ABSTRACT

EMULATION OF THE DATAFLOW COMPUTING PARADIGM USING FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

by
Segreen Ingersoll

Building a perfect dataflow computer has been an endeavor of many computer engineers. Ideally, it is a perfect parallel machine with zero overheads, but implementing one has been anything but perfect. While the sequential nature of control flow machines makes them relatively easy to implement, dataflow machines have to address a number of issues that are easily solved in the realm of control flow paradigm. Past implementations of dataflow computers have addressed these issues, such as *conditional* and *reentrant* program structures, along with the *flow of data,* at the processor level, i.e. each processor in the design would handle these issues. The design presented in this thesis solves these issues at the memory level (by using intelligent-memory), separating the processor from dataflow tasks. Specifically, a two-level memory design, along with a pool of processors was prototyped on a group of Altera FPGAs.

The first level of memory is an intelligent-memory called Dataflow Memory (DFM), carrying out dataflow tasks. The second level of memory called the Instruction Queue (IQ) is a buffer that queues instructions ready for execution, sent by the DFM. The second level memory has a multiple bank architecture that allows multiple processors from the processor pool to simultaneously execute instructions retrieved from the banks. After executing an instruction, each processor sends the result back to the dataflow memory, where they fire new instructions and send them to the IQ.

This thesis shows that implementing dataflow computers at the intelligent-memory level is a viable alternative to implementing them at the processor level.

EMULATION OF THE DATAFLOW COMPUTING PARADIGM USING FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

by
Segreen Ingersoll

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 Computer Engineering

Department of Electrical and Computer Engineering

January 2001

APPROVAL PAGE

EMULATION OF THE DATAFLOW COMPUTING PARADIGM USING FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

Segreen Ingersoll

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

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

Dr. Durga Misra, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Segreen Ingersoll

Degree: Masters in Computer Engineering

Date: January 2001

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2001
- Bachelor of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1998
- Bachelor of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1998
- Bachelor of Science in Physics,
  Bombay University, Bombay, India, 1992

Major: Computer Engineering

Presentations and Publications:

Segreen Ingersoll and Sotirios Ziavras,
"Dataflow Computation With Intelligent Memories Emulated on Field-Programmable Gate Arrays (FPGAs)." VLSI Design Journal, submitted for publication, December 2000.

To my beloved parents

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to Dr. Sotirios Ziavras, who not only served as my thesis advisor, providing valuable resources and insight, but also for supporting and encouraging my ideas. Special thanks are given to Dr. Edwin Hou and Dr. Durga Misra for actively participating in the committee.

I would like to take this opportunity to also express my gratitude to Dr. Edip Niver, who supported my work as a graduate student, providing me with an assistantship, and Dr. Raashid Malik who first introduced me to dataflow computing.

Finally I would like to mention a note of thanks to all the staff at ECE department for their assistance over the years.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

1 INTRODUCTION

1.1 Objective

Computing or information systems today are the most complex human artifacts, in many respects, from their material composition. The invention of the transistor revolutionized the computing industry and has had a tremendous impact on the society in a manner that no one could have foreseen or predicted. Current estimates of transistor production stands at about 5 ~ 6 billion transistors every second [21]. As manufacturing processes get smaller and cheaper, add to that ever-faster computers, and simulation tools that can replicate an engineer's design to the smallest detail, allows innovators to stretch their imagination and ingenuity without any bounds. Every few years, what seemed improbable is accomplished and exceeded. The dataflow computer is one such entity, every new generation of design engineers has been coming closer to realizing a pure dataflow machine. Dataflow computing was first proposed in the 1950s, followed by dataflow schemas proposed by Karp and Miller in 1966 [1] and Rodriquez in1969 [1]. This document presents yet a new way of implementing a dataflow computer using FPGAs (Field Programmable Gate Arrays).

1.2 Fundamentals of Computing Paradigms

Computing architectures can be broadly classified into two major groups based on their execution model, namely *Control Flow* and *Data Flow*. This section briefly presents the properties of the two paradigms.

1.2.1 Control Flow

By far, the Control Flow (CF) paradigm is the oldest surviving computing paradigm, which also happens to be the most popular architecture being implemented today. The CF model at its core is a sequential machine proposed by Von Neumann. It is surprising to observe that the basic Von Neumann sequential computation model has remained essentially the same through more than four decades. It is worth noting here that sequential machines have been shown to be universal computing machines by Alan Turing, and that the Von Neumann model can be considered as a pragmatic embodiment of the Turing machine [7].

The heart of a CF machine is the Program Counter (PC), which *steps* sequentially through a program (machine instructions) in memory, until it is explicitly changed by a instruction in memory such as a conditional instruction, or a jump instruction. Each *step* is divided into *sub-steps*; during the first *sub-step,* the memory address in the PC is used to fetch an instruction from memory while simultaneously incrementing the PC (PC = PC + 1). This is followed by a *sub-step* where the fetched instruction is decoded. In the subsequent *sub-step,* the operands specified by the instructions are fetched. This is followed by a *sub-step* where the instruction is sent to the Arithmetic Logic Unit (ALU) or execution unit, in general, to be executed. In the final *sub-step,* the result obtained earlier from the ALU is written back to memory or CPU registers. The whole process is then repeated and continues till a conditional instruction or a jump sends the PC in a new direction of execution. Other reasons the PC would change direction of execution, is when a procedure is called or when returning from a procedure, and when an exception occurs.

Within the framework of the von Neumann model, relatively few architectural innovations characterize today's sequential computers. These can be summarized as follows:

- The *indexed modification of addresses* and the *memory hierarchy* ideas was conceived by a group at Manchester University in 1949. The index registers permitted the execution of loops without modifying the instruction addresses and the automatic reallocation of programs in memory. The memory hierarchy idea led later to the *caches* and *virtual machine* concepts [7].
- In 1951, Wilkes proposed the *micro-programmed control* technique, as a new and systematic way of controlling the operation of computers [7].
- Another important innovation incorporated into the von Neumann model is the *stack architecture* proposed by Barton in 1958, as a tool for compiling and executing expressions, in order to have the machine architecture reflect the organization of a specific programming language. The same concept has been subsequently recognized as particularly advantageous for operating systems in managing subroutine invocation and in general program context [7].
- Other major innovations thrown into the cauldron of von Neumann model to particularly "improve performance" are the introduction of the *pipeline architecture, vector arithmetic units, super-scalar structures* and, *separation of I/O from major processor tasks* [7].

One reason for the whole hearted adoption of the Von Neumann architecture can be traced to its conceptual simplicity, which was necessary at the time it was conceived due to the high cost (and unreliability) of the original electronic components (vacuum

tubes). A more poignant reason for its success is the fact that it has operated as an efficient bridge between software and hardware, permitting the hardware to be developed almost independently from the software and vice-versa.

1.2.2 Data Flow

The concept of data-driven computation is as old as electronic computing. It is ironic that the same Von Neumann, who is some-times blamed for having created a bottleneck that dataflow architecture tries to remove, made an extensive study of neural nets, which have a data-driven nature [1]. Asynchronously operating in/out channels, introduced in the 1950s, which communicate according to a ready/acknowledge protocol, are among the first implementations of data-driven execution. The development in the 1960s of multiprogrammed operating systems provided the first experience with the complexities of large-scale asynchronous parallelism. After exposure to these problems in the MULTICS project, in 1969 Dennis developed the model of dataflow schemas, building on work by Karp and Miller (1966) and Rodriquez (1969) [1]. These dataflow graphs, as they were later called, evolved rapidly from a method for designing and verifying operating systems to a base language for a new architecture. The first designs for such machines (Dennis and Misunas 1974; Rumbaugh 1975 [1]) were made at Massachusetts Institute of Technology. The first dataflow machine became operational in July 1976 [1].

Dataflow and Control flow are two extremes of execution model spectrum. While control flow is inherently sequential, dataflow is inherently parallel. That is to say, to extract the parallelism out of a control flow machine, additional work has to be done to find instructions that can be executed concurrently. On the other hand, a dataflow machine's inherent parallelism eliminates this overhead.

