataflow graphs.

As far as single processors are concerned, the *hyper-threading* technology, introduced in the Intel *Pentium-IV*, provides thread-level-parallelism (TLP) on a single processor, resulting in increased utilization of processor execution resources [19]. As a result, resource utilization yields higher processing throughput. The hyper-threading technology is a form of *simultaneous multi-threading technology* (SMT) where multiple threads of software applications can run simultaneously on one processor. However, this is achieved by the duplication of resources on each processor. To match the instruction level parallelism (ILP) of applications, it is a usual practice to design microprocessors with resource duplication. Several copies of commonly used functional units are implemented in the CPU, which is called super-scaling. Multiple-issue processors [8], an alternative to vector processing units, apply this super-scaling principal for dynamic execution whereas VLIW can be used for static scheduling. In VLIW, the compiler combines many independent instructions together to be sent simultaneously to the CPU. Each component instruction is to use its own execution unit in the CPU. Resource widening [4] is another concept implemented in the Intel IA-64. This *Explicit Parallel Instruction Computing* (EPIC) design approach used in the Intel IA-64 is similar to the VLIW paradigm but increases the hardware complexity.

1.4 Motivation

The high complexity of individual processors has a dramatic negative effect on the overall complexity and performance of parallel computers. Current design families, like RISC, CISC, and VLIW processors show several deficiencies. They are characterized by large amounts of redundant operations and low utilization of resources directly related to the

implementation of application algorithms. In all these architectures, an instruction fetch operation is still required only due to the von Neumann PC-driven basic model. The CPU request to the memory is not part of any application algorithm but the result of centralized control during program execution. To reduce this time penalty, all of today's implementations use instruction *pre-fetching* with an instruction cache. This wasted recourses which could be otherwise used in more direct application related tasks. Another problem with current designs is the fact that the operands do not often follow their instructions to the CPU. The only exception is the instructions that either use immediate data or their operands reside in the CPU registers. Additional fetch cycles may then be needed to fetch these operands from either the main memory or the attached cache. However, these fetch cycles also should be avoided, if possible. These fetch cycles are even unavoidable with current dataflow designs that use activation frames. Again, to mitigate this problem current designs choose data cache memories; corresponding transistors could be otherwise be used in more productive tasks. In contrast, in the pure dataflow paradigm computing, the instructions go to the execution unit on their own (Intelligent Memory) if needed, along with their operands, as soon as they are ready to execute.

Thus, advances in current CPU design lack the potential for dramatic performance improvements because they don't match well with the natural execution of program flows. To get rid of such critical problems or, sometimes, to lessen this effect, designers used many expensive hardware techniques. However, this hardware is not used to run the relevant program directly but just aid in increasing the efficiency or through put. This results in small productive utilization of the overall hardware system. The time penalty of fetching instructions and operands in conventional von Neumann architectures is reduced by extensively using instruction and data pre-fetching, software preprocessing, internal data

forwarding and cache techniques. Resulting new architectures result in the following penalties:

1. In an effort to hide the mismatch between the application's needs and the PC-driven execution model, we waste numerous on chip resources. Many hundreds of thousands or millions of transistors are needed to implement some of the above techniques within a single CPU, whereas the productive utilization of these resources is rather small.

2. Power consumption increases for two reasons. Firstly, the overheads of the instruction fetch cycle, which is not an application requirement, appears for each individual instruction in the program. This is too much an overhead to pay for centralized control during program execution. Since these are inter-chip data transfers that are quite expensive and time consuming, this cost is very substantial. Secondly, unnecessary power consumption results from pre-fetching unneeded instructions and data into caches. Mobile computing, recently popular and dominating the computer field needs very high power efficiency for longer battery usage.

3. Numerous cycles are wasted when a hardware exception or interrupt occurs. This is because after the CPU gets informed about the external event, it has to store the current state of the machine and then fetch code to run the corresponding interrupt service routine. If the appropriate context switching is selected outside of the CPU, then the appropriate instructions can arrive promptly.

1.5 Objective

1.5.1 Design Objective

It is now widely accepted that the procedure applied within many advanced microprocessors for the execution of CPU resident instructions resembles closely the data-driven computation paradigm. This is due to the fact that these advanced microprocessors apply Tomasulo's algorithm with super-pipelining techniques with using resource reservation stations that keep track of data dependencies between instructions in the CPU.

The data-driven CPU (D^2-CPU), proposed in [1], is a design technique based on the pure dataflow computation paradigm. For ease and efficiency of instruction decoding and implementation, it also uses principles like large register files and active instructions with their operands within the processing unit, simple instructions, and multiple issues of

instructions. In this design, the dataflow model of execution is applied simultaneously to all instructions in the program. This proposed D^2-CPU design has the following architectural objectives:

1. This innovative design has a radical single processor design that implements the data-driven computation paradigm in its pure form. It also employs active memory techniques.

2. A processor design with distributed control that minimizes the amount of redundant operations and maximizes performance.

3. High utilization of resources in productive work, i.e. work not associated with redundant operations but supports direct application flow.

4. Low hardware complexity for high performance.

5. Low cost and power consumption.

Our main objective here is to implement the D^2-CPU design on the FPGAs.

1.5.2 Reconfigurable Computing Systems

Field-Programmable Gate-Arrays (FPGAs) have been used in systems spanning a broad range of applications ever since their introduction in 1985 [14]. Most of the systems use FPGAs as a glue logic providing the advantages of high integration levels without the expense and risk of custom ASIC devices. However, as FPGAs have increased in capacity, their use as in-system configurable computing elements has received considerable attention. The use of FPGAs as reconfigurable computing elements is poised to expand rapidly in the commercial market, where FPGA–based parallel processors will compete with parallel computers and even some supercomputers in computationally intensive applications. Many research projects were done over the past few years in developing these FPGA-based high-performance machines. Reconfigurable FPGA technology holds the potential of reshaping the future of computing by providing the capability to dynamically alter hardware resources to optimally serve immediate computational needs [13].

The FPGA-based reconfigurable systems can be used as specialized co-processors, processor-attached functional units, attached message routers in parallel machines, and specialized systems for parallel processing. This was made possible with the advent of multi-million gate FPGAs. In the past decade, FPGA-based configurable computing machines have acquired significant attention for improving the performance of algorithms in several fields, such as DSP, data communications, genetics, image processing, pattern recognition, etc. FPGA-based co-processors are implemented as attached co-processors dedicated to off-loading computationally intensive tasks from host processors in PCs and workstations. Reconfigurable co-processors are viable platforms for a wide-range of computationally-intensive applications. The FPGA-based configurable computing systems have garnered support from the scientific and academic communities. Many research projects have demonstrated the viability of configurable computing systems that can deliver the performance of supercomputers for specific applications. Most of the FPGA–based parallel machines currently reside in multi-FPGA systems interconnected via a specific network [15]. Some of the configurable computing systems are:

1. The Ganglion Project at the IBM Almaden Research Centre used XC3090 and XC3042 FPGA devices to implement a feed-forward, fully interconnected neural network on a single VME board.

2. DEC's Paris Research Lab has designed and implemented four generations of FPGA-based configurable co-processors called Programmable Active Memories (PAMs).

3. SPLASH-1 includes a 32-stage linear-logic array with a VME-interface to a SUN workstation. Each stage consists of an XC3090 FPGA and a 128Kbyte static memory buffer. SPLASH-1 outperformed Cray-2 by a factor of 325 in specific applications and a custom built NMOS device by a factor of 45. SPLASH-2 uses 17 XC4010 FPGA devices arranged in a linear array and also interconnected via a 16x16 crossbar.

4. PRISM-1 from Brown University coupled XC3090 with the Motorola M68010 microprocessor and PRISM-11 coupled XC4010 FPGA devices as co-processors to an AMD29050 RISC processor.

Advances in VLSI technology not only brought about multi-million gate FPGAs, but also facilitated the integration of numerous functions onto a single FPGA chip. Peripherals formerly attached to the FPGA at the board level now can be embedded into the same chip with the configurable logic. According to Xilinx predictions, the count of FPGA system gates will exceed 50 million and FPGA chips will operate at more than 500 MHz [16].Thus, the availability of multi-million system gates in FPGAs introduced a new design paradigm, System-On-a-Chip (SOC), with which entire systems can be implemented on a single FPGA chip without the need for expensive non-recurring engineering charges or costly software tools.

The FPGAs have provided an alternative method to computing by supporting the fine-tuning of hardware to match software requirements. The fact that the number of system gates in FPGAs has been increasing rapidly in recent years encourages the development of large–scale application-specific custom computing machines on FPGAs for better hardware performance. While these FPGA-based Custom Computing Machines (CCMs) may not challenge the performance of microprocessors for all applications, for specific applications an FPGA-based system can offer extremely high performance. This led us to develop an FPGA-based D^2-CPU proposed in [1].

The main objective of this thesis is to design a general purpose D^2-CPU architecture and implement it on an FPGA. The data-driven computation model is applied simultaneously to all the instructions in the program. Not only the CPU but the L_1 cache, L_2 cache, and main memory are also implemented with this principle. This thesis mainly aims at implementing this architecture to prove the viability of the data-driven computational paradigm with current FPGA technologies.

The proposed design concept is discussed in Chapter 2. The detailed implementation is reported in Chapter 3. The theory of dataflow graphs and programming with the D^2-CPU

are introduced in Chapter 4. Chapter 5 summarizes design results and comparative analysis with conventional designs and Chapter 6 concludes and proposes future design objectives and challenges related to the data-driven paradigm and especially to the D^2-CPU. The target system is the Annapolis Micro systems (AMS) Wildstar-II development board that has two Xilinx Virtex-II FPGAs.

CHAPTER 2

THE D^2-CPU

2.1 Introduction

2.1.1 Some Commonly Used Terms in the Data-driven Paradigm

This Chapter begins by introducing briefly the semantics of the data-driven computer paradigm. The terms here basically describe the sequence of steps for the implementation of an instruction under the data-driven computation paradigm.

1. *Instruction Issuance or Firing*: It is the departure of the instruction for the execution unit. An instruction is fired just after all of its operands become available to it.

2. *Token Propagation*: It is the propagation of an instruction's result to other instructions that need it. As soon as an instruction completes execution, it makes copies of its result for all other instructions that need it. Different tokens that contain the same result are then forwarded to different needy instructions.

3. *Instruction Dissolvement*: It is the destruction of the instruction just after it produces its entire token for other receiving instructions. It depends upon the instruction. Loop instructions have to be treated differently because they may be reused in the programs.

2.1.2 Design Requirements

Following are the major requirements clearly mentioned in [1] for the D^2-CPU design that satisfied our objective and are in line with the data-driven computation paradigm.

1. Programs are developed using fine-grain graphical, or equivalent, languages that show explicitly all data dependencies among the instructions. Libraries of existing routines can further aid programming, as long as they are developed in this manner. Also, usage of a graphical language simplifies code development and facilitates better assignment of tasks to parallel computers containing many D^2-CPU.

2. Instructions contain all their operand fields, as in the pure data-driven model.

3. A software preprocessor finds all the instructions in the program that can run in the very beginning because of non-existent data dependencies. These head instructions are to be sent first to the execution unit.

4. Following the head instructions to the execution unit are instructions that are to receive all their input operands from one or more head instructions. These instructions can proceed for execution just after they receive their operands.

5. Instructions that are to receive one or more operands from instructions that are ready to execute but are still missing one or more operands leave for an external cache, called EXT-CACHE, where they wait to receive their tokens. To reduce the traffic, instructions that will receive the same result are grouped together in the cache in an effort to collectively receive a single token that can be used to write all relevant operand fields. If not all of the token receiving instructions can fit in the EXT-CACHE, then a linked list is created in the memory for instructions that do not fit.

6. Only one copy of each instruction, including its operands, resides at any given time within the entire machine, i.e. in the memory, cache, and CPU. This is in contrast to the wide redundancy of instructions and data present in the cache, memory and CPU of the conventional control driven model.

7. Instructions do not keep pointers to their parent instructions. Therefore, they are dormant till they are forced into the EXT-CACHE or the execution unit in order to receive their tokens.

8. After an instruction is executed, it is dissolved. However, special care is needed for instructions that have to be reused in software loops. A relevant technique that permits instruction reuse is presented in the next Chapter.

9. Instructions have unique IDs for token passing only while they reside outside of the execution unit. These IDs are used to find instructions and force them into the EXT-CACHE or execution unit. In the latter case, an interface actually keeps track of these IDs so that minimal information is manipulated or stored in precious execution unit resources.

2.2 D^2-CPU Design

This Section will present the innovative D^2-CPU design proposed in [1]. Primarily this design takes advantage of advances in Processor In Memory (PIM), cache memory, and IC technologies to implement efficiently the data-driven paradigm. Figure 2.1 shows the system architecture. We will start with the core of this design, i.e. the Execution Ready Unit (ERU) and will advance to the main-memory. The instruction format at each level is different.

2.2.1 General Instruction Format

Each instruction comprises an opcode field (OPCODE) and, without loss of generality, up to two operand fields (OPD_1 and OPD_2). The number of operand fields depends upon the type of operation, i.e. unary or binary respectively. Depending upon its location in the system, each instruction also consists of its own instruction ID defined by its location in the main memory (IAD), instruction IDs upon which OPD_1 and/or OPD_2 are depended- IID_1 and IID_2, respectively. It also comprises the FLAG field that points to instruction location for dependence. This FLAG field makes each instruction intelligent and decides its further action. The instruction format at each location will be discussed in detail below. For token propagation, each token consists of an IAD i.e. the instruction ID that generates this token and its RESULT.

2.2.2 Execution Ready Unit (ERU)

The ERU is the core of this system and replaces conventional CPUs. It consists of functional units and big register files in terms of on-chip caches. The ERU comprises the following components:

- *Processing Unit* (PU): In the PU, the operations specified by the instructions are executed. It contains several functional units that can be used by a single, or simultaneously, by multiple instructions. Its design follows the basic RISC model. The PU contains at least one copy of an adder, a multiplier and a logic unit. For multimedia and engineering applications a vector unit also can be added to the PU. Each instruction at the input to the PU level only comprises of the OPCODE, OPD_1, OPD_2 and IAD fields, whereas at the output it forms tokens with the IAD and RESULT fields.

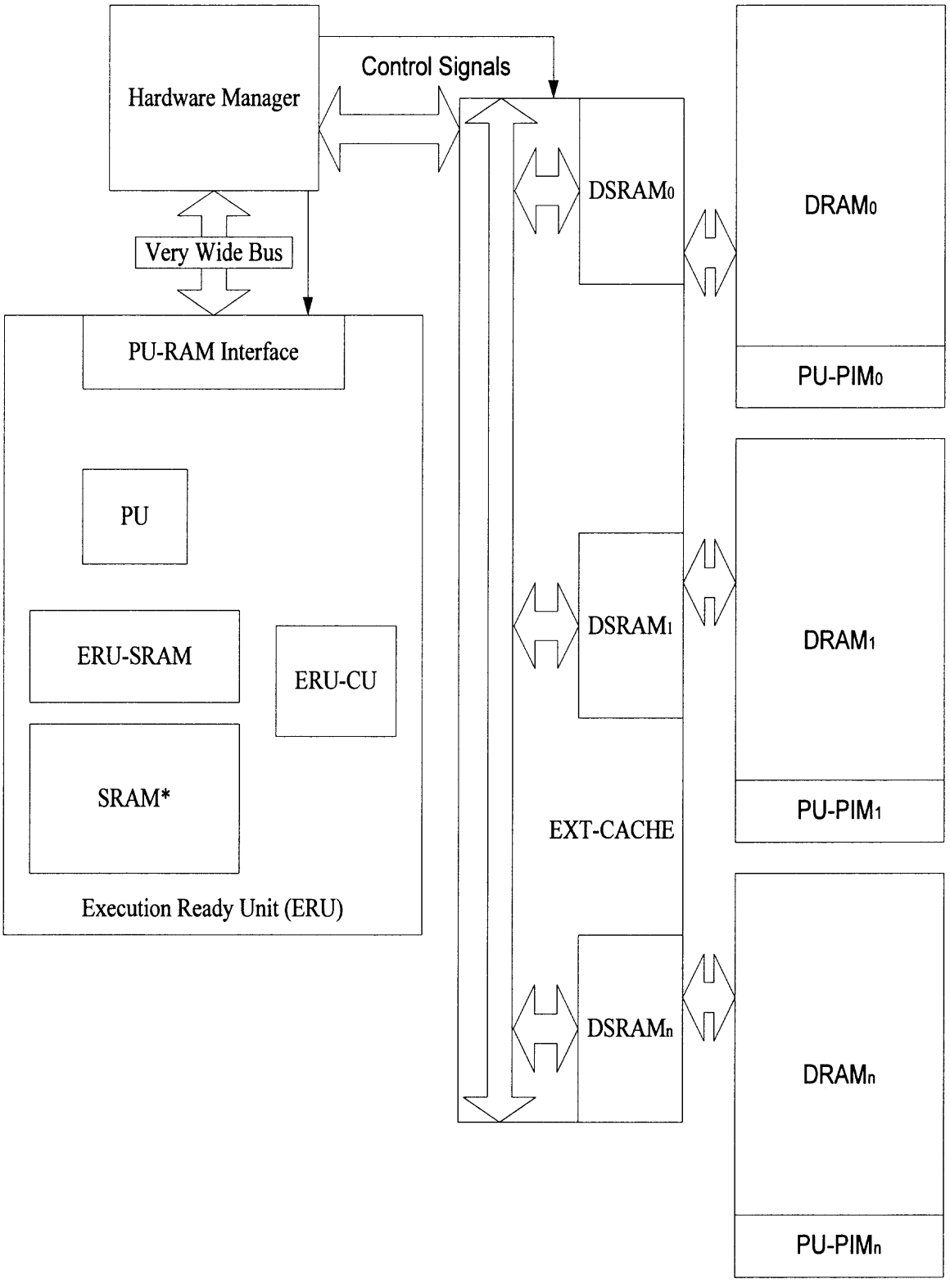


Figure 2.1 D^2- CPU Architecture [1].