In a dataflow computer, the execution of an instruction is driven by data availability instead of being guided by a program counter (described in the previous section). In theory, any instruction should be ready for execution whenever all operands needed for it's execution become available. Unlike sequential computers, where the instructions resident in memory need to be stored in an ordered manner (since the program counter steps through an ordered set of instructions), instructions in a data-driven program can be ordered randomly in memory. Though data and instructions reside in the same memory in a sequential von Neumann machine, they are stored in separate memory locations, i.e. an instruction and the data it operates on, does not reside in the same memory location. On the other hand, in a dataflow machine data are held inside the instructions [20].

Due to the inherent parallelism of dataflow machines, it is not possible to describe their operation in neat discrete steps as was possible in the operation of the Von Neumann computer. Instead, an overview of what happens inside a dataflow machine is presented. Results obtained from the execution of an instruction are called *data tokens,* and are passed directly between instructions [20], i.e. when a instruction is executed the produced result is duplicated into many copies, and forwarded directly to all instructions that need that value. Each of these instructions that receive the value check to see if they possess all the operands they need to execute. If any such instructions exist, they *fire* (they are dispatched to be executed), producing more data tokens upon execution, which in turn may fire other instructions. This process continues till no more data tokens are produced, and no more instructions are readied for execution. A data token once consumed by an instruction, is no longer available for reuse by other instructions [20].

This data-driven scheme requires no shared memory (since results are passed directly to needy instructions), no program counter, and no control sequencer (since no order of execution needs to be specified at run time) [20]. However, it requires special mechanisms to detect data availability, to match tokens with needy instructions, and to enable the chain reaction of asynchronous instruction executions. No memory sharing results in no *side effects*, which is a problem in von Neumann computers.

Asynchrony implies the need for handshaking or token-matching operations [20]. These two operations produce considerable overhead on a dataflow computer; a reason why dataflow machines have not made it into mainstream computing. A pure dataflow computer exploits fine-grain parallelism at the instruction level. Massive parallelism would be possible if data-driven mechanism could be cost-effectively implemented with low instruction execution overhead [20].

To get a better overview of the two paradigms covered, an example is presented of how a program is executed within the realm of either paradigms. Consider the execution of the following program:

Addr	**Instruction**
1.	`C = A + B;`
2.	`D = C * A;`
3.	`E = C - B;`
4.	`F = C + D;`

Figure 1.1 Sample Program

It can be easily seen from Table 1.1, that the dataflow model needs one less step to execute this program and that it extracts the maximum parallelism out of this set of instructions. Control flow on the other hand does not extract any parallelism and would

require special software (compiler or OS) or hardware added on to the sequential model (look-ahead or branch prediction) to accomplish the same speedup.

Table 1.1 Execution of a set of instructions under the two execution paradigms; Control Flow and Data Flow

STEPS	CONTROL FLOW	DATA FLOW
I	When PC is 1, operands A and B are added and the result is stored in C. PC is incremented.	When operands A and B become available, instruction 1 is sent for execution. The resulting sum C is sent to instructions 2, 3, and 4.
II	Operands C and A are multiplied and the result is stored in D. PC is incremented.	Both instructions 2 and 3 have all their operands available (C from instruction 1, A & B from before instruction 1 was executed). Both instructions 2 & 3 are sent for execution and result D is sent to instruction 4, and E to any other instructions that need it.
III	Operand B is subtracted from operand C and the result is stored in E. PC is incremented.	Instruction 4 has all its operands available (C from 1 and D from 2), so it is dispatched for execution and the result F is forwarded to any needy instructions.
IV	Operands C and D are added and the result is stored in F. PC is incremented.	

A program memory layout is presented below to show how the above set of instructions is laid out in memory of a control flow and data flow computer.

```
1.   C = A + B;
2.   D = C * A;
3.   E = C - B;
4.   F = C + D;
```

Figure 1-2 (a)

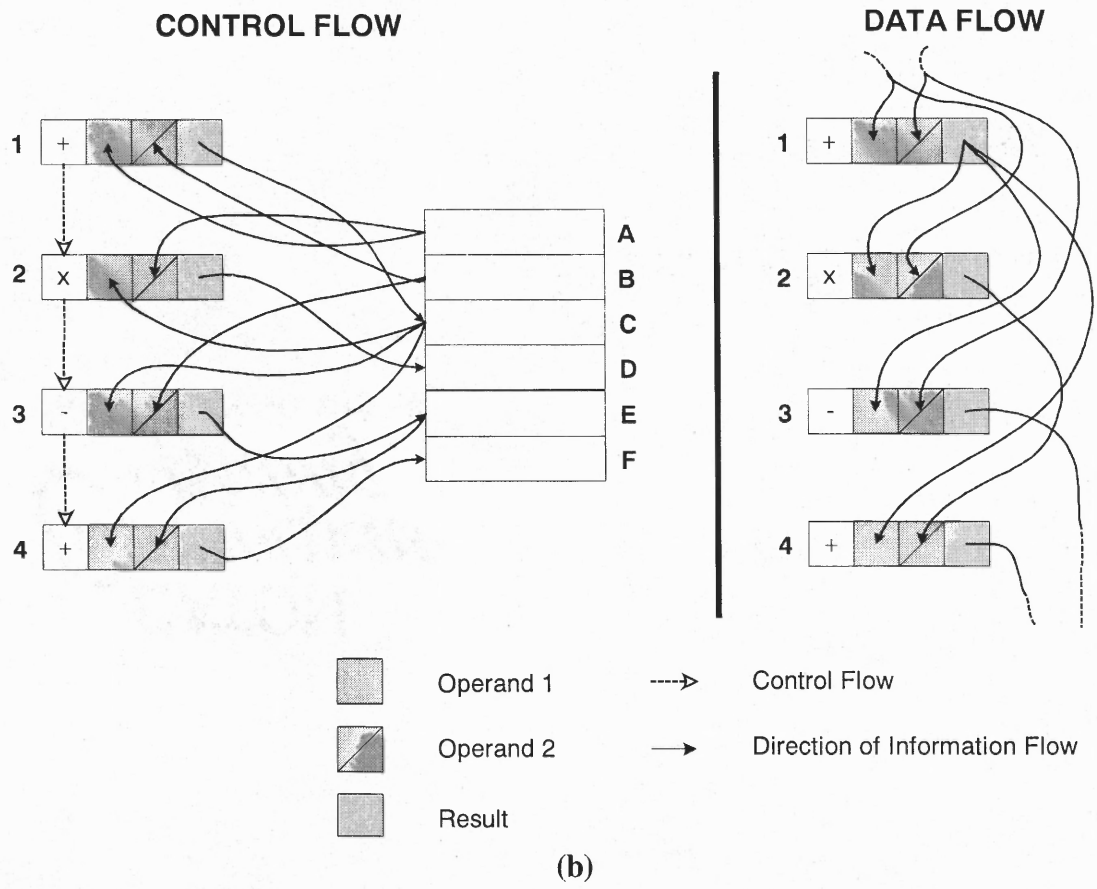


Figure 1.2 (a) Sample Program (b) Memory layout of instructions in Control Flow and Data Flow Computers.

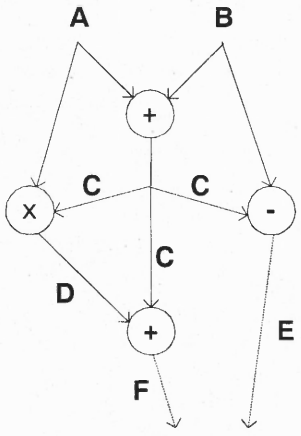
The left side of the figure represents the program in memory of a control flow computer and the right side represents the program in memory of a data flow computer. Though the memory locations in the control flow computer, labeled 1,2,3,4 and A,B,C,D,E,F are shown separately, they belong to the same memory space. They are drawn side-by-side to accommodate the representation of the flow of operands and results. Note that there are no control flow arrows in the dataflow computer, since there aren't any. Also note that, while results are stored in designated locations in the memory of the control flow computer, in a dataflow computer the results are stored in the same location as the instructions receiving the results.

Now that the two paradigms have been adequately understood, it is only appropriate that issues that are to be dealt with when designing and implementing dataflow computers are discussed.

2 ISSUES AND PRIOR RESEARCH

2.1 Issues (Data Flow in Depth)

The dataflow-computing paradigm is a direct result of dataflow graphs, which are a characteristic component of graph theory. Shown in the figure below is the dataflow graph of the program that was presented earlier.




```
1.    C = A + B;
2.    D = C * A;
3.    E = C - B;
4.    F = C + D;
```

Figure 2.1 Dataflow graph of the program presented on the right

Each dataflow graph is represented using directed arcs and nodes; arcs are arrows entering or leaving the nodes, which are points that hold the instruction to be executed. The number of arcs entering a node equals the number of operands the node (instruction) needs to fire (execute), and the number of arcs leaving a node equals the number of nodes (instructions) that need the result of the firing node (instruction execution).