- *Static RAM* in the ERU (ERU-SRAM): It contains instructions ready to execute, i.e. instructions with all of their required operand fields filled. However these instructions cannot proceed to the PU, because the functional units that they require are currently used by some other instructions. This cache storage of ready to execute instructions guarantees very high performance. An instruction at this point consists of the OPCODE, OPD_1, OPD_2 and IAD fields.
- *SRAM** (static RAM): It contains instructions with one or more unfilled operand fields that are all to be written by one or more instructions currently residing in the PU and/or ERU-SRAM. Therefore, these instructions are going to execute in the very near future. Whenever the PU unit becomes available, an instruction from the ERU-SRAM, which has a large number of recipient instructions in the SRAM*, will go for execution. An instruction at this point consists of the OPCODE, OPD_1, OPD_2, IAD and IID_1 and/or IID_2 fields.
- *ERU-Control Unit (ERU-CU)*: It is the control unit of the ERU. It keeps track each time of the total number of result recipient instructions in the SRAM* for each instruction currently in the ERU-SRAM. It also facilitates data forwarding within the ERU, for recipient instructions in the SRAM*.
- *PU-RAM interface*: It receives all instructions entering to the ERU from the hardware manager. It distributes the instructions accordingly to the PU, ERU-SRAM and SRAM*. When an instruction produces a result, then the PU-RAM interface propagates this token to the EXT-CAHCE and DRAM.

2.2.3 Hardware Manager (HM)

The hardware manager is placed between the ERU and EXT-CACHE. It performs the following tasks:

- It initially sends the head instructions from the EXT-CACHE of the program to the ERU for execution.
- Whenever one of the remaining instructions proceeds to the ERU, it first makes a request to the HM for a virtual ID. This virtual ID will uniquely identify the instruction during its residency in the ERU. The ID is a small number in the range 0 to n-1, where n is the maximum number of instructions that can reside simultaneously in the ERU. Obviously these IDs are recycled. Virtual IDs are assigned to instructions by the HM, in place of their physical ID/address due to the following reasons:
    1. To minimize the required bandwidth between the ERU and external components. This is due to the fact that each instruction carries with it IAD, IID_1 and/or IID_2.
    2. To minimize the size of the ERU internal resources storing the instructions' IDs, especially SRAM* resources.
    3. To minimize the required resources, that processes the ERU resident information.
- It maintains a table that can be accessed to quickly translate on the fly virtual IDs into physical IDs and also vice-versa.

The instruction at this point consists of the OPCODE, OPD_1, OPD_2, IAD and IID_1 and/or IID_2 fields. The IAD, and IID_1 and/or IID_2 fields contain the virtual ID at the ERU side whereas the physical IDs at the EXT-CACHE side.

2.2.4 External Cache (EXT-CACHE)

The external to the execution unit cache (EXT-CACHE) is distributed. It is formed as a collection of Distributed SRAMs (DSRAM). The main-memory is also correspondingly in distributed in nature and formed as a collection of Distributed RAMs (DRAM). For each DRAM module there is one DSRAM module. The EXT-CACHE contains at any time instructions that are to receive a token from instructions residing at that time in the ERU. In fact, three classes of instructions may reside in the EXT-CACHE at any time during the program execution. Those are:

1. Instructions with two unfilled operand fields. One of these fields is to be filled with data that will arrive from an instruction currently in the ERU.

2. Instructions with one unfilled operand field for which the token is to arrive from an instruction currently residing in the ERU. These instructions can not fit in the ERU because the SRAM* is fully occupied.

3. Instructions that are not missing any operands but they can not fit in the ERU-SRAM because it is fully occupied. But such instructions ideally have to be in the ERU.

As already mentioned in our objective, only one copy of each yet to execute instruction is present in the system at any time during the program execution. The part of the program that still needs to be executed is distributed among the off-chip DRAM and EXT-CACHE, and the on-chip ERU-SRAM, SRAM*, and PU. The currently achievable transistor density for chips allows the implementation of large memories to realize the ERU-SRAM, SRAM*, and DSRAM components so that they very rarely overflow. Without hardware faults, there is no possibility for the appearance of deadlocks in this design. Even if the ERU-SRAM is fully occupied at some time, the instructions in it will definitely execute in the near future because the PU will be released soon by the currently executing instructions. If one or more instructions outside of the ERU are ready to execute but can not enter the ERU because the ERU-SRAM is fully occupied, then they wait in the external queue until space is released in the ERU-SRAM. A similar technique is applied if the SRAM* is fully occupied. In fact,

the ERU-SRAM and SRAM* can be combined to single component. For the sake of simplicity, it was proposed to be separate.

For each program memory (main memory) module $DRAM_i$, there is a distinct EXT-CACHE module $DSRAM_i$, for each i = 0, 1, 2, 3 ..., $2^d - 1$. An instruction in the EXT-CACHE consists of the OPCODE, OPD_1, OPD_2, IAD, IID_1, IID_2 and FLAG fields. The operand field locator (OPFL) in each FLAG field indicates instruction dependency or status, i.e. how many operands (nil, one or two) the instruction still needs to go for execution.

As already discussed above each token leaving the ERU also contains the virtual ID of that instruction. The hardware manager changes this virtual ID to the physical ID and then broadcasts this token to all DSRAMs in the EXT-CACHE. The important point is that the DSRAM entries are created dynamically by the hardware manager, have a very short life span, and exists only inside the DSRAM. They are created only when instructions leave for the ERU. That is, it doesn't load into the computer system pointers to parent instructions, which is in line with objectives specified in Chapter 1.

The ERU receives instructions from the hardware manager for execution. Truly the hardware manager forces instructions into the ERU by first storing them into the FIFO buffers and then prompting the ERU to read from these buffers using a very wide bus. Asynchronous communications with appropriate acknowledgments between these two units achieve this task. Therefore, it is not the ERU that fetches instructions for execution, but it is fed with instructions directly by the EXT-CACHE which is a fundamental principle of data-driven paradigm. Here, the program counter is replaced by short IDs i.e. IADs. Fetching shorter IDs is not a heavy penalty to pay for the elimination of the program counter and still the PC- driven CPU requires the implementation of a wide address bus and appropriate control lines. The ERU needs fewer pins to fetch this ID, whereas PC-driven CPUs need more pins to access instructions.

2.2.5 Processor Unit for the PIM (PU-PIM)

Each DRAM has a unique PU-PIM attached to it. This unit carries out the following tasks:

1. It loads the corresponding $DSRAM_i$ with all those instructions from the $DRAM_i$ that are to receive tokens from the instructions leaving for the ERU and also missing data for two operand fields. Also, it always updates appropriately the $DSRAM_i$ directory.

2. It removes instructions from the $DRAM_i$ that are not to be executed further because of loop exiting. The reuse of instructions for the implementation of program loops is addressed later in this Chapter.

3. It maintains three distinct lists of addresses for instructions in the $DRAM_i$, if any, that do not fit in the EXT-CACHE, ERU-SRAM and SRAM*, respectively. These lists are kept in the local $DRAM_i$ for instructions that do not fit in one of these units because of respective overflow.

4. It copies data from tokens broadcast by the ERU via HM into the appropriate fields of instructions appearing in the EXT-CACHE and SRAM* units.

5. It caries out garbage collection in the $DRAM_i$ since the data-driven model of computation necessitates deallocation of the memory space dynamically through instruction dissolvement.

6. It finds the instructions in the $DRAM_i$ and $DSRAM_i$ that are to receive their last operand from instructions leaving for the ERU and forwards them to the HM that finally stores them into the SRAM*.

7. It services requests by the program loader and the operating system for instruction loading and relocation in the $DRAM_i$.

Incorporation of the DRAM i.e. program memory in the D^2-CPU is necessary [1], as accessing data with distinct addresses is quite natural. In fact, there exist many devices that work extreme efficiently using strict memory addressing schemes.

2.2.6 Support for Instruction Relocation

Multiprogramming and virtual memory are now common practices, and very convenient features for the PC-driven paradigm. But both of them require support of instruction relocation. Instruction relocation in the data-driven computation seems to be a very difficult problem to solve because of the need for token passing with ever changing instruction

addresses. [1] Proposes the following solution for the implementation of instruction relocation in a way that token passing using original instruction IDs is still possible.

- The compiler-loader combination assigns the original instruction IDs to correspond to absolute memory addresses. If a memory location is free at that time, then the corresponding instruction, if any, is loaded there. The instruction's context ID, in other words program number, is also stored in the memory along with that instruction. If the memory location is occupied by another instruction, then the former instruction is relocated early according to the method described below.
- A distinct ID memory module ID_MEM_i is associated with each $DRAM_i$. The two memory modules have the same location. The j^{th} entry in ID_MEM contains the starting address of a hash table containing pointers to all instructions with original ID equal to j, but with different context IDs, for $j = 0, 1, 2 \ldots 2^m - 1$. When an instruction with original ID 'k' relocates in the DRAM, then the respective PU-PIM unit stores in the hash table pointed at by the value in address 'k' of the ID_MEM the context ID and the new address of this instruction.
- The PU-PIM unit keeps track of the location of all instructions in the DRAM. It updates the hash table whenever an instruction is relocated. This scheme implements memory indirect addressing for token propagation with maximum flexibility.

2.2.7 Support for Exceptions

Exception are of two types, either software or hardware. The D^2-CPU [1] handles both in different way. If it is a software exception and code determines that an erroneous result will show up with the execution of such an instruction, then a thread of instructions are activated to deal with this problem. This thread basically removes faulty instructions from the system. This leads to run time availability of some exception routines. It is not necessary to halt the

execution of instructions that do not belong to the exception routine. If required, however, because of high priority, then the HM can temporarily ignore all instructions in the EXT-CACHE with context ID different from that of the exception determining instruction.

For hardware exceptions, exception routines are initially stored in the DRAM memory. The HM receives the exception request along with an exception ID. This ID uniquely determines the address of the first instruction in the exception routine. The hardware manager forces the PU-PIM to make a copy of the exception routine code, sends the activation token to the first instruction and disables all transfers to the ERU of instructions that have different context ID than this exception ID. It also sends this exception ID to the ERU to disable the execution of instructions with different context IDs. Every exception routine contains a last instruction that upon execution forces the hardware manager to enable all context IDs for the resumption of program execution.

2.2.8 Support for Loop Implementation

A bit in each instruction can indicate its inclusion in a loop, so that the instruction can be preserved at the end of its execution for future executions. Only its operand fields are emptied, if necessary, after each execution. Upon exiting a loop, the last instruction sends a special dissolve token to the first instruction in a special routine that removes all loop instructions from the memory; only the PU-PIMs are involved in this process. As far as conditional branching, instructions that are not executed are dissolved similarly by special routines.

Though the methods described in [1] are adequate, a lot of work may be needed in instruction relocation, exceptions and in loop implementations.

The D^2-CPU design has some similarities with the VLIW architecture. Similar to VLIW, D^2-CPU has a wide instruction bus, long instructions and many instructions can travel

at a time from the HM to the ERU. Since VLIW is a PC-driven architecture, there are lots of redundant instructions. Secondly, any code is portable with D^2-CPU, as there is no need for the compiler to group together simple instructions into large ones as needed in VLIW. So there is no need of expensive compilers for the D^2-CPU model.

The next Chapter discussed in depth a practical approach to D^2-CPU design. The D^2-CPU design is implemented on a FPGA, which alters some of the architectural part proposed above. Also, some trivial architectural part is modified as first priority of this thesis work is to prove the feasibility of the pure data-driven model on FPGAs. Instruction relocation and exception handling are not implemented in this work, whereas loop support is implemented.

CHAPTER 3

DESIGN OF A FPGA BASED D^2-CPU

3.1 Introduction

This Chapter deals with a practical implementation of the D^2-CPU design. The first part explains architectural details. This part demonstrates how the D^2-CPU design proposed in Chapter 2 will take shape into reality and also discuses difficulties and solutions to them. It also shows how some difficulties lead to few minor changes in the proposed architecture, keeping its purity, as a data-driven machine, intact. The second part mainly deals with its implementation on an FPGA board. Required software and hardware issues are dealt with detail in this part.

3.2 Implementation of the D^2-CPU

3.2.1 Instruction Set Format

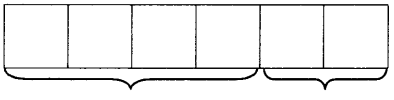
As already discussed, the instruction format changes at each level of the D^2-CPU design. All fields used by the instructions at different levels are explained below. The instruction format at each level is dealt with detail at the respective design description in this Chapter.

There are in all 12 fields:

- Operand1 (OPD_1): It holds the first operand. As specified in [1], there are two operand fields either for unary or binary operations. Each is 16 bits wide. Either this field is already filled at compile time or required data from some other instruction, whose address is specified in IID_1. The length of IID_1 depends upon instruction locality, as either it presents a virtual ID or physical ID. In the first case it is six bits wide and in the second it is seven. Each time, when a token is broadcasted from the

ERU, its address field (namely IAD) is compared with IID_1, and if matched the OPD_1 is filled with data associated with the token. As a general rule, for unary operations like shift, only this operand is used.

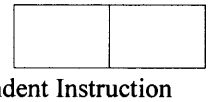
- Operand2 (OPD_2): It holds the second operand. This is 16 bits wide. If required, it is either supposed to be filled at compile time or depends on another instruction whose address is specified by IID_2. IID_2 is seven bits wide.
- Operation Code (OPCODE): It is the opcode of the instruction to be executed. There are six bits in this field, out of which the last two bits are reserved to indicate which pipeline it belongs to, i.e. adder, multiplier or shift/logical/comparator. The remaining four bits are used for different shift/logical/comparator instructions. The details of the opcodes and their use in dataflow graphs are discussed in the next Chapter.



0000: AND	0001: OR	0010: NAND	00: Invaild
0011: NOR	0100: XOR	0101: NOT	01: ADD
0110: EQT	0111: NEQT	1000: GT	10: MUL
1001: LT	1010: GET	1011: LET	11: S/L/C
1100: SHL	1101: SHR	1110: RAL	
1111: RAR			

Figure 3.1 OPCODE Format.

- Instruction Address (IAD): Except DRAM, this field is identity for instructions at the remaining levels. Even a token contains this field, where it signifies the address of the result producing instruction. The number of bits in IAD is either six or seven depending upon virtual ID or physical ID, respectively.



00: Independent Instruction
01: OPD1 Needed, IID1 Valid
10: OPD2 Needed, IID2 Valid
11: Both OPDs Needed, IID1 & IID2 Valid

Figure 3.2 OPFL Format.

- Operand Field Locator (OPFL): This is a two-bit field and decides the dependency of instructions. IID_1 and/or IID_2 provide instruction addresses on which OPD_1 and/or OPD_2 are dependant.
- Clause Answer (CAN): A bit flag which holds the boolean answer for a clause on which the particular instruction depends upon. If the instruction does not depend upon any clause, then CAN is set to '1' and the clause required (CR) bit is set to '0' at compile time. Otherwise CAN is set to '0' and CR to '1'. Clause address (CAD) provides the instruction address on which CAN depends. Whenever a token is propagated its address is compared to CAD, and if matched, the result's last bit is stored in CAN. The instruction will go for execution only if the CAN bit is set to '1'. Obviously CAD is the address of such an instruction which provides a boolean answer, like EQT, NEQT, LT, GT etc.

CAD (7)	CAN(1)	CR(1)

CR	CAN	CAD	Status
0	1	Invalid	Ready
1	0	Valid	Not-ready
1	1	Valid	Ready
0	0	Invalid	Invalid

Figure 3.3 CAN, CAD, and CR Format.

- Valid Bit (VB): This is a one-bit flag. '1' in this field indicates the instruction is valid and '0' indicates the instruction is invalid. This is a peculiar flag, as this is present at every level in the D^2-CPU design, including DRAM. As all types of memories are implemented in the conventional style of cache design, this bit indicates the validity of the instruction.
- Loop (LP): This is a two-bit field. This field is used in particular for implementing loop structures in the D^2-CPU design. "00" indicates the instruction is not involved in any loop structure. "01" in those fields indicates a "merge" node, "11" indicates a

"lock" node, and "10" is assigned to a "switch" node in a loop i.e. a conditional instruction that immediately follows lock node. Details about loop implementation, switch node and merge node are explained in the next Chapter for dataflow graphs and programming with D^2-CPU design.

- Operand Reuse (ORE): This is also a two-bit field. This field is for instructions, which are involved in loop execution. This field affects the OPFL field. After an instruction inside the loop is sent for execution, its OPFL changes depending upon ORE. '1' in any field of ORE indicates the respective OPD has to be reused, so the respective OPFL field is set to '0'. Thus, in the next iteration it will keep the respective OPDs intact. Details are discussed in the next Chapter.

3.2.2 ERU Design

The ERU is basically divided into three parts; the SRAM* and ERU-SRAM memories and the functional units. As dataflow machines are inherent parallel machines, it doesn't make any sense to use single functional unit. As also proposed in [1], one adder, one multiplier and one logic unit are implemented. As the multiplier and adder will take more time to execute than any shift, compare or logical operation, the latter three functionalities are grouped together in one logic unit. By sticking to the basic, "simple is fast" of superscalar RISC principle, three pipelines are implemented, one for each functional unit. So, the SRAM* and ERU-SRAM are also divided into three parts. A First-In-First-Out (FIFO) buffer is used, for tokens propagated from the ERU to the off-chip memory systems. Figure-3.4 shows an architectural overview of the ERU. The design details of each part in the ERU are discussed below:

- Functional Unit (FU): As already mentioned, three functional units are implemented. 16-bit functional units are generally implemented. For the sake of simplicity, a non-

pipelined 8-bit multiplier available by Xilinx as a standard component is used. Though it is an 18 * 18 multiplier, only the last 8 bits are used, tying the remaining bits to '0'. It produces a 16-bit result. The adder also is a non-pipelined unit. The adder is a 16-bit unit that produces 16-bit results. The logical unit performs shift, rotate, logical functions (and, or, not, etc.), and compare functions (equal to, less than, greater than, etc.). For unary shift and rotate functions only OPD_1 is considered. As all compare instructions are boolean in nature, they produce results as '0' or '1' and this is assigned to the 0^{th} bit of the result. This result is used for clause answers; programming is discussed in detail in the next Chapter.

The input to the functional units is a 44-bit wide instruction, consisting of an OPCODE (6-bit), OPD_1 (16-bit), OPD_2 (16-bit), and IAD (6-bit) whereas the output is 22-bit wide token, comprising of IAD and RESULT (16-bit). Each unit puts its result on a 66-bit wide data bus and let FIFO know about it.

- ERU-SRAM: The ERU-SRAM contains ready to execute instructions. The instructions with all required operands filled, that can not execute due to the unavailability of a functional unit are located inside ERU-SRAM. So, the ERU-SRAM can be of any size, but for the sake of simplicity and FPGA realization, minimum of two instructions are assumed in the ERU-SRAM. As mentioned already, the valid bit indicates the resident instruction's validity; the reaming fields constitute the OPCODE (6-bit), OPD_1 (16-bit), OPD_2 (16-bit), and IAD (6-bit).

  A count filed, associated with each instruction in the ERU-SRAM, where 2-bit count indicates the number of instructions in the SRAM* depending upon the particular instruction, is also implemented. Since relatively large logic is required to

implement it, it is not included in this version of the D^2-CPU design due to our FPGA realization.

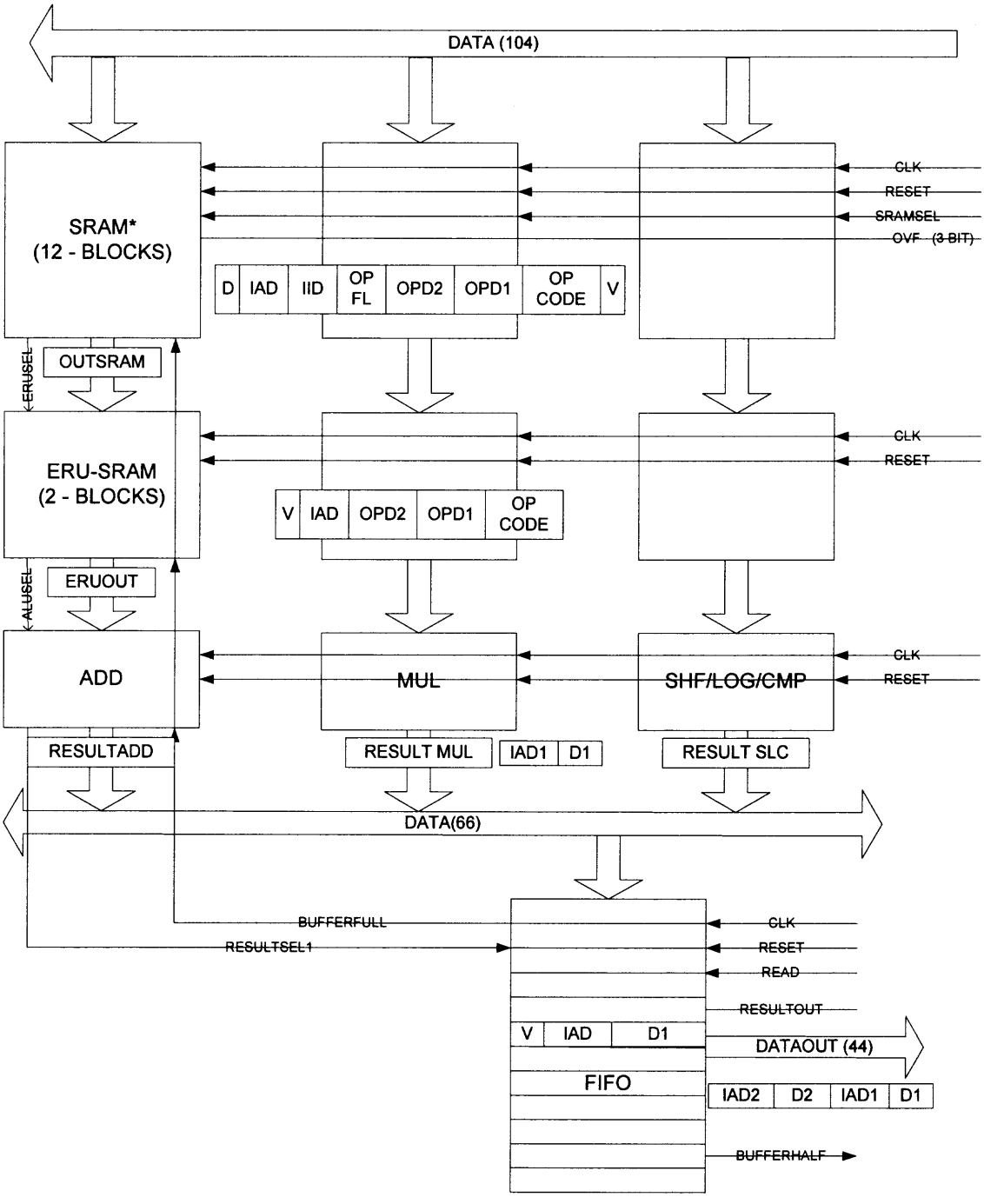


Figure 3.4 The ERU Design.

- SRAM*: The SRAM* consists of instructions with either an unfilled operand field or filled with both/required operands. Each SRAM* has 12 blocks (instructions) in it. The instruction format is as shown in Figure 3.4. A single bit field D determines a dependency whereas OPFL determines which operand field needs to be filled with a token from the instruction IID. Table 3.1 below clears this functionality. The valid bit as usual indicates validity of the instruction.

Table 3.1 SRAM* Instruction Dependency

D	IID	OPFL	STATUS
0	Invalid	Invalid	Independent
1	Valid	0	OPD1 Needed
1	Valid	1	OPD2 Needed

In general, the SRAM* closely resembles in functionality with a large number of reservation stations in Tomasulo's algorithm and it truly makes the whole system work "totally out of order". This is not restricted to only the CPU in the D^2-CPU design, as it is generally implemented in new microprocessors (e.g. Intel Pentium-IV), where execution becomes out of order for only the instructions residing in the CPU. The EXT-CACHE and DRAM look like as a large extension of reservation stations. This idea makes the D^2-CPU design a pure data-driven processor, but complicates it.

Each SRAM* receives two instructions from HM on a very wide (104-bit) bus. Both instructions strictly do not belong to the same functional unit. Each SRAM* unit scans just the last two bits of each instruction to find out its place. Each SRAM* indicates its empty status to the memory system through overflow (OVF) signal.

- FIFO Buffer: FIFO plays two important roles in the overall working of the system; first, the FIFO buffer plays a cushion between totally unreliable (in number) ERU outputs and consistent inputs to the memory system by means of tokens. Dataflow machines are runaway machines; and firing one instruction subsequently fires many instructions in different parts of the code. This causes any number from 0 to 3 outputs from functional units at any clock cycle. Whereas the in-out data bus to ERU is a bidirectional data bus, it needs some kind of consistency in its operation. FIFO provides this consistency in the input and output of the ERU. The FIFO has bufferhalf and bufferfull signals which play an important role in the overall working of the system as follows:

  Bufferhalf: This signal indicates its half filled status to the memory system, which in turn stops sending instructions to ERU and receives tokens from ERU.

  Bufferfull: This signal indicates its full status to the all functional units, ERU-SRAMs, and SRAM*s, so that they will temporarily stop giving outputs and just insert "bubbles" in the pipeline. Still SRAM* keeps on the receiving instructions, till it gets filled.

  Secondly, the FIFO buffer provides the necessary scanner for each SRAM* unit. At each clock pulse, an unfilled instruction operand in the SRAM* gets its operand if its IID field matches with any IAD field in the FIFO. This is an important mechanism to keep consistency in data in and around ERU.

  The FIFO sends two tokens out to the hardware manager, whenever there is no data in to the ERU. This is achieved with the help of the bufferhalf signal as explained already.

All the activities inside the ERU are synchronized with the clock, even keeping the necessary handshaking signals. Synchronization is implemented for the sake of simplicity, whereas handshaking signals provide more flexibility. This synchronization makes each SRAM*, ERU-SRAM, functional unit, and FIFO buffer very similar to a conventional four-stage pipeline; a very important difference is that it is "totally out of order". Another important feature of the ERU is that it doesn't have any central control unit, and all the controls are distributed in each unit. Each unit (e.g. SRAM*) has an autonomous behavior for processing of data and just depends for data upon another unit. This feature achieves our objective of distributed control in the D^2-CPU design.

In short, the ERU of the D^2-CPU processor is a superscalar, pipelined, and executed totally out of order assuming distributed control.

3.2.3 Hardware Manager (HM)

Figure 3.5 shows the hardware manager. The basic function of the HM is to convert on the fly a physical address to a virtual one and vice-versa. The HM accepts two instructions at a time from the memory system on a bidirectional data bus (108 bits wide) and converts its required IAD and IID fields from the physical to a virtual address using a table. Then, it puts the same instructions to the input bus of the ERU. Similarly, it accepts two tokens from the ERU, converts the IAD fields to a physical address, makes two copies of it and puts it on the same bidirectional data bus. The reason for two copies is explained in the memory system Section for better understanding.

Let us justify the bidirectional data bus. In fact unidirectional data buses can be used in place of one bidirectional data bus, which would have increased the overall through put of the system. As the data consistency is the main issue in the data-driven design. A mismatch between an instruction and its required token can lead to non-execution of such an instruction

and some serious flaws. Even this flaw can be corrected using buffers at each communication level with two unidirectional buses, in place of a bidirectional bus, where these buffers provide a guarantee for data consistency. Again, it is a tradeoff between logic required to implement and the design throughput. For FPGA realization, a bidirectional data bus is used, to reduce logic required to implement the buffers.

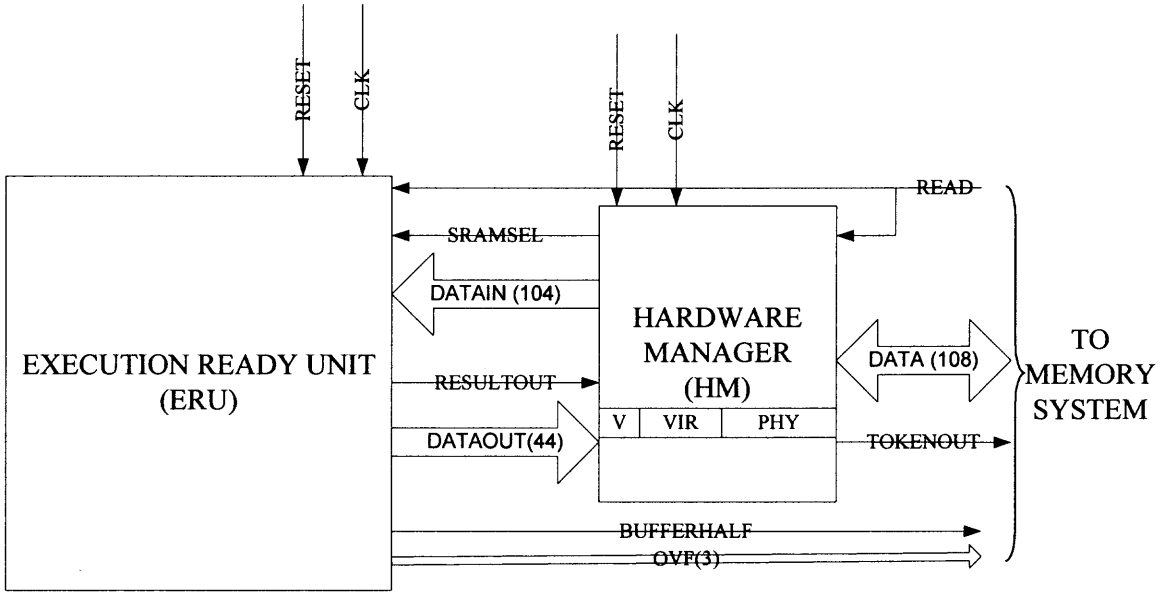


Figure 3.5 Hardware Manager.

From here onwards memory system part is assumed to contain an out-buffer, two cache memories, and the corresponding two main memories.

3.2.4 Out-Buffer

The Out-buffer is just an extension to the SRAM* in the memory system. It can contain 12 instructions with the same format as that for the SRAM*; the only difference is that IAD and IID represent physical addresses in palace of virtual. The format is shown in Figure 3.5. The Out-buffer receives one instruction from each $DSRAM_i$ and $DRAM_i$ pair of memory. It's an FIFO buffer which decides, depending upon the status of the overflow (OVF) signal from the SRAM* which instructions should be put on the data bus to the ERU. It makes sure that no

two instructions requiring the same fictional units will be available on the data bus. If no such two instructions are available then it will just assign one instruction to the data bus; such cases should be avoided by properly loading the program memory modules.

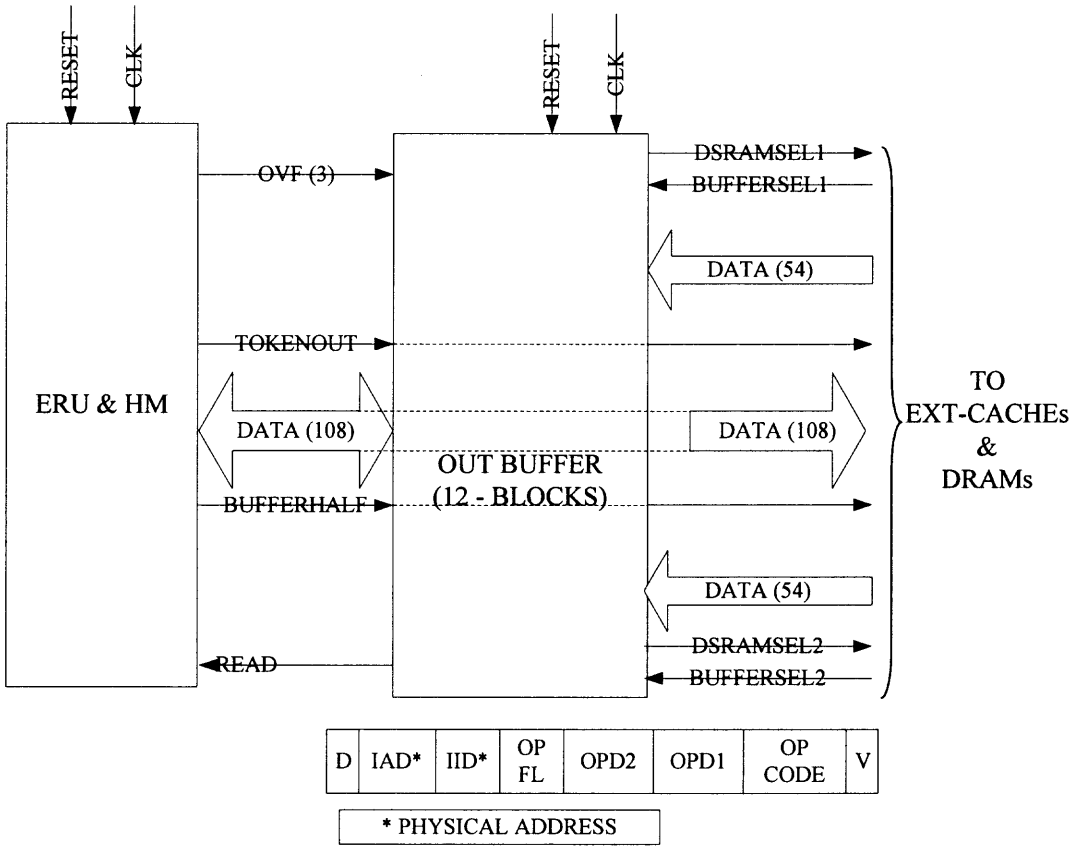


Figure 3.6 Out-Buffer.