This simplistic execution model of dataflow computers is also its biggest drawback. The lack of an inherent control mechanism as in the von Neumann architecture requires other mechanisms to keep a dataflow computer in check.

The first problem to be addressed is that of "How does an instruction know that its operands have arrived?" Since instructions are static objects without intelligence, some mechanism has to be provided to accomplish this task. This is normally done by a device called the *token matching unit,* as shown in the Figure 2.2. The figure below shows a typical construct of a dataflow computer called the *Processing Element (PE),* which handles a number of nodes stored in the box labeled *memory for nodes* [1].

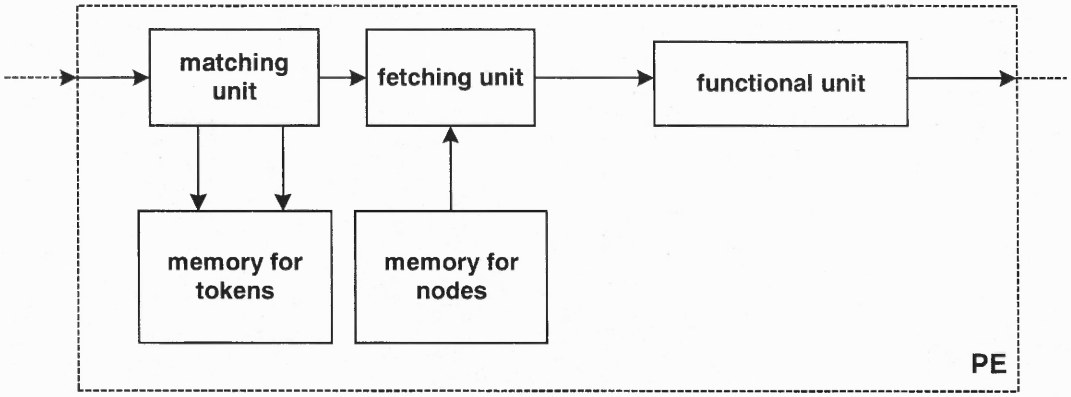


Figure 2.2 Functional diagram of a processing element in of a tagged-token machine [1].

Input arcs (data tokens) enter the PE from the left and output arcs (result) exit from the right of the PE. Each token entering the PE has two fields, 'a list of destination nodes' and 'a value' that has to be forwarded to the list of destination nodes. In most dataflow machines, the number of input arcs is limited to two, and associated with each item in the 'list of destination nodes' in the token is an extra bit that indicates whether the addressed node is monadic or dyadic. Only for a dyadic node does the matching unit check whether its local memory already contains a matching token, i.e. it looks for a token with the same destination. When the matching unit finds a node ready to fire, the fetching unit extracts the addressed node (instruction) from the 'memory for nodes' and

forwards the entire set (instruction, operand/s, and destination addresses) to the functional unit, which in turn executes the instruction and sends out the result (a token containing the result and destination addresses) [1].

Other issues that plague dataflow machines are related to the implementation of simple programming constructs that are taken for granted in Von Neumann architecture, such as conditional statements, loops (count and conditional) and modularization (procedures and functions). Accommodating loops and conditionals requires nodes that implement controlled branching.

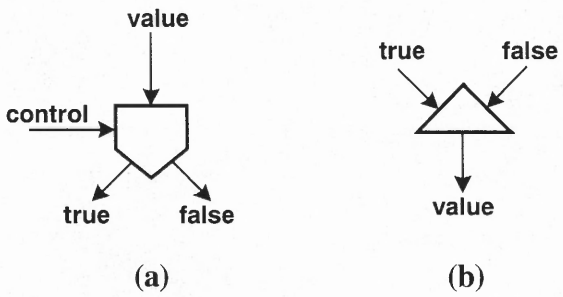


Figure 2.3 (a) A BRANCH node. (b) A non-deterministic MERGE node.

The conditional jump of a dataflow program is represented in a dataflow graph by BRANCH nodes. The most common form is the one depicted in Figure 2.3(a). A copy of the token absorbed from the value port is placed on the true or on the false output arc, depending on the value of the control token. Variations of this node with more than two alternative output arcs or with more than one value port (compound BRANCH) can also be used. A MERGE node does not have a strict enabling rule; that is, not all input ports have to contain a token before the node can fire. In the deterministic variety, the value of a control token determines from which of the two input ports a token is absorbed. A copy of the absorbed token is sent to the output arc. The nondeterministic MERGE node

shown in Figure 2.3 (b) (i.e., a MERGE node without control input) is enabled as soon as one of its input ports contains a token; when it fires, it simply copies the token that it receives to its successors.

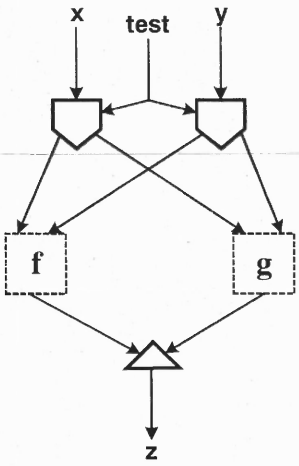


Figure 2.4 Conditional Expression Graph.

Figure 2.4 shows an implementation of a conditional construct graph corresponding to the expression z := **if** test **then** *f(x, y)* **else** *g(x, y)* **fi.** If one token enters at each of the three arcs at the top of the graph, the two BRANCH nodes will each send a token to subgraph ***f*** or to subgraph ***g*** depending on the value of **test**. Only the activated subgraph will eventually send a token to the MERGE node [1].

An iterative loop can also be implemented using the same two specialized nodes, BRANCH and MERGE. Figure 2.5 shows an implementation of a loop construct graph corresponding to the expression **while** *f(x)* **do** *(x, y)* := *g(x, y)* **od.**

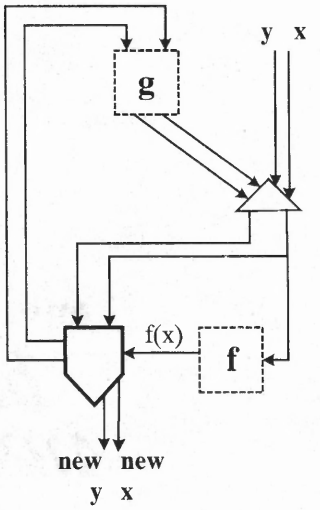


Figure 2.5 Loop Construct Graph

Initially the values *(x, y)* are presented at the input arcs of the nondeterministic MERGE node which simply copies the values to its output arcs, which puts the value *x* at the input arc of the subgraph *f* activating it. The output of the subgraph *f* determines the output of the compound BRANCH. If the value at the control port of the BRANCH node is *true,* then *x* and *y* at the value ports of the BRANCH node are forwarded to the subgraph *g*. If the value at the control port of the BRANCH node is *false,* then *x* and *y* at the value ports of the BRANCH node are sent the other way. Reception of *(x, y)* by the subgraph *g* fires it and produces two new values of *(x, y)*, which are forwarded to the MERGE node. This process continues till the subgraph *f* evaluates a *false* output.

It may be noticed here that there is no exploitation of concurrency in executing this loop. This method is called the lock method [1][12]. It is safe and simple, but not very attractive for parallel machines. The level of concurrency is low since the BRANCH node acts as a lock that prevents the initiation of a new iteration before the previous one has been concluded. An alternative approach is the acknowledge method. This can be

implemented by adding extra acknowledge arcs from the consuming node to the producing node. These acknowledge arcs ensure that no arc will ever contain more than one token and the graph is therefore safe. One arc provides space for one token. In a manner too complicated to show here, the proper addition of dummy nodes and arcs can transform a reentrant graph into an equivalent one that allows overlap of consecutive iterations in a pipelined fashion. The acknowledge method therefore allows more concurrency than the lock method, but at the cost of at least doubling the number of arcs and tokens [1].