From here onwards there will only be unidirectional buses (54-bits wide) from each lower level of the memory to a higher level as no data flows from the opposite way. Thus, there will be a unidirectional bus from the $DRAM_i$ to the $DSRAM_i$ and from the $DSRAM_i$ to the Out-buffer. All such buses end at a single Out-buffer. Tokens flow directly from the ERU to each memory module (e.g. $DRAM_i$, $DSRAM_i$ etc.) via a unidirectional bus (108-bit), which is just a continuation of the data bus between the ERU and Out-buffer. The respective tokenout and bufferhalf signals accompany this bus. Data consistency is maintained by stopping all the transactions between any memory modules, whenever a token is out from the

ERU. This part is very important for correct functionality of the D^2-CPU and tokenout and bufferhalf signals achieve this objective.

This 108-bit unidirectional data bus is further divided into two 54-bit unidirectional data buses which carry two identical tokens to two different pairs of the memory module. That's the reason why the HM manager makes two copies of the same token and put them on the 108-bits wide data bus. The functionality of the pair of memory modules and instruction format in each is discussed further. This special instruction format takes advantage of intelligent memories and each memory controller (PU-PIM_i) uses this format to feed instructions further in the system hierarchy and to the ERU ultimately.

3.2.5 EXT-CACHE ($DSRAM_i$)

Figure 3.7 shows two pairs of the $DRAM_i$ and $DSRAM_i$ and also includes their instruction formats at the bottom. As proposed in [1], there can be multiple pairs of such memory modules, but two are used in our FPGA implementation. Each block consists of one instruction with the OPCODE (6-bit), OPD_1 (16-bit), OPD_2 (16-bit), IAD (7-bit), IID_1 (7-bit), IID_2 (7-bit), OPFL (2-bit), and VB (1-bit) fields. There are two approaches to implement such caches, as both needs extensive support from the main memory. The main memory construction will be discussed below whereas these two approaches are immediately discussed here. The first approach is multithreading, while the second one is the pure data-driven approach.

- *Multithreading* approach: In this case, each cache consists of different threads or blocks, where each thread consists of four (in fact, any small number) instructions. Out of these four, the first instruction will be always an independent instruction, i.e. an instruction with required operands already available at compile time and the remaining three instructions either can depend upon the first or may be

interdependent. But none of them depends upon any other instruction out of that thread. So, each thread is totally independent of each other for data but can be depended for a clause on one another.

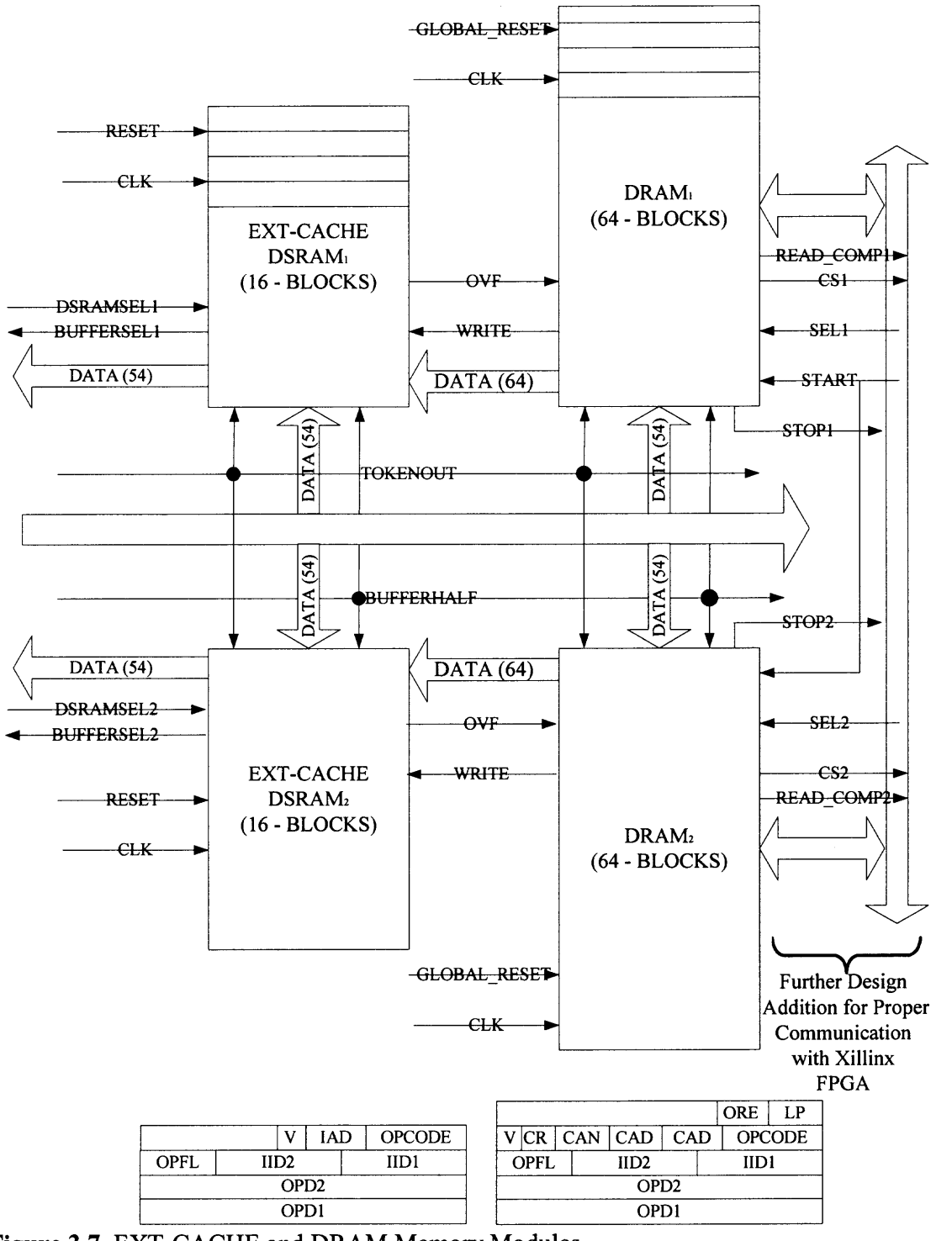


Figure 3.7 EXT-CACHE and DRAM Memory Modules.

This clause is totally taken care in main memory and when any thread complies with this condition in main memory, can be moved further to EXT-CACHE. So each EXT-CACHE consists of set of totally independent threads.

Advantages: By doing this, the design complexity considerably decreases. A conventional cache design can be directly implemented in which each thread is considered as a block with one tag. An example of a fully associative cache is shown in Figure 3.8. It also shows the IAD/IID format once the instruction leaves the EXT-CACHE.

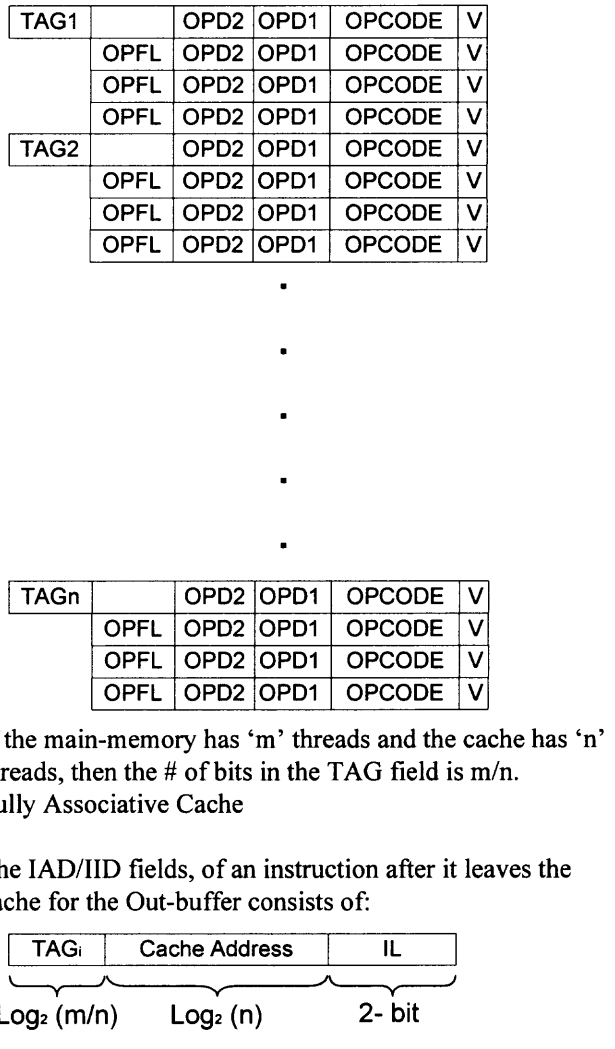


Figure 3.8 Multithreading Approach to EXT-CACHE.

So by keeping the ERU and Out-buffer design the same we can still enjoy the use of conventional proven cache designs. Each OPFL locator indicates dependency; if any bit in OPFL is '1', the last 2-bit of the corresponding OPD will give the address of the instruction, out of the remaining three instructions in that particular thread, which needs to execute before these instructions. In fact a lot of memory is saved in two IID and one IAD fields. Also, whenever any token is out, it just need to match tag address of only one thread, and if that matches then the particular result will be dropped in any required operand field of the remaining unexecuted instructions in that thread. So a lot of saving in the logic is achieved.

- Pure *Data-driven* Approach: In the multithreading approach flexibility is lost as dependency remains only inside a thread. But our approach develops a pure dataflow structure without any relevant compromise. So as shown in Figure 3.9, a block consists of only one instruction in our implementation. Each instruction belongs to one of the three classes already defined in Chapter 2.

  A total of 16 such blocks are implemented, which can operate totally in parallel. Depending upon its internal states, defined by its flags, each block will "inform" the cache-controller about its readiness. Then, the cache-controller will choose on a first-come-first-serves basis an instruction to forward to the Out-buffer. Similarly, tokens propagated by ERU also broadcasted on a same data bus. Each block, if required, compares the IAD of the token with its IIDs and if a match occurs, fills the respective OPD, indicated by OPFL. This gives maximum flexibility, which is used dynamically to exploit the full level of parallelism in the application program.

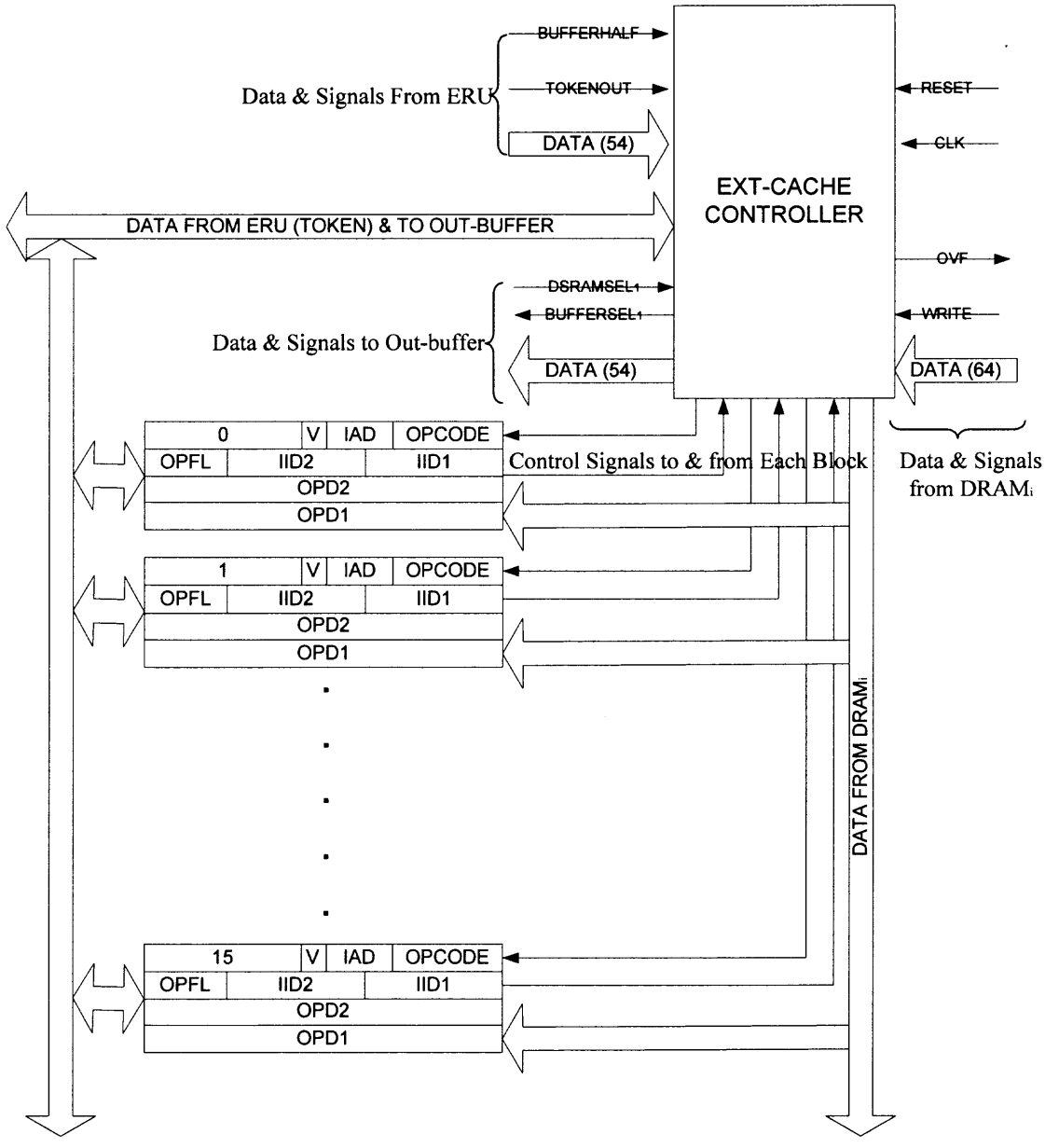


Figure 3.9 Pure Data-driven Approach for the EXT-CACHE.

3.2.6 Main-Memory and PU-PIM$_i$ (DRAM$_i$)

The main memory has same structure with the EXT-CACHE. As already mentioned, to implement any of the above two approaches respective main-memory support is needed in the same style as that for EXT-CACHE. As the pure data-driven approach is used, here we mainly discuss its efficient implementation in the main-memory.

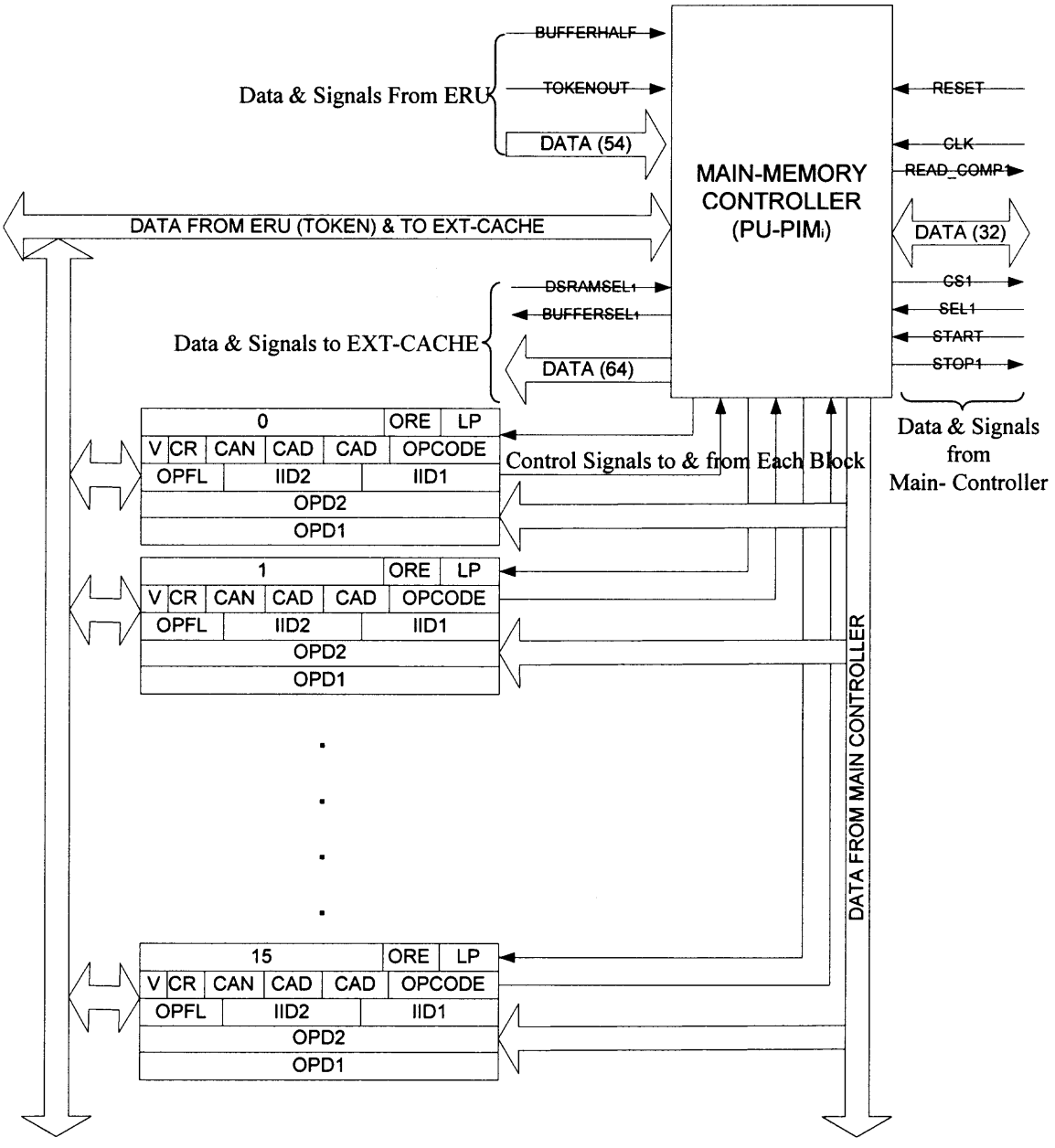


Figure 3.10 Pure Data-driven Approach to Main-Memory.

Figure 3.10 shows the actual implementation of $DRAM_i$. The structure is the same to that of the EXT-CACHE. The only difference is instruction format, which supports clauses and static loop implantation with the help of the Lock method. Both these uses are discussed in the next Chapter. Whenever a block is ready, it will indicate this to the PU-PIM (main-memory controller) with control signals and then PU-PIM will decide, on a first-come-first-

served basis, which instruction should go to EXT-CACHE and should modify the flags accordingly.

As shown in Figure 3.10, additional data and control signal communications are needed with the Xilinx Virtex-II Block RAM for reading and writing data to the DRAMs of the D^2-CPU design from the host CPU to test our architecture. These additional architectural features are discussed below.

3.2.7 Main-Controller and the Xilinx Virtex-II Block RAM

Figure 3.11 shows the implementation of the main-controller and the Xilinx Virtex-II Block RAM. This figure is a continuation of Figure 3.7, in the complete system architecture.

The block RAM is a standard Virtex-II component available on the FPGA. There are a total of 144 block RAMs available on the Virtex-II FPGA. We used just one 36 * 512 RAM, which has 512 registers each of 36-bit wide. Out of these 36, 32-bits are used for storing data and the remaining four bits are used for parity. The main-controller can read from and write data to this RAM, using the signal shown in Figure 3.11.

Reset, clock and Lad Bus interface signals are standard components available by Annapolis Microsystems. Details of the Annapolis board are in follow in this Chapter. Whenever this whole architecture is configured in the FPGA, we can read from and write to the block RAM from the host computer system. In turn, when the global-reset is used, the main-controller will start reading from the block RAM and fill the $DRAM_i$. Then it will issue a local reset to the D^2-CPU design implemented in the FPGA. The D^2-CPU then will start its execution. After completion of the program stored in the DRAM, it will let main-controller know about it. The main-controller now will read the results from the DRAM and write them into the same block RAM. We can then read these results from the block RAM through the host-CPU using the LAD bus interface of the Annapolis board.

This is the overall architecture of the D^2-CPU. We will now discuss the details of the Annapolis board and the actual design cycle for FPGA implementation.

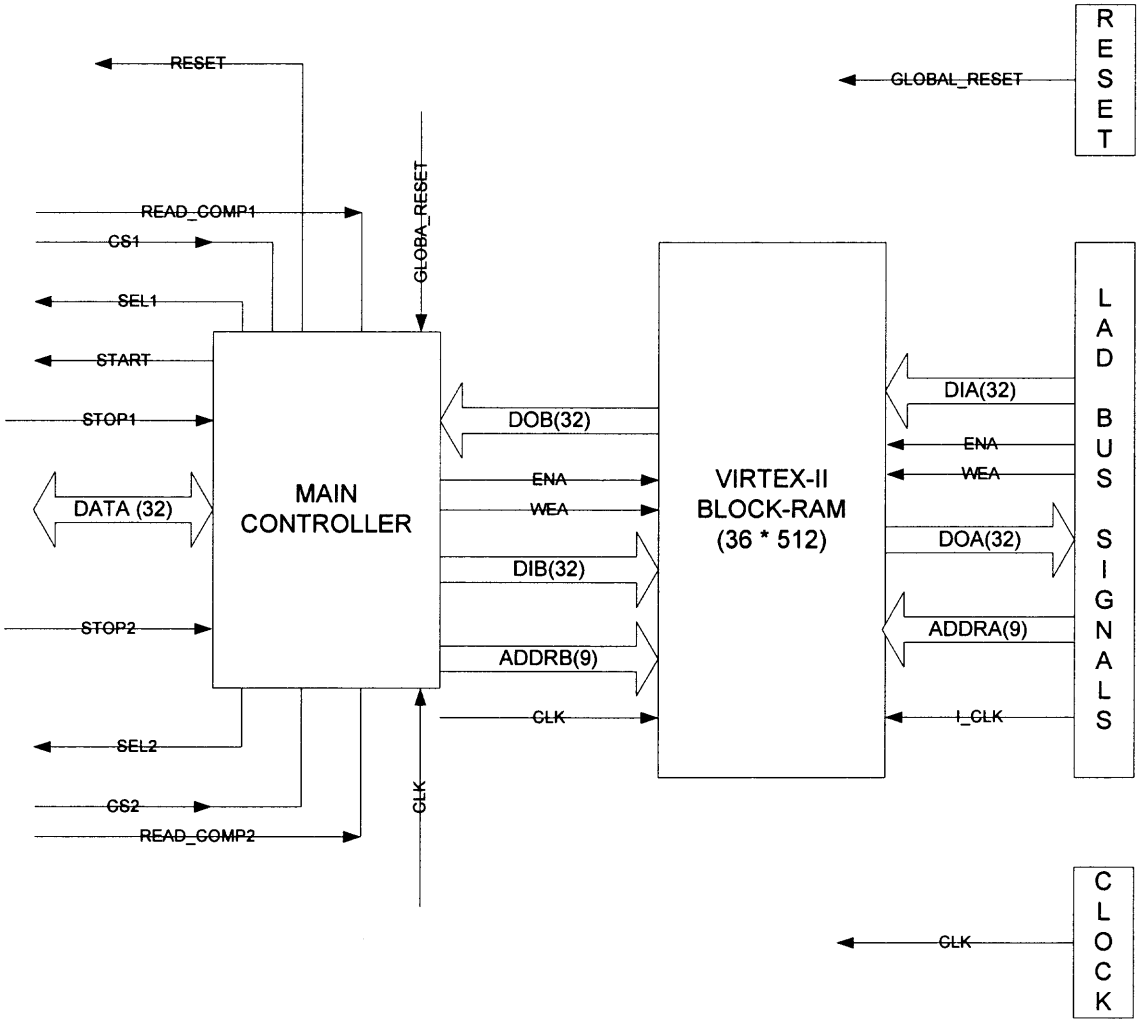


Figure 3.11 The main-controller and the Xilinx Virtex-II Block RAM.

3.3 Overview of the Wildstar-II Board

The Annapolis Microsystems high-performance Wildstar-II board combines the high density of reconfigurable system gates from Xilinxs Virtex-II FPGAs with very large memory and high I/O bandwidth. We chose the PCI-based Wildstar-II board as its two XC2V6000 Virtex-II FPGAs can deliver great levels of processing power, and its substantial on-board DDR or

DDR-II SRAM and DDR DRAM memories make it an ideal choice for building custom computing machines.

Figure 3.12 shows the block diagram of the Annapolis Microsystems's Wildstar-II /PCI board. It uses two Xilinx XC2V6000 FPGAs, with up to 16 million system gates each. A host computer can communicate with the board via the PCI interface. The PCI bus interface communicates with the Wildstar-II board's PCI controller. The PCI controller has access to the FPGAs and Euro I/O cards using the Local Address Data (LAD) bus. The host has direct register access and communicates with the FPGAs and the I/O cards over the LAD bus. It has 12Mbytes of DDR II SRAM and 128MB of DDR SDRAM on the board. It has a programmable Flash bank per FPGA for image storage.

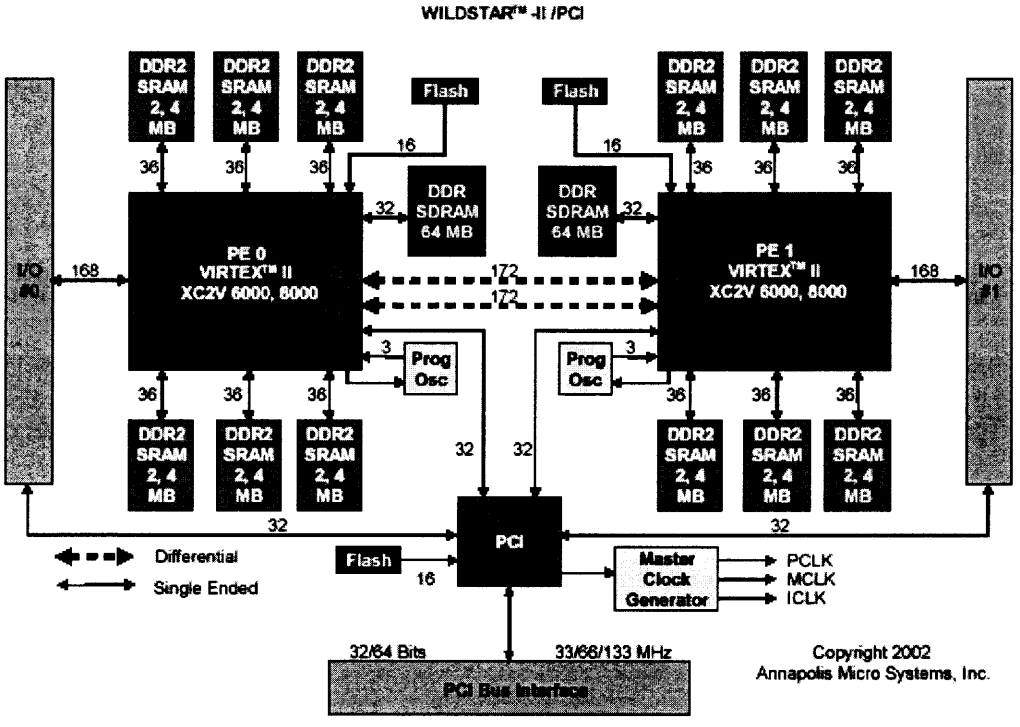


Figure 3.12 Wildstar-II/PCI Block Diagram [17].

Each processing module, as shown in Figure 3.13 consists of a Xilinx Virtex-II FPGA, six independent DDR2 SRAM ports, one bulk DDR DRAM port, three input/output

Transmit (T_x) and Receive (R_x) clocks, and a 32-bit LAD bus. It also consists of flash storage for multiple FPGA images, three global clocks, three user clocks and three user LEDs.

The Wildstar-II board has two types of clocks: the global board clocks MCLK, PCLK, ICLK, and the local clocks for each FPGA consisting of ACLK, BCLK and CCLK. MCLK is differential and asynchronous to PCLK. It is configurable through the Wildstar-II host software. PCLK is differential and asynchronous to MCLK, and is configurable through the Wildstar-II host software. ICLK is the Local Address Data Bus clock. It is fixed at 132MHz and the FPGA uses this clock to interface to the PCI controller for host access via the LAD bus.

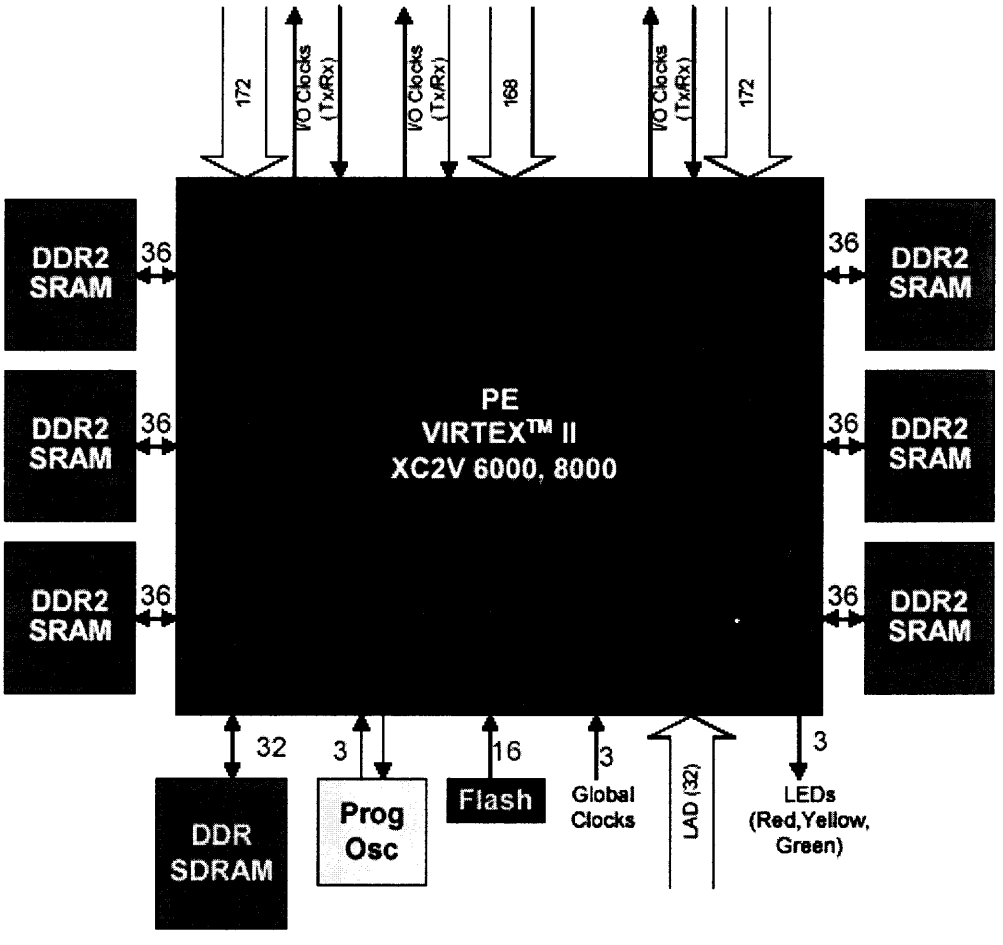


Figure 3.13 Wildstar-II Processing Module [17].

The host communicates with the board using Wildstar-II Application Programming Interfaces (APIs). The host software includes Wildstar-II APIs, device drivers, a run time library and utilities, which enable efficient communication between the host and the board through the PCI bus. The APIs are a set of functions coded in the C language allowing communication between an application and the Wildstar-II run-time library. Many APIs are provided to open the board, close the board, program the FPGAs, deprogram the FPGAs, write onto them and read from them.

3.4 Design Flow and Implementation

The D^2-CPU machine is designed using the VHDL hardware description language. Also, the design of the main-controller and glue logic to interface the LAD bus is done in VHDL. During this design, different tools at various levels of integration are used. We have followed the standard Xilinx design flow in generating the complete system as shown in Figure 3.14. We discuss below the details of design flow. Figure 3.15 shows the basic steps in the Xilinx standard design flow.

The following are the steps followed in the FPGA design flow:

1. The design of all modules required by the D^2-CPU design is done using a synthesizable subset of the VHDL language. The coding and compilation are done using the Mentor Graphics Modelsim simulator.

2. The functional simulation is performed using the Modelsim simulator. Many test benches are developed to test the D^2-CPU design using simulation. All the instructions for the D^2-CPU are tested using test benches.

3. Annapolis standard interfaces, like reset, clock and LAD bus interfaces, available in VHDL are included at this stage. Steps one and two are performed again for design verification.

4. These VHDL files are given as input to the Synplify Pro synthesis tool. During synthesis the behavioral description in the HDL file is translated into a structural netlist and the design is optimized for the Xilinx device XC2V6000. This generates a netlist in the EDIF (Electronic Design Interchange Format) and VHDL formats.