A higher level of concurrency can be obtained when each iteration is executed as a separate instance (or copy) of the reentrant subgraph. This *code-copying* method requires a machine with facilities to create a new instance of a subgraph and to direct tokens to the appropriate instance. A potentially more efficient way to implement code copying is to share the node descriptions between the different instances of a graph without confusing tokens that belong to separate instances. This is accomplished by attaching a *tag* to each token that identifies the instance of the node that it is directed to. These so-called *tagged-token* architectures have an enabling rule that states that a node is enabled if each input arc contains a token with *identical tags.* Though this method increases concurrency, implementation of such a technique is not easy and involves considerable overhead [1].

The last problem that needs to be mentioned is the issue of procedure invocation. Calling a procedure introduces similar problems as with reentrancy, to which the methods described above can be applied. In code-copying architectures, a copy of the called procedure is made. In tagged-token architectures, a new tag area is allocated for each

procedure call so that each invocation executes in its own context. Nested procedure calls, recursion, and co-routines can therefore be implemented without any additional problems. An extra facility is however required to direct the output tokens of the procedure activation back to the proper calling site. This is usually implemented as shown in Figure 2.6.

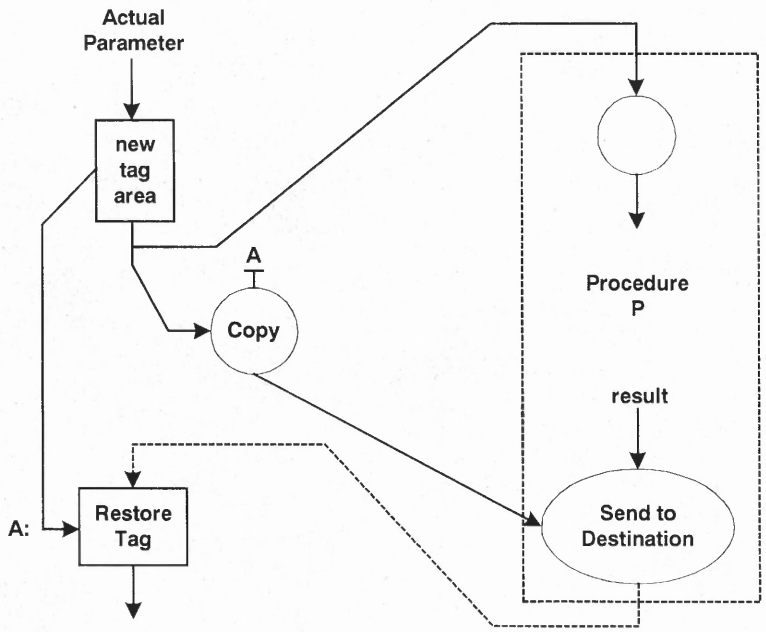


Figure 2.6 Interface for a Procedure Call.

On the left, a call is made to procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P, and concurrently a token containing address A is sent to the output node SEND-TO-DESTINATION. This SEND-TO-DESTINATION node transmits the other input token to a node whose address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which then restores the tag belonging to the calling expression. These

output nodes are special nodes capable of sending tokens to nodes, to which they have no static arc [1].

Machines that handle reentrancy by the lock or acknowledge method are called static; those employing code copying or tagged tokens are called dynamic. Static machines are much simpler than dynamic machines, but for most algorithms their effective concurrency is lower. Algorithms with a predominantly pipelining type of parallelism, however, execute efficiently on static machines with acknowledging [1].

COMMUNICATION IN A DATA FLOW PARALLEL COMPUTER:

Communication in a dataflow machine is accomplished either by *direct* communication or via a *packet communication network* [1].

In *direct communication* machines, adjacent nodes in the graph are allocated to the same processing element or to processing elements that have a direct connection with each other. An important property of direct communication architectures is that the communication medium delivers tokens in the same order as they were received [1].

Packet communication offers the greatest opportunity for load distribution and parallelism in the communication unit, since it can be constructed from asynchronously operating packet-switching modules. Such a module can accept a token and forward it to another module, depending on its destination address [1].

This section covered the issues affecting dataflow computers, and the solutions that researchers have come up with to overcome these issues. The apparently simple concept of data driven execution has many complex implementation issues, yet researchers have come up with brilliant ideas of implementing such machines, which is the topic of the next section.

2.2 Prior Research

Figure 2.7 illustrates dataflow machines categorized according to the nature of the communication unit and the architecture of the processing elements.

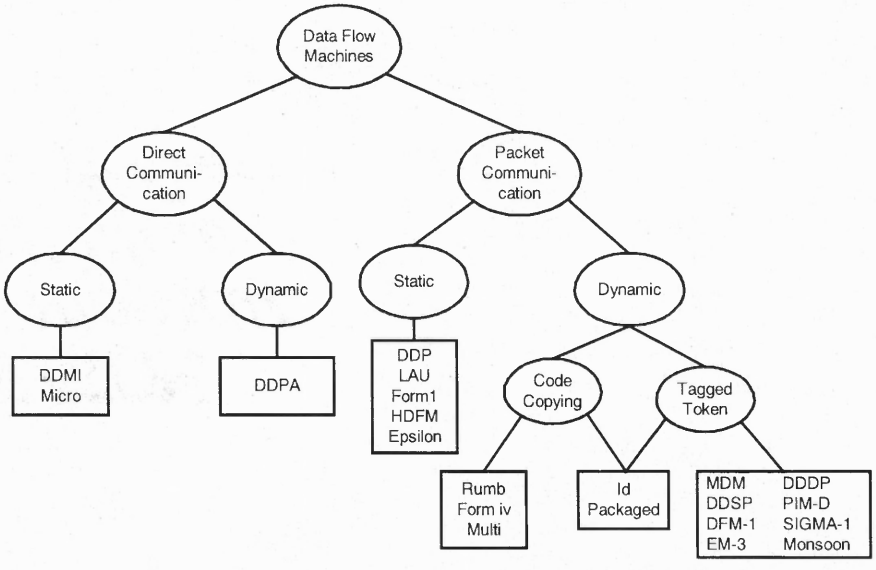


Figure 2.7 A survey of dataflow machines, categorized according to their architecture and implementation. The keys in the boxes refer to the machines [1]

This categorization is not strict, but does broadly cover most dataflow machines implemented or hypothesized till date. At the end of this section, dataflow machines that do not strictly fall under the categories shown in Figure 2.7 will be presented.

2.2.1 Direct Communication Machines

The main drawback of direct communication machines is that for many graphs it is difficult to find a good mapping onto the network (processor allocation problem). It may be a fruitful approach, however, for applications that have predictable and regular communication patterns matching the machine's topology. The most important member of this class is the oldest working dataflow machine, the DDMI [1]. The processing elements of this machine are arranged as a tree. Allocation is simplified by preserving the

hierarchical tree structure of the program. Any internal node of the processing tree can allocate a part of its program (a subtree) to any of its descendants. Allocation is simple and distributed, but far from optimal with respect to load distribution over the processing elements. The root of the tree forms a bottleneck in the communication between processing elements.

In Japan, an interesting dynamic direct communication machine has been developed for large-scale scientific calculations, such as solving partial differential equations [1]. The processing elements are arranged on a two-dimensional grid and use tags to distinguish tokens belonging to different activations. To avoid the necessity to allocate unique tag areas dynamically, the input language is somewhat restricted (no general recursion) so that static allocation is possible. A hardware simulator for 4 x 4 processing elements, each connected to eight neighbors has been used to study small applications. It confirmed analytical predictions that communication delay does not seriously degrade performance, provided that programs have enough parallelism.

2.2.2 Static Packet Communication Machines

The first packet communication dataflow machine that became operational is the Distributed Data Processor [1], built at Texas Instruments. The references suggest that the DDP uses a locking method to protect reentrant graphs. Although the compiler may create additional copies of a procedure to increase parallelism, this copying occurs statically. It is a one-level machine with a ring-structured communication unit, augmented with a direct feedback link for tokens that stay within the same processing element. A prototype comprising four processing elements has been built.

Dennis and his colleagues at the Massachusetts Institute of Technology produced the first designs for dataflow machines [1]. The earliest design had a two-stage structure, an enabling unit (called an instruction cell) dedicated to one node and heterogeneous functional units. This design was later extended into a series of machines differing in the way they handled reentrancy and data structures. They ranged from the elementary Form I processor, which was static and could only handle elementary data, to the full-fledged Form IV processor, which had extensive structure facilities and could copy subgraphs on demand. The prototype that is now operational consists of eight processing elements and an equidistant packet routing network built from 2 x 2 routing elements.