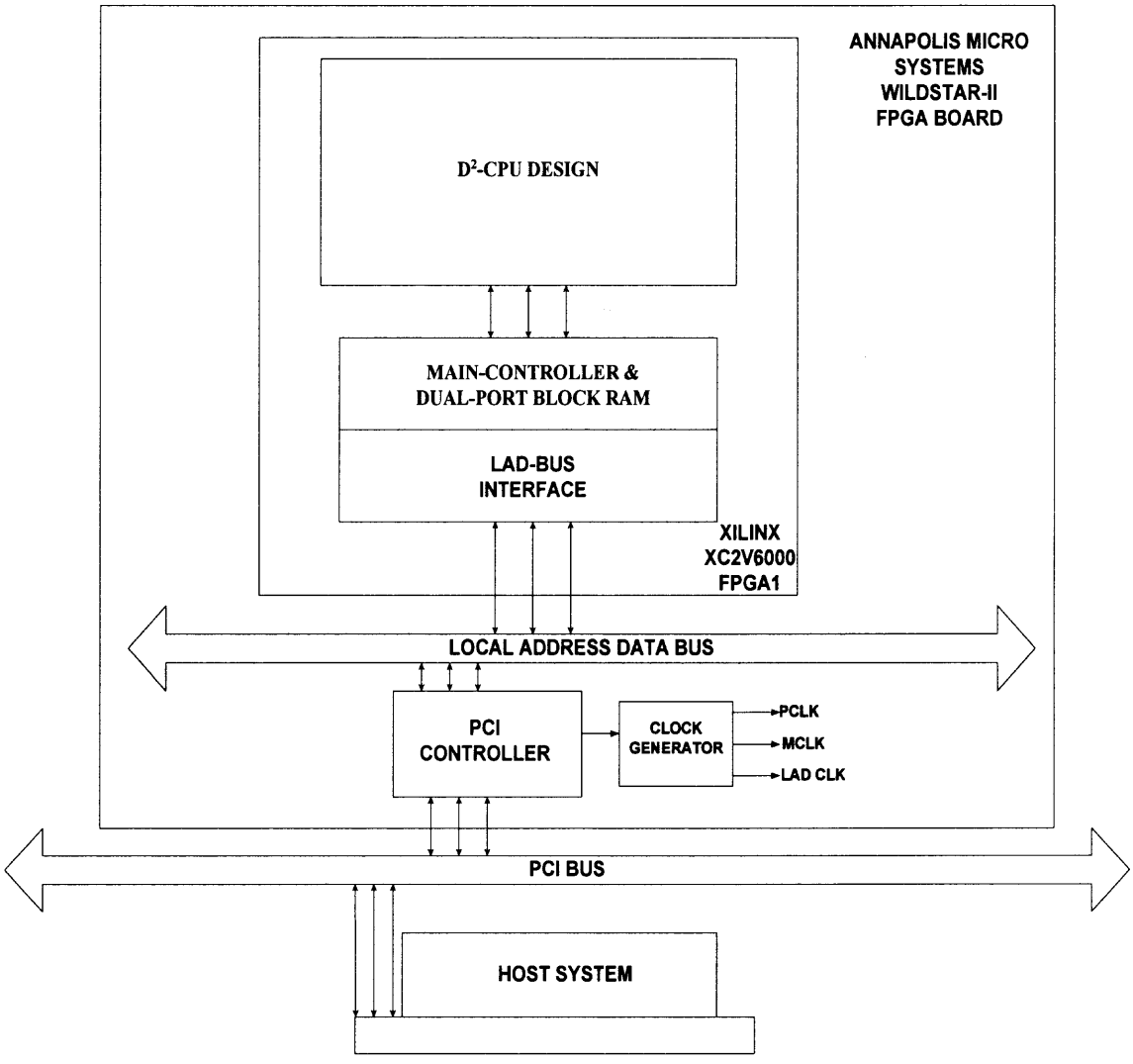


Figure 3.14 Communicating with the Host System.

5. The output VHDL file from the synthesis tool is used to verify the functionality by doing post synthesis simulation using the Modelsim simulator.

6. The netlist EDIF file is given to the implementation tools of the Xilinx ISE (5.1-I). This step consists of translation, mapping, placing and routing, and bit stream generation. The design implementation begins with the mapping or fitting of the logical design file to a specific device, and is complete when the physical design is completely routed and a bitstream is generated. Timing and static simulations are done to verify the functionality. This tool generates an X86 file which is used to program the FPGA.
7. Then a program in the C language is used. In this program different standard API functions available by Annapolis Microsystems are used for communication between the host system and the board. During execution of this program the host CPU programs the FPGA using available X86 format file, write the program data on the block RAM, reset the whole board and after finite given delay the results are read

back from block RAM. These results are compared with the required for correct functionality of the whole system.

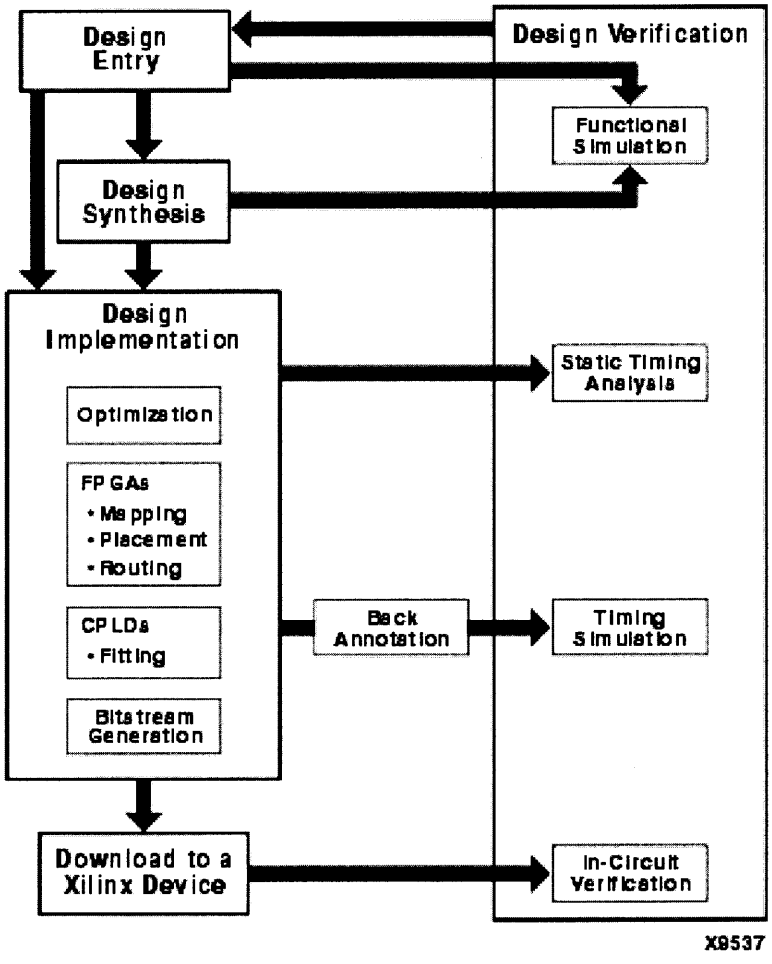


Figure 3.15 FPGA design flow [18].

All these steps are followed in a general design methodology to program the FPGA. A small change in VHDL for correct execution leads to again start the design cycle from scratch. This is done till we get the correct results.

Appendix consists of timing report, device utilization summary, design summary, mapping report, and final place and route report. The timing report was generated by Synplify the synthesis tool and gives the maximum frequency at which this particular design

can run. For this design, Synplify estimates 31.4 MHz. Therefore, 24 MHz is set to start. The device utilization summary gives the amount of logic used by this design, which includes the number of Block RAMs, slices, LUTs, and CLBs used. The place and route report indicates the design complexity by means of time required to place and route the particular design. It generates the total time required to place and route and also the detailed floor plan within FPGA. Mapping and place and route can be done manually for optimized use of logic, but it takes a lot of time. Automatic mapping and place and route are used.

In the next Chapter dataflow graphs and programming with the D^2-CPU is discussed. The results and comparative analysis are discussed in Chapter 5.

CHAPTER 4

DATA FLOW GRAPHS AND PROGRAMMING WITH THE D^2-CPU

4.1 Dataflow Graphs

4.1.1 Introduction

In dataflow machines, programs are stored in totally unconventional style. There are a lot of different dataflow machines available at the research level. Although every machine has a different programming language, their basics are the same. They share many common principles. A short summary of dataflow programs and common terminology used is discussed below which follows the programming with the D^2-CPU.

4.1.2 Dataflow Programs

Figure 4.1 shows a comparison between conventional control flow programming and dataflow programming. There are two types of pointers for control flow, control flow pointers and data flow pointers. Both have to be specified explicitly in the program. The program counter (PC) takes care of control flow and thus the correct execution of the program. This type of execution fits perfectly with sequential programming, but for parallel programming the shared data memory has to be managed cautiously for correct execution of the program. Truly, if data dependencies are preserved then there is no need of control flow pointers, and then we can combine the instruction and data memories to form a single global memory. As there are only dataflow pointers, these can be implicitly stated inside each instruction. These pointers along with control flags can take care of data dependencies, whereas control flags alone can be used to enable each instruction for execution.

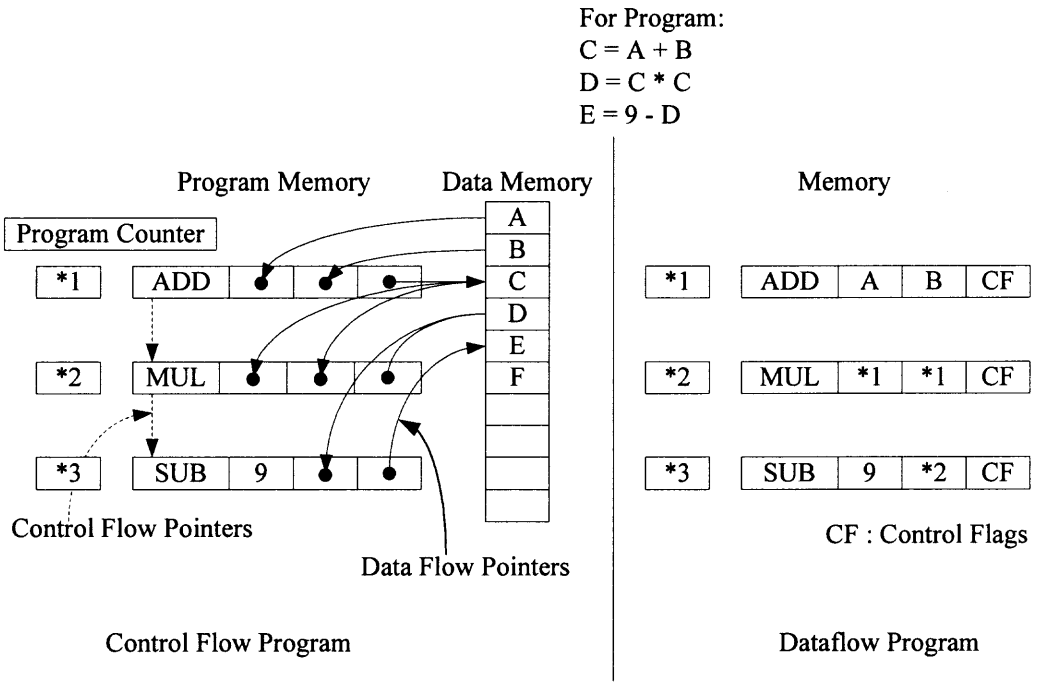


Figure 4.1 Comparison: Control Flow vs. Dataflow [2].

Basically, in dataflow machines, each instruction is considered as a process, either independent or dependent via either data or a clause to another instruction (process). Data can be passed from a parent instruction to a child instruction, either by having the parent instruction keep pointers to all child instructions or by having each child instruction keep pointers to the parent instruction. Each instruction is considered as a node and communicates with another node by a token, which is nothing but data transferred to another node with some ID field. Arcs connect these nodes to each other. Each simple node consists of input and output ports. Whenever a node has received all its operands it is fired to the execution node .A node is fired only when it is enabled. Enabling rules are different for different types of nodes. Strict enabling rules are followed for correct execution of programs. Figure 4.2 depicts the general flow for the dataflow graphs.

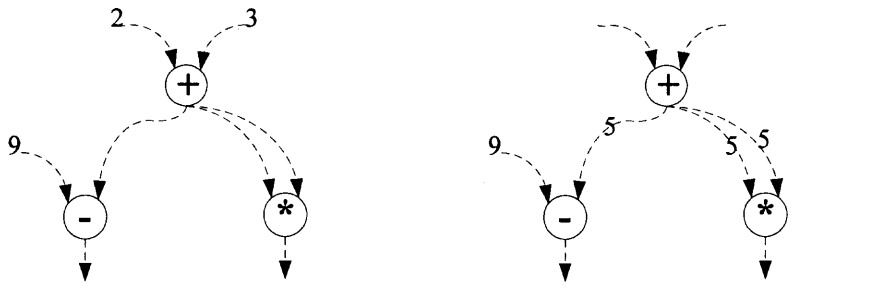


Figure 4.2 Nodes, Arcs and Firing of Nodes.

4.1.3 Types of Nodes

There are three basic types of nodes. These nodes support a regular instruction, conditional instruction or any loop instruction in the program. They are as follows:

1. Common nodes: The common nodes are shown in Figure 4.2. They represent common instructions in a program. They will fire if and only if both of their input values are available.

2. Switch Node: The switch nodes are shown in Figure 4.3. The value arriving at the input is placed either on true or false output arc depending upon the value of the control token. These are very useful in implementing conditional constructs.

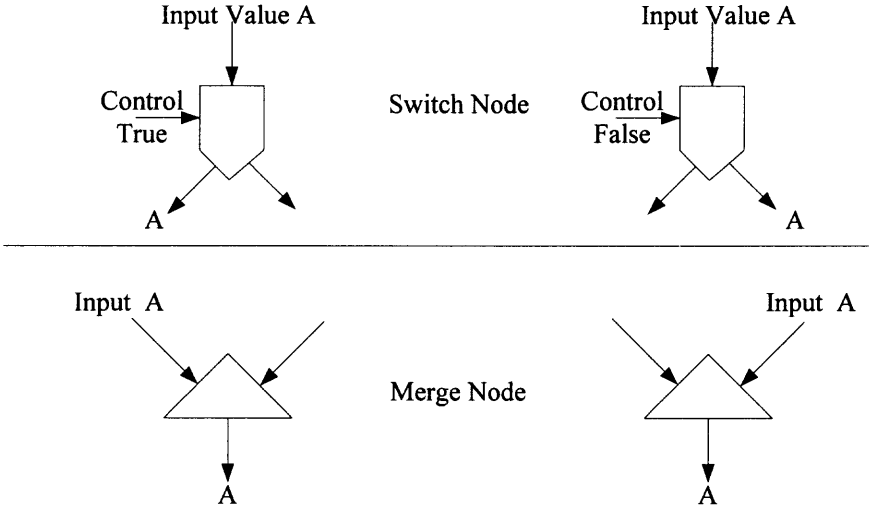


Figure 4.3 Switch and Merge Node.

3. Merge Node: The merge nodes are shown in Figure 4.3. When one of their input ports carries data, it fires and just copies input data to the output port.

Switch and Merge nodes are used to implement conditional and loop constructs. Unwise use of these two nodes in either of any constructs can lead to an erroneous result. That's why we change these two nodes to follow some strict enabling rules in the D^2-CPU design. The implementations of reentrancy and iterative constructs are discussed further.

4.1.3 Reentrancy

As described already, dataflow graphs are run away in nature. Simultaneous firing of many nodes can increase throughput but leads to instability if not handled properly. Reentrancy needs cyclic graphs, which in turn, may produce either a deadlock or a never finishing graph, if not implemented correctly. There are in general four ways [2] to handle iterative constructs in parallel machines, which are in fact classified under two styles of execution, dynamic and static.

- *Lock Method*: For any loop, the first time we need a value from outside the loop and for the remaining iterations, it derives it from the inside of it. If proper control is not kept, then the simultaneous execution of two iterations can lead to totally disastrous results. So, there has to be a mechanism to lock execution of iterations, so that the next iteration will start when the first one finished. The lock method is used to do that. Figure 4.4 shows the implementation of the lock method with use of merge and switch nodes. For the first time the values X is obtained from outside and the merge operator puts the value to its output when it becomes available; this value is checked for necessary condition by f(X). This enables the switch node, if the value is true it enters in the loop function "g", otherwise it comes out of the loop. This style preserves the correct execution, as the second iteration will only start, if the first one

is over. It is a safe and simple method but not at all attractive for any type of parallel machine, as the level of concurrency reduces with only one iteration executing at any given time. This is a the static style of implementation.

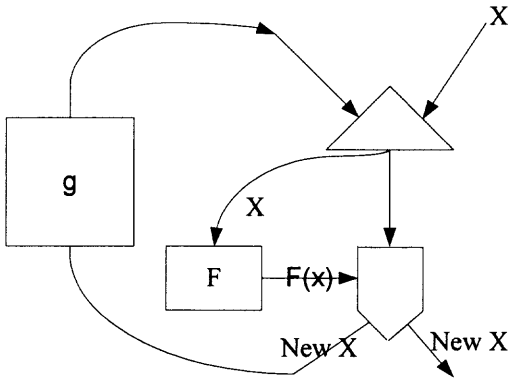


Figure 4.4 Lock Method for Reentrancy.