Around 1990, Sandia National Laboratories in Albuquerque, NM, designed and implemented a static dataflow computer called the εpsilon dataflow computer, based on the εpsilon processor [9][11]. The overall εpsilon system was designed as a scalable multi-processor architecture consisting of εpsilon processors and structure memory units connected with a packet switched network. The whole design was implemented on a single board using off-the-shelf components. At the time the group demonstrated a sustained performance of the εpsilon machine comparable to commercial mini-supercomputers.

2.2.3 Machines with Code-Copying Facilities

The dataflow machines with potentially the highest level of parallelism are the dynamic dataflow machines; they employ either code copying or tags to protect reentrant graphs. It is a characteristic of a code-copying machine, that the physical address of a node cannot

always be determined statically. The first detailed design of a dataflow machine was of this type was presented by Rumbaugh in 1975 [1]. Allocation in this machine is per procedure; all the nodes and intermediate results of each procedure are stored in the memory of one processing element. There is a fast connection from the output to the input port of a processing element, such that a circular pipeline is created. Tokens stay within this pipeline unless they are directed to another procedure, in which case they are routed to a special processing element called the scheduler. This scheduler sends a copy of the called procedure and its input values to an idle processing element. If there is no idle processing element, it waits until a processing element becomes dormant and then saves its state (i.e., all the unprocessed tokens) and declares itself as idle. The Massachusetts Institute of Technology Form IV dataflow processor refers to a whole family of designs, some of which have implemented the code-copying scheme.

2.2.4 Tagged-Token Machines

The first tagged-token dataflow machine built was the Manchester Dataflow Machine [1]. The group developed the tagged-token concept to increase parallelism for reentrant graphs independently from similar work done by Arvind and Gostelow in 1977 [1]. The processing element of the Manchester Dataflow Machine had a pipeline of four units: token queue, matching unit, fetching unit, and functional unit. Each unit works internally in a synchronous manner, but they communicate via asynchronous protocols. More than 30 packets can be processed simultaneously in the various stages of the pipeline. To maximize the communication speed, the data paths are all parallel (up to 166 bits wide), transmitting a complete packet at a time. Consequently the sizes of packets, and thus of tokens, are fixed. The token queue is implemented as a simple FIFO buffer.

One of the newest dynamic dataflow machines following the tagged-token communication model is the Monsoon Computer built at the Massachusetts Institute of Technology in collaboration with Motorola Inc. [10]. The Monsoon is an experimental multi-threaded, multi-processor targeted to large-scale, general-purpose scientific and symbolic computations. The Monsoon machine includes a collection of 64-bit pipelined PEs that can execute up to 8 threads simultaneously. The PEs are connected via a multistage packet switch network to each other, and to a set of interleaved I-structure memory modules. An I-Structure memory module is a two stage pipelined structure consisting of a memory stage and an output stage. In the memory stage, an incoming request is decoded into an operation code, memory address, and a value or return continuation [10]. A memory operation is performed on the memory address, and for some operations, a response is generated to be sent to the requesting PE. If a response is generated, it is injected into the interprocessor network in the output stage, which forwards the response to the appropriate PE. The PE nodes implement hardware primitives for direct support of efficient multi-threading, including zero-cycle context switching, single cycle fork and join, and split-phase memory references with arbitrary reordering. The basic run-time execution state of a Monsoon program comprises a tree of activation records, which correspond to the invocation state of many simultaneous procedures that can be executing at the same time.

2.2.5 Other Architectures

The EM-4 dataflow machine was based on the EMC-R dataflow processor, and was a system proposed by the Electrotechnical Laboratory in Japan around 1989 [4]. This architecture denounced the traditional architectural model in dataflow computing of

simple packet-switching, circular pipeline, and colored token style. It introduced the concept of a *strongly connected arc* and that of a *strongly connected block.* In the strongly arc model of representing a dataflow graph, arcs are categorized into two types: normal arcs and strongly connected arcs. A strongly connected arc is a normal arc that the user defines as strongly connected. A dataflow subgraph whose nodes are connected by strongly connected arcs is called a strongly connected block (SCB). Two firing rules were used. One, a node on a dataflow graph is firable when all the input arcs have their own tokens (a normal data-driven rule). The other is that after each SCB fires, all the processing elements which will execute a node in the block should execute nodes in the block exclusively. The architectural detours made in the evolution of this computer were based on these new concepts.

The Cydra 5 Directed Dataflow Architecture that was proposed in 1988 by Cydrome Inc, California, added a new twist to the dataflow paradigm [2]. According to the author Dr. B. Ramakrishna Rau, each executing operation on a data flow computer needs to address five issues:

- Will the operation be executed at all?
- If so, when will it execute?
- On which processing element will it execute?
- Where are the input operands located?
- Where will the result be placed?

Although a directed dataflow computer retains the important benefits of the dataflow architecture, it also makes the concept commercially viable by moving as much

decision-making as possible from runtime to compile time. While in a regular dataflow computer the listed five issues must be addressed at run-time for each operation executed, in directed dataflow architecture these issues are settled at the compile time to the extent possible.

The MADAME computer (MAcro-DAtaflow MachinE) was proposed at the Jozef Stefan Institute, Slovenia in 1991. The idea presented adopting dataflow scheduling in larger chunks of instructions, instead of the traditional instruction level scheduling (*fine-grain* dataflow) [8].

2.3 Motivation and Objectives

The concept of dataflow computers was first introduced to me as an undergraduate student of computer engineering. Its conceptual simplicity struck a chord within me immediately, but at the same time, the lack of methods to efficiently implement the dataflow concept challenged and motivated me to study it further. After investigating the work that other researchers in the field of computer engineering had done on dataflow computers, I realized that engineers were trying to tackle the problem of implementing a data flow computer at the processor level. Dataflow computers in the past have been implemented by using a modified processor, often called the *Processing Element* (PE), which composed of a processing unit along with memory to store partially active instructions and tags. The PE would also contain a matching unit to match the incoming tags, and in addition was assigned the task of sending and receiving tags, from and to other PEs (described earlier in Section 2.1). Most of the inactive instructions would reside in some memory outside the PE.

This thesis presents a way of implementing dataflow computer at the memory level, i.e. intelligent memory is used to perform dataflow tasks. Processors in this system perform the role of execution units; executing whatever instructions the dataflow memory sends it. The motivation behind this idea was that, such a method would incur lesser overhead than the previous methods used to implement data flow computers. In addition, this idea follows the dataflow paradigm very closely, as opposed to the pseudo-dataflow architectures that past researchers have proposed.

The objective of this thesis was to design and implement a proof-of-concept dataflow computer in which the memory is dataflow. Since building this machine on silicon was not a possibility, it was decided that FPGAs would be used instead to build the machine. Speed, efficiency and performance were not primary goals of this design, rather feasibility of the "dataflow memory" (active or intelligent memory) architecture was given priority.

The architecture that is presented in this thesis is a breakaway from the traditional methods of implementing data flow computers, and I believe that this architecture could possibly offer a promising solution in designing an efficient dataflow computer.