- *Acknowledge Method*: This method is implemented by introducing extra acknowledge arcs between two nodes. These acknowledge arcs work in a same fashion of merge and switch node, and take care of proper execution of loops, though this method is more complicated than the lock method. This is also a static loop implementation.
- *Code copying*: To derive a high level of concurrency from a reentrant graph, the best method is to allow iterations to execute as a separate instance of the graph. This can be done by using a code copying technique. This needs an intelligent compiler, as loop unfolding is done at the compiler level, where as a token is passed between two copies is preserved by the hardware. It is a real good method if a lot of concurrency occurs between iterations. This is a dynamic method.
- *Tagged-Token* [3]: First implemented in MIT's Tagged-Token-Dataflow-Architecture (TTDA), it is a really impressive method to exploit loop level parallelism. A tag, sometimes referred to as color or label, is attached to each node. A

tag represents a different iteration in a reentrant graph. So the firing rule is changed as a node is fired if and only if its input arcs contain data with the same tag. This is the most impressive dynamic method to implement loops. But this needs a lot of hardware support, with plenty of new logic blocks. It is the most complicated method.

Implementing reentrant graphs is again a tradeoff between performance and cost (required logic). In the next Chapter we will show how we can use the lock method and code-copying methods in the D^2-CPU design.

4.2 Programming with the D^2-CPU

4.2.1 Instruction Set

We already discussed the instruction set format in Chapter 4. Following are the instructions in the design implemented with their OPCODEs.

Each instruction below is modified by attaching a CAD (Clause Address), CAN (Clause Answer) and a CR (Clause Required), where if the CR of any instruction is set to '1' then that particular instruction will not be executed till its CAN becomes '1'; it is set to '0' at compile time. The CAN is provided by an instruction with ID CAD and has to be a boolean compare (SWITCH) instruction.

Instructions like MERGE, LOCK and STOP don't go to the ERU for execution. As we have implemented clause and loop support only in the DRAM, they even don't need to travel out of the particular DRAM. Each DRAM (main memory) controller takes care of such instructions.

Table 4.1 Instructions and OPCODEs

Sr. No.	Instruction	OPCODE	Description
1	ADD	000001	16 bit ADD instruction
2	SUB	000101	16 bit SUB instruction
3	MUL	000010	8 bit MUL instruction
4	AND	000011	16 bit AND instruction
5	OR	000111	16 bit OR instruction
6	NAND	001011	16 bit NAND instruction
7	NOR	001111	16 bit NOR instruction
8	XOR	010011	16 bit XOR instruction
9	NOT	010111	16 bit NOT instruction
10	EQT	011011	Equal To - Boolean Result
11	NEQT	011111	Not Equal To - Boolean Result
12	GT	100011	Greater Than - Boolean Result
13	LT	100111	Less Than - Boolean Result
14	GET	101011	Greater Than/Equal To - Boolean Result
15	LET	101111	Less Than/ Equal To - Boolean Result
16	SHL	110011	16 bit Shift Left
17	SHR	110111	16 bit Shift Right
18	RAL	111011	16 bit Rotate Left
19	RAR	111111	16 bit Rotate Right
20	MERGE	000000	Merge Node with LP 01
21	SWITCH	xxxx11	Any Compare Instruction with LP 10 If Included in Loop Else LP 00
22	LOCK	000000	Lock Node with LP 11
23	STOP	100000	Stop Instruction LP 11

As described earlier in this Chapter, MERGE (with LP 01) fires whenever its CAN is '1'; it receives any of its input tokens and just copies the required the input token to the output. It receives its CAN only once from outside the loop. A SWITCH instruction (with LP 10) fires when its CAN is '1' and it receives its operand from MERGE immediately above it. The answer to this SWITCH instruction is the CAN for the reaming normal instructions in a

loop, which set its CAN again to '0' when that instruction fires for each iteration. This same clause is for the LOCK instruction, a special instruction, which is a modified version of MERGE. This instruction fires if and only if its CAN is set to '1' and both of its operands are available. Its first operand is the output of a MERGE instruction whereas its second input is out put of the last instruction in the loop. So, this instruction preserves the correct execution by the lock method. This instruction just copies its second input to the output, which is the input to MERGE instruction for the next iteration, which fires when it receives this token. Figure 4.5 below shows modification of Figure 4.4 to adapt to the D^2-CPU design.

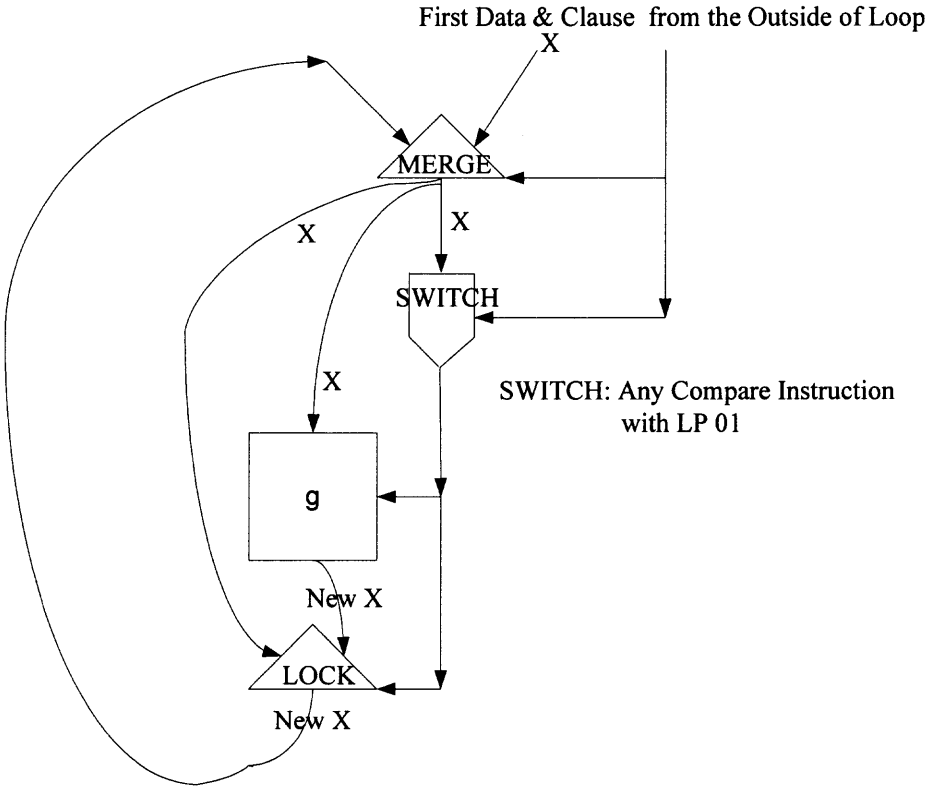


Figure 4.5 Lock Method Used in the D^2-CPU.

The code copying technique is very easy to implement in the D^2-CPU design but needs intelligent compiler support. There is no need to use MERGE and LOCK nodes, but we can directly use only SWITCH before each reentrant sub-graph 'g', which is alway

checked for a defined condition in each iterations. CAN, CAD, and CR suffice for this purpose.

The ORE (Operand Reuse) field in each common loop instruction is used as already described in Chapter 3; it takes care of data consistency for loops.

4.2.2 Sample Program

Table 4.2 below shows a sample program implemented in a high level language and also its corresponding assembly language conversion for conventional microprocessors. Table 4.3 gives the equivalent D^2-CPU code. Figure 4.6 shows a flow diagram for D^2-CPU code.

Table 4.2 Sample Program in High Level and Assembly Languages

High Level Language		Assembly Language	
Get (x)			Get (x)
y = (2 * x) – 10			MOV R1, x
If y > 0 then	Conditional		MUL R1, 2
b = 7			SUB R1, 10
for I = 1 to 10	Loop		MOV R2, 7
a = b * y			CMP R1, 0
y = a			JIL B1
end for			MOV R3, 1
else		LP1	CMP R3, 10
y = 7 * b			JIG EXIT
end if			MUL R1, R2
			ADD R3, 1
			JUMP LP1
		B1	MUL R1, R2
		EXIT	

Table 4.3 Equivalent D^2-CPU Code

Address	ORE	LP	VB	CR	CAN	CAD	OPFL	IID2	IID1	OPD2	OPD1	OPCODE	
A	Get (x)												
B	00	00	1	0	1	0	01	0	A	2	0	000010	MUL
C	00	00	1	0	1	0	01	0	B	10	0	000101	SUB
D	00	00	1	0	1	0	01	0	C	0	0	100011	GT
E	00	00	1	0	1	0	01	0	C	0	0	101111	LET
F	00	01	1	1	0	D	10	K	0	0	1	000000	MERGE
G	00	10	1	1	0	D	01	0	F	10	0	101111	LET
H	00	01	1	1	0	D	11	C	I	0	0	000000	MERGE
I	10	00	1	1	0	G	01	0	H	7	0	000010	MUL
J	11	11	1	1	0	G	11	I	F	0	0	000000	LOCK
K	10	00	1	1	0	G	01	0	J	1	0	000001	ADD
L	00	00	1	1	0	E	00	0	0	7	7	000010	MUL

Figure 4.6 and Table 4.3 show the exact similarity between the flow diagram and the assembly language for the D^2-CPU, which proves that the D^2-CPU follows pure data flow. In fact, a compiler can very easily support direct conversion from a flow diagram to assembly code. So the flow diagram can also be used by a graphical language for the D^2-CPU, which is one of the objectives specified in Chapter 2. As shown above, redundant instructions such as move are totally eliminated in the D^2-CPU design, which is a major achievement.

If we use the code copying technique for loop implementation, we can totally remove MERGE and LOCK instructions. We just need to copy the loop code and check the LET 10 condition periodically. This is the best method but need an intelligent compiler.

The next Chapter shows results of the D^2-CPU design implemented on an FPGA and also analysis of the design.

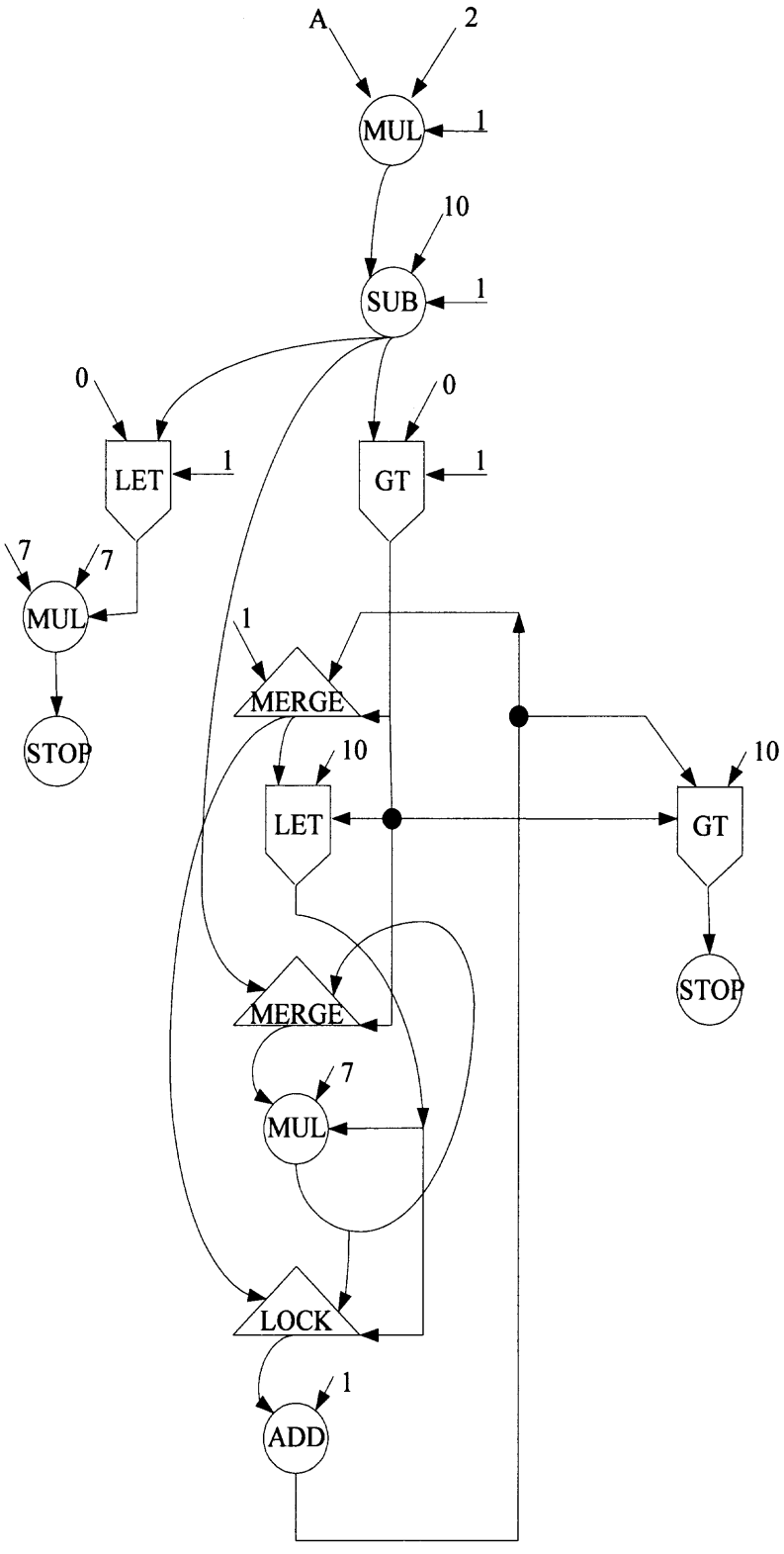


Figure 4.6 Flow Diagram for Code Presented in Table 4.3.

CHAPTER 5

RESULTS AND ANALYSIS OF THE D^2-CPU

5.1 Results

The host CPU communicates with the Annapolis board through APIs written in high level language C. Annapolis Microsystems provides standard functions in C, which we can use to communicate with the board. There are two standard functions WSII_WriteRegs_32 and WSII_ReadRegs_32, through which we can write to and read from the block RAM, respectively. For this purpose, two buffers are created pReadBuffer and pWriteBuffer. The pWriteBuffer can be assigned values, which in turn is used by the WSII_WriteRegs_32 function, whereas WSII_ReadRegs_32 writes its result to the pReadBuffer, which can be displayed. Table 5.1 shows pWriteBuffer and pReadBuffer for a very small program run on the D^2-CPU.

Each buffer is 32 bits wide, so numbers are specified in the hexadecimal format. As each block in the program memory consists of five 16-bit registers, three buffers represent a single block in the program memory. Two LOCK instructions are used to just a store the results of the first three instructions so that they can be verified.

Each Valid Bit (VB) is set to '0' when the instruction leaves for execution. This modifies each program memory, and in turn the block RAM. Also, the out put of the first LOCK instruction is just the output of the second instruction, as the LOCK instruction is a MERGE node just copying OPD2 to the output.

Bold numbers below show the changes occurring due to program execution and the OPD fields of LOCK instructions show the output results of the first, second, third, and fourth instructions, respectively.

Table 5.1 Results: Contents of Read and Write Buffer

Buffer Number	pWriteBuffer Contents (Hexadecimal Format)	Description	pReadBuffer Contents (Hexadecimal Format)	Description
1	00000000		00000000	For all three
2	A0010000		00010000	instructions
3	00010001	ADD 1, 1	00010001	Valid Bit is set to
4	00000000		00000000	0' after execution
5	A0020000		00020000	of instruction
6	00020002	MUL 2,2	00020002	
7	00000000		00000000	
8	A0330000		00330000	
9	00000002	SHL 2	00000002	
10	00000003		00000003	LOCK Instruction
11	A000C101		00000101	Stores Results of
12	00000000	LOCK *1, *2	00400002	1st & 2nd Instruction
13	00000003		00000003	LOCK Instruction
14	A000C203		00000203	Stores Results of
15	00000000	LOCK *3, *4	00400004	3rd & 4th Instruction

5.2 Analysis

Finally a comparative analysis between the D^2-CPU and a conventional processor is necessary to prove the viability of this project. A comparison is made below on two bases, first the hardware required and second the turnaround time. Non-pipelined units are assumed.

5.2.1 Storage Resources and Bus

If there are no software loops, then the D^2-CPU system stores only one copy of an instruction at any moment, whereas duplicate copies of instructions are stored in the cache and main

memory for conventional architecture. In the D^2-CPU, each instruction carries with it, the address fields for each operand. Except for the immediate addressing mode in a conventional CPU each instruction carries with it the respective register or memory addresses. Data is stored separately, which is the most common technique. So, both redundancies cancel each other. Each instruction carries its own physical address in the D^2-CPU. This increases the required width of the data bus but conventional CPU also has its address bus.

In the main memory, the D^2-CPU needs the CAD, CAN, and CR. These extra hardware recourses are needed in the D^2-CPU to support clauses. This extra hardware reduces the time penalty required to pay in conventional processors when any JUMP instruction occurs. In fact, such JUMP instructions are common features in conventional assembly languages, as they effectively implement very common conditional constructs in higher-level language programming (like If...else). With pipelining, such a penalty causes a huge difference, as the whole pipeline with all pre-fetched data has to be cleared till the JUMP instruction gets executed and then again starts from scratch to fill the pre-fetch data buffer. In the D^2-CPU, such a JUMP instruction doesn't exist, and even till the time the clause (in form of a token) reaches the required instructions to awake them, the remaining independent instructions still execute and no flushing of the pipeline or data buffer (EXT-CACHE or SRAM*) is needed.

The D^2-CPU requires PU-PIM units for each memory, but these units are external to the CPU and don't count towards chip area. These PU-PIM units make the memory part intelligent in the D^2-CPU.

5.2.2 Turnaround Time

A conventional CPU pays a lot of time penalty when either a page fault occurs or writing back results from cache to memory for data consistency. For the D^2-CPU, there is zero

probability of page fault as when an instruction leaves for the ERU, some other instruction in the main memory takes its place in the cache. Secondly, there are no writing back results from the cache to the main memory, as results or tokens are only propagated by the ERU and received by other units.

A single clock cycle is required to transfer the data from the cache memory to the CPU. For all memory addressing modes, a conventional CPU needs two or more cycles to fetch all of its required operands. Even for the regular fetching the CPU first has to put address on the address bus and then in the second clock cycle it gets data from the cache so two cycles are needed to fetch an instruction. For writing back results to memory again two clock cycles are needed, whereas writing back results from the cache to the main memory are considered in the above paragraph.

For the D^2-CPU, instructions are supplied from the outside so there are no addressing modes to count towards any time increase. As each instruction requires one clock cycle to move from the cache to the ERU, the D^2-CPU saves a clock cycle behind every cache to ERU transfer. Tokens released irrespective of ERU execution cycles need only one clock cycle to transfer from the ERU to cache, and again the D^2-CPU saves one clock cycle.

This saving in time is compensated by the logic required to implement intelligent memories. Each block in the memory is associated with logic units, which basically constitutes comparators. It is not a big penalty in terms of hardware resources out side of the ERU chip. With the advent of the PIM concept, this is a practically possible design.

5.2.3 Software Support

As shown in Chapter 4, the D^2-CPU uses a graphical language which differs very slightly from actual assembly language. Because of this, the D^2-CPU needs very little help from the compiler. The compilers needed for this design are not required to be very complicated,

which in turn saves time in compilation and also the cost for making them. So, this is another benefit of the D^2-CPU design. These graphical languages are very easy to use compared to current high level languages. This creates more user friendly language constructs than present languages.

This concludes the result and analysis part of the D^2-CPU design, though a lot of work is still needed in terms of testing programs on this D^2-CPU design. As each design change needs 15 to 16 hours for the present logic complexity to complete a whole design cycle, a lot of time is needed to do such an analysis. Although this time is nothing, compared to ASIC designs, as the latter need similar times to just prove the design at the post-layout simulation level, whereas the actual manufacturing of the chip requires several months and lots of money.

CHAPTER 6

CONCLUSIONS

This thesis work was able to implement the D^2-CPU design [1]. Successful implementation of the D^2-CPU design on a Xilinx Virtex-II FPGA mounted on the Annapolis Microsystems Wildstar-II board proves the viability of the data-driven paradigm with the FPGAs at the single processor level. This work is able to fulfill the required objectives like:

- A radical single processor design supporting the pure data-driven paradigm.
- A design with distributed control and minimized redundant operations.
- High utilization of resources in directly application related work, i.e. towards more productivity.
- Low hardware complexity, which leads to low cost and low power consumption.

Though it is a quite successful design implementation the following improvements are needed:

- Instruction relocation proposed in [1] needs to be implemented for multiprogramming and virtual memory support.
- Exception handling is needed for a fault tolerant architecture.
- Functional units need to be pipelined for increase of the system frequency.
- ERU-SRAM units need to be smarter, by adding a count field, which increases more dynamic parallelism in the ERU.
- The size of the EXT-CACHE and DRAM has to be increased for more instruction support.
- Implementation of full-duplex bidirectional buses throughout will increase the throughput of the D^2-CPU design.
- Extensive testing is needed for a fault tolerant architecture and also to show the exact speedup over the convectional CPU in different styles of programming.

APPENDIX

DESIGN REPORT FILES

Report files, generated by different design tools are listed below. These files give sight of actual design on hardware level.

Synthesis Report File: Below is the content of log file generated by Synplify-Pro synthesis tool, which gives detail timing report and resource utilization.

$ Start of Compile

#Tue Nov 11 18:52:55 2003
Synplicity VHDL Compiler, version 7.1, Build 158R, built Apr 18 2002
Copyright (C) 1994-2002, Synplicity Inc. All Rights Reserved
VHDL syntax check successful!
Synthesizing wsii_pe_lib.system1.struc
@N:"C:\anish\Dataflow\new\system.vhd":64:0:64:1|Instance c1 is bound to entity memory_system, architecture struc.
@N:"C:\anish\Dataflow\new\system.vhd":65:0:65:1|Instance c2 is bound to entity ERU, architecture eru.
Synthesizing wsii_pe_lib.eru.eru
@N:"C:\anish\Dataflow\new\eru.vhd":53:0:53:1|Instance c1 is bound to entity cpu, architecture rtl.
@N:"C:\anish\Dataflow\new\eru.vhd":54:0:54:1|Instance c2 is bound to entity hm, architecture hm.
Synthesizing wsii_pe_lib.hm.hm
Post processing for wsii_pe_lib.hm.hm

START TIMING REPORT
Timing Report written on Tue Nov 11 21:42:33 2003
#Top view: system1
Slew propagation mode: worst
Paths requested: 5
Constraint File(s):
@N| This timing report estimates place and route data. Please look at the place and route timing report for final timing.
@N| Clock constraints cover all FF-to-FF, FF-to-output, input-to-FF and input-to-output paths associated with a particular clock.

Performance Summary
****
Worst slack in design: -16.730

Starting Clock	Requested Frequency	Estimated Frequency	Requested Period	Estimated Period	Slack	Clock Type
I_CLK	66.0 MHz	200.7 MHz	15.152	4.982	10.169	inferred
c1.c3.token1_inferred_clock[5]	66.0 MHz	76.0 MHz	15.152	13.152	2.000	inferred
clk	66.0 MHz	**31.4 MHz**	15.152	31.881	-16.730	inferred
System	66.0 MHz	34.7 MHz	15.152	28.782	-13.630	system

==========

END TIMING REPORT

Resource Usage Report for system1

Mapping to part: xc2v6000ff1517-4
Cell usage:
VCC 9 uses
MUXF5 2627 uses
FDC 2823 uses
FDCE 2713 uses
GND 9 uses
MUXCY_L 297 uses
XORCY 249 uses
MUXF6 240 uses
FDPE 7 uses
MUXCY 7 uses
FDE 247 uses
MULT18X18 1 use
RAMB16_S36_S36 1 use
FDP 5 uses
LDC_1 1952 uses
LDC 1632 uses
LDCE_1 448 uses
LDCP_1 32 uses
LD 8320 uses
LD_1 7808 uses
LDP_1 128 uses

I/O primitives:
OBUF_F_24 34 uses
IBUF 45 uses

BUFG 7 uses

BUFGP 1 use
I/O Register bits: 2
Register bits not including I/Os: 5793 (8%)

Internal tri-state buffer usage summary
BUFTs + BUFEs: 10068 of 16896 (59%)

RAM/ROM usage summary
Block Rams : 1 of 144 (0%)

Global Clock Buffers: 8 of 8 (100%)

Mapping Summary:
Total LUTs: 59155 (87%)

Mapper successful!
Process took 9854.3 seconds realtime, 9854.31 seconds cputime
---

Mapping Report File: This file gives design summary after mapping the design to the required technology, here Xilinx Virtex-II.

Release 5.2.03i - Map F.31
Xilinx Mapping Report File for Design 'system1'

Design Information

Command Line : C:/Xilinx/bin/nt/map.exe -quiet -p xc2v6000-ff1517-4 -cm area -detail -pr b -u -k 4 -c 100 -tx off -o system1_map.ncd system1.ngd system1.pcf
Target Device : x2v6000
Target Package : ff1517
Target Speed : -4
Mapper Version : virtex2 -- $Revision: 1.4 $
Mapped Date : Tue Nov 11 21:57:47 2003

Design Summary

Number of errors: 0
Number of warnings: 1723
Logic Utilization:
Total Number Slice Registers: 26,113 out of 67,584 38%
Number used as Flip Flops: 5,793
Number used as Latches: 20,320
Number of 4 input LUTs: 58,823 out of 67,584 87%
Logic Distribution:
Number of occupied Slices: 33,790 out of 33,792 99%
Number of Slices containing only related logic: 32,792 out of 33,790 97%
Number of Slices containing unrelated logic: 998 out of 33,790 2%
*See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs: 58,946 out of 67,584 87%
Number used as logic: 58,823
Number used as a route-thru: 123

Number of bonded IOBs: 80 out of 1,104 7%
IOB Flip Flops: 2
Number of Tbufs: 10,068 out of 16,896 59%
Number of Block RAMs: 1 out of 144 1%
Number of MULT18X18s: 1 out of 144 1%
Number of GCLKs: 8 out of 16 50%

Total equivalent gate count for design: 610,922
Additional JTAG gate count for IOBs: 3,840
Peak Memory Usage: 541 MB

NOTES:

Related logic is defined as being logic that shares connectivity - e.g. two LUTs are "related" if they share common inputs. When assembling slices, Map gives priority to combine logic that is related. Doing so results in the best timing performance.

Unrelated logic shares no connectivity. Map will only begin packing unrelated logic into a slice once 99% of the slices are occupied through related logic packing.

Note that once logic distribution reaches the 99% level through related logic packing, this does not mean the device is completely utilized. Unrelated logic packing will then begin, continuing until all usable LUTs and FFs are occupied. Depending on your timing budget, increased levels of unrelated logic packing may adversely affect the overall timing performance of your design.

Place and route report file: This file gives place and route details. As shown below maximum pin delay is 28.9 ns , so in fact we can use 38.1 MHz clock frequency in place of 31.4 MHz indicated by synthesis tool.

```
Release 5.2.03i - Par F.31
Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.

XEON-2::  Tue Nov 11 22:00:11 2003

C:/Xilinx/bin/nt/par.exe -w -ol 3 -t 1 -ub -detail system1_map.ncd system1.ncd
system1.pcf
Constraints file: system1.pcf
Loading device database for application par from file "system1_map.ncd".
   "system1" is an NCD, version 2.37, device xc2v6000, package ff1517, speed -4
Loading device for application par from file '2v6000.nph' in environment
C:/Xilinx.
The STEPPING level for this design is 0.
Device speed data version:  PRODUCTION 1.114 2002-12-13.

Device utilization summary:

   Number of External IOBs          80 out of 1104    7%
   Number of LOCed External IOBs    0 out of 80       0%

   Number of MULT18X18s             1 out of 144      1%
   Number of RAMB16s                1 out of 144      1%
   Number of SLICEs             33790 out of 33792   99%

   Number of BUFGMUXs               8 out of 16       50%
   Number of TBUFs              10068 out of 16896    59%

Overall effort level (-ol):  3 (set by user)
Placer effort level (-pl):   3 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):   3 (set by user)

Phase 1.1
Phase 1.1 (Checksum:a377af) REAL time: 18 mins 29 secs

Phase 3.23
.....
Phase 3.23 (Checksum:9896bb) REAL time: 22 mins 31 secs

Phase 4.3
Phase 4.3 (Checksum:26259fc) REAL time: 24 mins 36 secs

Phase 6.5
```

Phase 6.5 (Checksum:39386fa) REAL time: 24 mins 58 secs

Phase 7.8
.....
Phase 7.8 (Checksum:82cb3cb) REAL time: 2 hrs 33 mins 38 secs

Phase 8.5
Phase 8.5 (Checksum:4c4b3f8) REAL time: 2 hrs 34 mins 3 secs

Phase 9.18
Phase 9.18 (Checksum:55d4a77) REAL time: 2 hrs 44 mins 53 secs

Phase 10.19
Phase 10.19 (Checksum:5f5e0f6) REAL time: 2 hrs 48 mins 53 secs

Phase 11.24
Phase 11.24 (Checksum:68e7775) REAL time: 2 hrs 48 mins 53 secs

Writing design to file system1.ncd.

Total REAL time to placer completion: 2 hrs 49 mins 2 secs
Total CPU time to placer completion: 2 hrs 42 mins 45 secs

Starting Router REAL time: 2 hrs 49 mins 20 secs

Phase 1: 287639 unrouted; REAL time: 2 hrs 49 mins 45 secs

Phase 2: 265447 unrouted; REAL time: 2 hrs 57 mins 43 secs

Phase 3: 125529 unrouted; (1777) REAL time: 3 hrs 4 mins 41 secs

Phase 4: 125706 unrouted; (0) REAL time: 3 hrs 42 mins 11 secs

Intermediate status: 15397 unrouted; REAL time: 4 hrs 13 mins 33 secs

Intermediate status: 3998 unrouted; REAL time: 4 hrs 51 mins 4 secs

Intermediate status: 1224 unrouted; REAL time: 5 hrs 27 mins 30 secs

Intermediate status: 385 unrouted; REAL time: 6 hrs 3 mins 46 secs

Intermediate status: 94 unrouted; REAL time: 6 hrs 37 mins 35 secs

Intermediate status: 17 unrouted; REAL time: 7 hrs 8 mins 25 secs

Intermediate status: 8 unrouted; REAL time: 7 hrs 40 mins 32 secs

Phase 5: 0 unrouted; (0) REAL time: 7 hrs 59 mins 39 secs

Finished Router REAL time: 7 hrs 59 mins 40 secs

Total REAL time to router completion: 8 hrs 34 secs
Total CPU time to router completion: 7 hrs 52 mins 20 secs

Generating "par" statistics.

****
Generating Clock Report
****
It's a huge report and not included here.
The Delay Summary Report

The Score for this design is: 689

The Number of signals not completely routed for this design is: 0

The Average Connection Delay for this design is: 2.844 ns
The Maximum Pin Delay is: 25.879 ns
The Average Connection Delay on the 10 Worst Nets is: 20.229 ns

Listing Pin Delays by value: (ns)

d < 5.00	< d < 10.00	< d < 15.00	< d < 20.00	< d < 26.00	d >= 26.00
221804	41113	13938	1675	41	0

All signals are completely routed.

Total REAL time to par completion: 8 hrs 37 mins 21 secs
Total CPU time to par completion: 8 hrs 27 mins 55 secs

Placement: Completed - No errors found.
Routing: Completed - No errors found.

Writing design to file system1.ncd.

PAR done.

REFERENCES

[1] S. Ziavras, "Processor Design Based on Dataflow Concurrency," Microprocessors and Microsystems, Vol. 27, No. 4, May 2003, pp. 199-220.

[2] A. Veen, "Dataflow Machine Architecture", ACM Computing Surveys, Vol. 18, No. 4, Dec. 1986.

[3] Arvind and R. Nikhil, "Executing a Program on the MIT Tagged-Token-Dataflow-Architecture", IEEE Trans. Comput., Vol. 39, no. 3, Mar. 1990, pp.300-318.

[4] D. Lopez, J. Llosa, M. Valero, and E. Ayguade, "Widening Resources: A Cost-Effective Technique for Aggressive ILP Architectures," MICRO '98, pp. 237-246.

[5] X. Tang and G. Gao, "Automatically Partitioning Threads for Multithreaded Architectures," Journ. Paral. Distr. Comput., Vol. 58, 1999, pp. 159-189.

[6] H. Hum et al., "A Design Study of the Earth Multiprocessor," International Conf. Paral. Arch. Compil. Techn., 1995, pp. 59-**68.**

[7] R. Korry, C. McCann, and B. Smith, "Memory Management in the Tera MTA System," Techn. Rep., Tera Comput., Seattle, WA, 1995.

[8] M. Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Jones and Bartlett Publ., 1995.

[9] H. Terada et al., "Design Philosophy of a Data-Driven Processor: Q-p," Journ. Inform. Proc. Vol. 10, No. 4, Mar. 1988, pp. 245-251.

[10] M. Chatterjee, S. Banerjee, and D. Pradhan, "Buffer Assignment Algorithms on Data Driven ASICs," IEEE Trans. Comput., Vol.49, No.1, Jan. 2000, pp. 16-32.

[11] H. Kung and M. Lam, "Wafer Scale Integration and Two Level Pipelined Implementation of Systolic Arrays," Jurn. Paral. Distr. Comput., Vol. 1, No. 1, Sept. 1984, pp.32-63.

[12] S. Ingersoll and S. Ziavras, "Intelligent Memories for Dataflow Computation and Emulation on Field Programmable Gate Arrays," Microprocessors and Microsystems, Vol. 26, No. 6, Aug. 2002, pp. 263-280.

[13] B. Radanovich, "An Overview of Advances in Reconfigurable Computing Systems," Proceedings, Conference on System Sciences, 1999.

[14] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," IEEE Proc. Int. Conf. Exhib. Design Automation, Testing Europe, Munich, Germany, 2001, pp.135-143.

[15] S. Hauck, G. Borriello, C. Ebeling, "Mesh Routing Topologies for Multi- FPGA Systems," International Conference on Computer Design, pp.170-177, 1994.

[16] X. Wang and S. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," Concurrency and Computation , 2003.

[17] Widlstar-II Hardware Reference manual, Annapolis Microsystems, revision 2.4, 2002.

[18] http://toolbox.xilinx.com/docsan/xilinx4/data/docs/lib/dsgnelpr32.html (taken on 21st Nov. 2003)

[19] http://www.intel.com/cd/ids/developer/asmo-na/eng/technologies/threading/applying (taken on 21st Nov. 2003)