3 THEORETICAL APPROACH AND IMPLEMENTING A DATAFLOW COMPUTER

3.1 Overview of the Design

The basic structure of the Data Flow Computer (DFC) is shown in Figure 3.1. It consists of three major parts: the instruction memory *Data Flow Memory* (DFM), the intermediate buffer *Instruction Queue* (IQ) and the execution units *Processor Pool.* In addition there are minor components, such as the *Bank Arbitrator,* the *Bus* and the *Bus Control.*

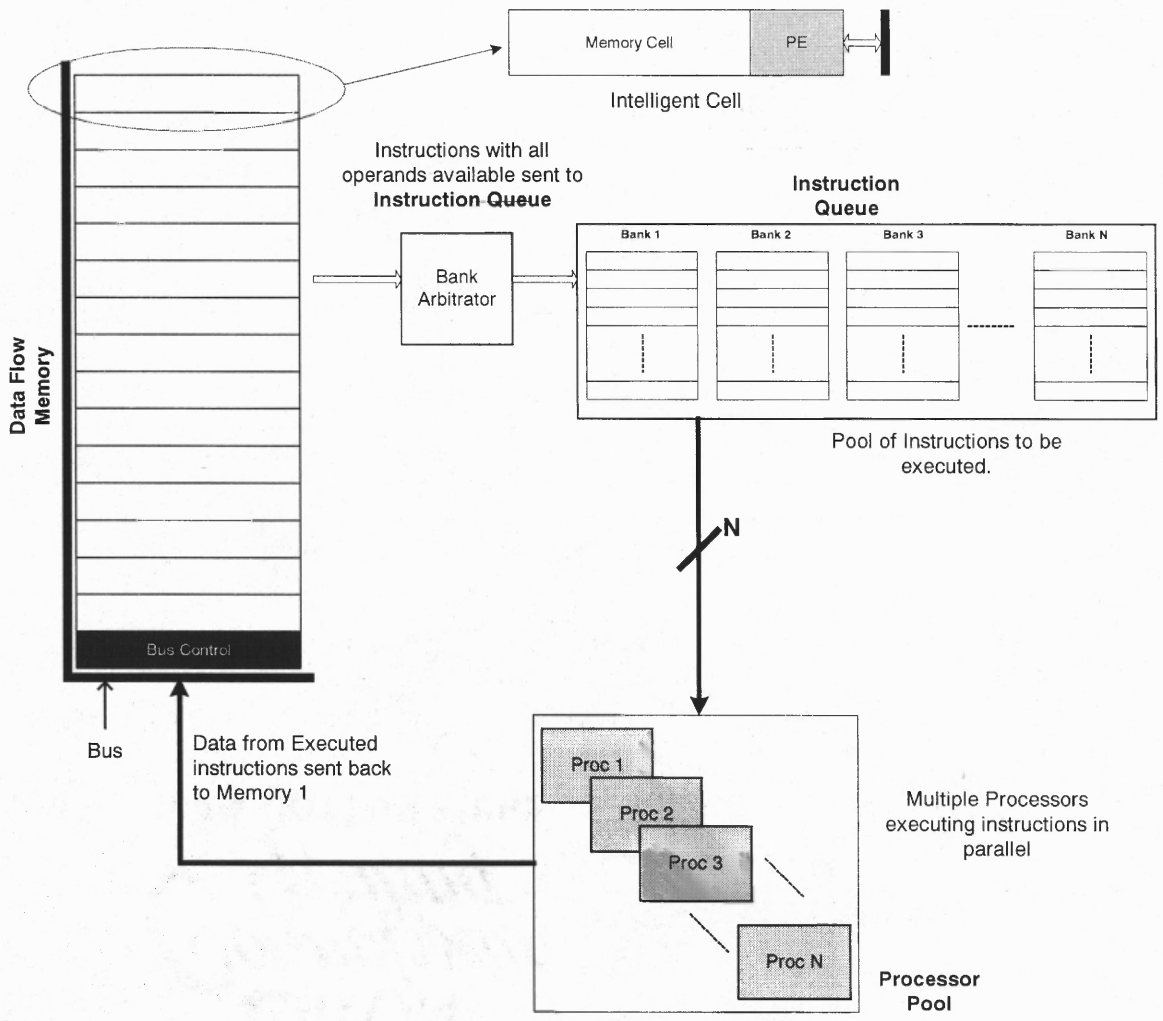


Figure 3.1 Overall Structure of Data Flow Computer

3.1.1 Data Flow Memory

Data Flow Memory (DFM) is the heart of the Data Flow Machine in this design. It is a special kind of intelligent-memory, where **each** memory location is an *intelligent cell* (cell) consisting of two parts, the *instruction part,* and the *processing element* part.

The instruction part is in turn divided into 6 components:

- The instruction opcode (OPCODE).
- Source of the first operand (D1A).
- Source of the second operand (D2A).
- Source of the clause operand (CAD).
- Operand to be obtained from the first source (OPD1).
- Operand to be obtained from the second source (OPD2).
- Flags used to control the behavior of the instruction (FLAGS).

The *result* of an instruction that has finished executing is immediately broadcasted to all the cells in the DFM. Each broadcasted packet contains the *result* along with the *address* of the instruction in DFM that sent it for execution, and is called a *result packet*. The processing element (PE, different from previous dataflow discussions) in each cell is responsible for picking up broadcasted result packets sent by the processor pool via the result bus. If the address in the broadcasted result packet is either equal to D1A or D2A, then the data is written to either OPD1 or OPD2 respectively, and appropriate flags are set. When both OPD1 and OPD2 become available (determined by examining appropriate flags), the PE sends an *executable packet* (composed of the instruction opcode, the operands and the cell address of the sending PE) to the bank

arbitrator. An *executable* is an instruction ready for execution and contains four fields; the instruction opcode, operands 1 and 2, and the cell address of the sending PE. It is important to point out here that the bus on which the result packets are broadcasted is independent (separate) of the bus on which executable packets are relayed to the bank arbitrator. This is done to avoid congestion that would occur if one bus were used.

The CAD field is used to store the address of an instruction that sends the clause. A *clause* is a boolean value stored as a flag in the flags field which acts like a permission for the instruction that needs it, i.e. an instruction will execute only if it's clause field is '1'. The clause is useful in the construction and execution of "conditional" or "looping" program constructs. For an instruction to fire, three flags have to be set, the two operand flags (indicating at both operands are available) and the clause flag.

The internal structure of an intelligent cell in DFM is shown in Figure 3.2. Each section within an intelligent cell is labeled as described earlier. The little arrows within each cell denote the direction of information flow from the cell to the PE. The instruction bus and the result bus are also shown along with the format in which the PE communicates with them.

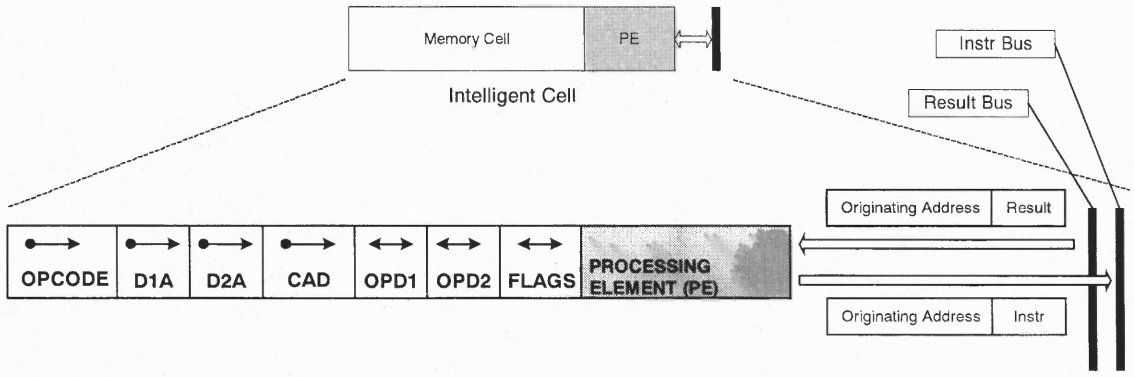


Figure 3.2 Internal Structure of Intelligent Cell in DFM

The *Bus Controller* (BC) in the DFM controls the arbitration of the result packets sent by the processor pool to the DFM. The bus controller assures that the result packets are broadcasted in orderly manner on the *Result Bus.*

3.1.2 Instruction Queue

The *Instruction Queue* (IQ) is an intermediate buffer between DFM and the Processor Pool, and is made up of the *Bank Arbitrator* (BA) and multiple banks of memory. The number of banks in IQ is equal to the number of processors in the processor pool. Each processor is assigned one bank exclusively. This assures that all processors can access memory at the same time. Ideally, a cross bar switch should be implemented so that a processor can access another bank if no instructions are available for execution in its own bank, but this method was avoided to keep the initial design simple. No processor is busy all the time or idles all the time; this is because the bank arbitrator makes sure that the number of executables allotted to each memory bank, hence to each processor, is the same. It uses a round robin scheme to allocate executables to the memory banks.

The IQ acts as a buffer from where the processor pool can access the executables. This leaves the DFM to do its tasks without worrying about distributing executables to the processor pool. When instructions are ready for execution, the DFM simply dispatches the corresponding executables to the IQ, where they await execution. This concept of a two level memory structure, one where all instructions are held, and another where only instructions ready for execution are held is similar to an idea presented by Dr. Dennis of the Massachusetts Institute of Technology [13].

3.1.3 Processor Pool

The processor pool is simply a pool of execution units. Each processor sequentially executes ready instructions (executables) from the bank that it is assigned. The instructions are executed in no particular order because all the instructions in the memory bank are waiting to be executed.

When any processor obtains an instruction, it also gets the originating address of the instruction. After execution, the obtained result along with the originating address (result packet) is sent to the *Bus Controller* of the DFM, which then broadcasts the packet to all the cells via the result bus.

3.1.4 Flow of Data and Instructions in the Data Flow Computer

In Data Flow Memory.

1 The memory cell in DFM that has all its data (OP1 and OP2) and appropriate flags set, sends an executable out for execution (to the bank arbitrator). A seed can be used to initiate this.

In Intermediate Buffer.

2 Once the bank arbitrator receives the executable packet, it places that information in the appropriate memory location. It chooses the appropriate location by first choosing the correct memory bank, which it selects using a round robin scheme (it keeps track of the next bank it must use), and then selecting the next available location in that memory bank.

In Processor Pool.

3 Each processor in the processor pool executes the available ready instructions (executables) sequentially, stepping through them as in a Von Neumann machine.

After the processors associated with their respective banks retrieve the executables, the bank is notified and those locations from where the executables were retrieved are made available to store new executables.

4 Each processor in the pool executes the ready instruction that it retrieved from the IQ and then sends a result packet to the bus controller in the DFM.

In Data Flow Memory.

5 The bus controller broadcasts this packet to all the cells in memory.

6 All the cells in DFM, pick up the broadcast packet. A cell absorbs the packet only if the originating address of the broadcast is equal to either one of its operand source addresses. (The PE of each cell does this comparison.)

7 Once a cell gets both its operands, the PE of that cell then sends a packet containing the executable (opcode + operands + cell address) to the bank arbitrator.

It may feel that broadcasting messages could cause considerable congestion on the bus, but the number of messages on the bus is never greater than the number of processors in the processor pool, since, besides the broadcast of the initial data (the seed used to initialize), all the broadcasting is done only by the processors.

3.1.5 Remarks

This design presented here is suitable for small-scale data flow computers, i.e. a machine having about 16-32 processors in the processor pool. This restriction is foreseen due to two reasons. First, the number of memory banks in IQ will increase linearly with the number of processors, which for a large number of processors such as 1024 will be unrealistic. Secondly, having a large number of processors increases the number of broadcasted messages causing congestion on the broadcasting bus.

This thesis shows a new way of implementing a data flow computer, at the memory level (using intelligent-memory). Taking advantage of current advances in Field Programmable Gate Array (FPGA) technology, an FPGA-based prototype will be developed with emphasis on feasibility of the design. Performance currently is a secondary issue and is left for future research.

3.2 Field Programmable Gate Arrays (FPGAs)

Fastest performance is achieved when a design is implemented directly on silicon, with dedicated logic for all the different units. Since this project was a prototype, it was more important to check the feasibility of the design than performance. Hence a device was needed that could be easily reconfigured, if design flaws were detected. In addition to this, the device was required to have sufficient logic to be able to accommodate the different components of the design. The ideal solution for prototyping the dataflow computer was to use a FPGA.

FPGAs are user-programmable devices, which are widely accepted as an excellent technology for implementing moderately large digital circuits. They offer a cost-effective solution for prototyping, and have a much faster turn around time. Since FPGAs can be reprogrammed in the field, they can be used in innovative designs where hardware needs to change dynamically to adapt to different user applications. Though dynamically changeable hardware is not a consideration for this project, the ability to reprogram an FPGA unlimited number of times is particularly useful when prototyping, where the design is constantly being changed and updated. They also provide other advantages, such as shortening design and development cycles. In terms of speed-

performance, most FPGAs are slower than Complex Programmable Logic Devices. However, the rapid advance in FPGA technology is quickly closing the gap on speed and device density. One prominent disadvantage using FPGA technology is that circuit propagation delays are dependent on the performance of the design implementation tools used. This however is not a handicap for this project.

The internal architecture of an FPGA consists of several uncommitted logic blocks in which the design is to be encoded. The internal logic blocks consist of several universal gates that can be programmed to operate like multiplexers, decoders, logic gates, registers, transistors, random access memory and a slew of other digital logic primitives. The internal logic blocks are connected through a maze of programmable interconnects which can used to implement buses and logic interconnection. They have elaborate clocking schemes and optimization methods (based on design tools) that could produce faster hardware or use lesser logic blocks. Since all the logic blocks are independent, multiple units can be built in a single FPGA, all of which work independently and in parallel (key in the design of a parallel computer, such as a dataflow computer).

The FPGAs used in this design are those made by Altera Corp. Three models were chosen to implement the three major pieces of the dataflow computer; the DFM was implemented on a FLEX10KE, the IQ along with the bank arbitrator was implemented on the ACEK1K and the processor pool was implemented on a MAX9000.

The FLEX10KE was an ideal candidate for implementing the DFM because each device could provide up to 98,304 RAM bits that could be configured as dual-port memory for the DFM [17]. In addition each device contained sufficient logic, up to

200,000 gates depending on the device chosen, to implement the PEs in the DFM. All this is connected together by a fast interconnect network that has predictable interconnect delays [17].

The ACEX1K is a less powerful relative of the FLEX10KE. It has the same features as the FLEX10KE, except there is less of everything. The largest ACEX1K has 49,152 RAM bits and about 100,000 gates [17]. Since the logic and memory requirements for the IQ are less than that of the DFM, this device was an ideal device to implement the IQ in.

The MAX9000 is the smallest of the three devices used. This device contains no memory bits, but has sufficient logic gates and flipflops, up to 12,000 and 772 respectively, which was sufficient to implement the processor pool [19].

A major factor in deciding to use Altera's FPGAs to develop the prototype was the availability of a free development kit from Altera called *MAX+II BASELINE version 10*. In addition, Altera also supports university programs, through which it is possible to get free manuals on how to use the software, programming language reference (AHDL, Altera Hardware Definition Language), sample boards for hardware development, and most important of all, a technical help line to resolve problems faced during development.

In short, Altera FPGAs provided a low cost, highly configurable solution, along with a simple environment to develop the dataflow prototype. The FPGAs used for this project were appropriate for testing and verification, and have proven to be a very important tool for successful project completion.

3.3 Detailed Layout and Implementation of the Design

The detailed design of the proposed dataflow computer is presented in this section. Each component (DFM, IQ and Processor Pool) is covered in depth. This section also discusses the decisions made while implementing the proposed dataflow computer.

3.3.1 DFM Architecture

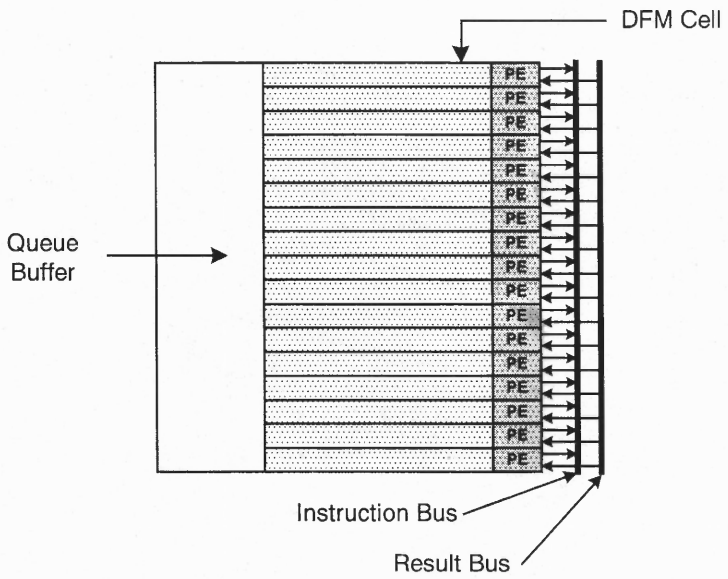


Figure 3.3 Dataflow Memory Structure

Besides the two buses shown in Figure 3.3, the DFM is divided into two sections, the *Queue Buffer* (QB) and the *DFM Cells*. The internal structure of a DFM cell is shown in Figure 3.4.

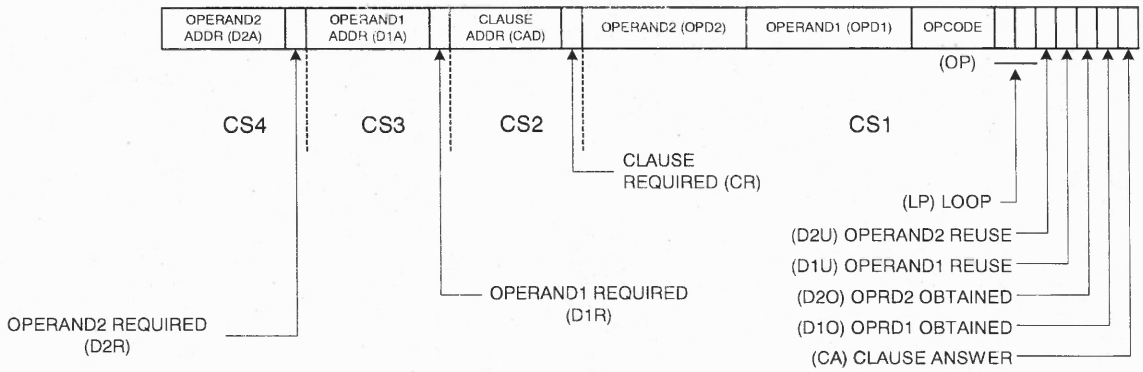


Figure 3.4 DFM Cell Structure

Each DFM memory cell is broken up into four sections, called *Cell Sections* (CS), CS1 through CS4. Each cell section, CS1 – CS4 holds a piece of the instruction to be executed. Controlling each CS, CS1 through CS4, is a 'Logic Unit' or LU. The LUs, LU1 through LU4, managing CS1 through CS4, respectively, constitute a PE. The bits in the CS's are grouped such that minimum communication is needed between the different LU's controlling the different CS's. By having an LU controlling only one CS, work on each CS is done independently and in parallel, avoiding potential waits that would arise if an LU controlled more than one CS across the cell.

CELL SECTIONS:

Cell Section 1(CS1): CS1 is made up of nine fields. Each field is described below.

Operand1 (OPD1): OPD1 is a field that holds operand1, which is received from the instruction whose address matches the value in D1A (source of operand1). OPD1 is sent to CS1 after LU3 picks up a result packet and makes a match between the *originating address* (OA) in the packet and D1A.

Operand2 (OPD2): OPD2 is a field that holds operand2, which is received from the instruction whose address matches the value in D2A (source of operand2). OPD2 is sent to CS1 after LU4 picks up a result packet and makes a match between the *originating address* (OA) in the packet and D2A.

Opcode (OP): OP is the opcode field, which holds the instruction to be executed.

Operand1 Obtained (D1O): D1O is a one-bit flag that is set by LU1. The flag is set when LU1 receives OPD1 from LU3. This bit may be set at compile time or at run time. If it is set at compile time then the operand is already available (immediate value).

OPRD2 Obtained (D2O): D2O is a one-bit flag that is set by LU1. The flag is set when LU1 receives OPD2 from LU4. This bit may be set at compile time or at run time. If it is set at compile time then the operand is already available (immediate value).

Clause Answer (CAN): This is a one-bit flag value which holds the boolean value that LU2 picked up from a message it received. This boolean value is sent by LU2 to LU1 when LU2 picks a message whose address matches the value in CAD. It is a bit used during execution of conditional and loop constructs. The CAN bit can be set at compile-time or at run-time. When a clause is not required to execute an instruction, this bit is set to '1' at compile time.

Operand1 Reuse (DIU) and *Operand2 Reuse (D2U):* Fields D1U, D2U and LP (described below) are used in implementing a loop, e.g. a FOR or WHILE loop. Often instructions from inside a loop require values from outside the loop, but instructions

from outside the loop execute and transmit their values only once (i.e. an instruction is fired only once). Hence the instruction inside the loop will receive that value only once, and it will fire only once. To overcome this problem the reuse bit is used. Setting this bit at compile time allows LU1 to realize that the received value has to be reused, and will not change the D1O/D2O bit of the firing instruction whose D1U/D2U bit/s is/are set. Thus that instruction can fire again when all its other operands/clauses are available.

D1U and D2U are one-bit flags, which are used to determine if operand1 and operand2 need to be reused. These bits are set only at compile time.

Loop (LP): Loops are a terrible construct to manage in Data Flow Machines, but they also are the most commonly used constructs in programming (the 90/10 rule). Loops usually start by checking some value before the start of the loop (e.g. while x < 5, where the value of x is set before the loop begins). This is particularly true for "while" and "for" loops. The next time around, the value of x is obtained from within the loop, which in the case of the example presented, would be an instruction such as "x = x + 1". Thus, the first time, the value of 'x' is obtained from outside the loop and every subsequent time, it obtained from inside the loop. Hence we need a primitive that is capable of obtaining the same variable from two different sources.

The second problem is that data flow machines are runaway machines; firing one instruction subsequently fires many instructions in different parts of the code. In particular when dataflow computers execute loop constructs, due to their inherent runaway nature it is highly probable that the machine may be executing different iterations of the same loop; i.e. part of the instructions in the loop are executing

iteration n, while other parts may be in iterations n+1, n+2, ... , n+m. This would not be a problem if there were no dependencies between consecutive iterations, but would be a disaster if there were any. Some way to control the execution of the loop is needed under these circumstances.

To accomplish these controls, a special 2-bit flag, called *LOOP(LP)* is used to implement 'loop constructs'. The 2-bit flag can only be set at compile time. The values that the LOOP (LP) flag can take are 0-3. The value 0 is used for instructions that are not used to implement a loop, i.e. only instructions that enclose the loop have their LP values greater than zero. All other instructions inside and outside the loop are set to zero. The other three values are used for constructing loops. Note that LP values 1-3 are not used to distinguish between different types of loop constructs, but are used to control how a loop executes.

When LP = 1.

A value of LP equal to 1 is used to initiate a loop (beginning of a loop). The instruction which has its LP flag set to 1, is a special instruction called SP (SPecial), which is not actually an instruction at all (it is never sent for execution). In the example above involving 'x', there would be a SP instruction whose LP value would be 1, its D1A field would contain the address from where the value of 'x' is obtained the first time (outside the loop) and D2A would contain the address from where the value of 'x' is obtained other times (inside the loop). The instruction fires whenever it receives the value of 'x', from either the address D1A or D2A.

When LP=2.

The SP instruction is always followed by a conditional instruction. This instruction

checks the loop conditional. In the example presented above, the conditional instruction would check for x < 5. The result of this conditional statement is sent out as a clause to all the instructions inside the loop. This instruction has its LP flag set to 2, because if it is treated as a regular instruction (LP = 0), upon execution its CAN bit will be reset instantly along with its D1O and D2O flags, and this instruction will not execute again when it receives its operands for the next iteration (since CAN is reset). For the loop to iterate, all the instructions inside the loop need the clause coming from this loop conditional instruction. To avoid such a situation, the LP flag of the loop conditional instruction is set to 2, which instructs LU1 to leave the CAN bit intact after it has sent the conditional instruction for execution.

When LP=3.

An instruction with a value of LP equal to 3 is used along with the loop instruction SP to avoid situations where the runaway effect will cause a problem; i.e. a control of iterations has to be maintained due to data dependencies between consecutive iterations. When a LP value of 3 is used, an effect similar to when LP equals 1 goes into action, except that instead of firing when the value of 'x' is obtained from either D1A or D2A, SP will fire only when the value of 'x' is obtained from both D1A and D2A. D1A is always the address of the SP instruction (beginning of loop), and D2A is always the address of the instruction, which must be executed before the next iteration starts. In the example presented it would be an instruction which modifies 'x' (example, x = x+1).

Cell Section 2(CS2): CS2 is made up of two fields.

Clause Address (CAD): The CAD field is used to store the address of an instruction that sends a clause. A *clause* as described in the previous section is a boolean value which acts like a permission for the instruction that needs, i.e. an instruction will execute only if it's clause field (in CS1) is '1'. If an instruction does not need a clause, then clause field of that instruction (in CS1) is set to '1' at compile time and the value in the CAD field is irrelevant.

Clause Required (CR): CR is a one-bit flag that is set at compile time, which determines whether the execution of an instruction depends on a clause.

Cell Section 3 (CS3): CS3 is made up of two fields.

Operand1 Address (D1A): D1A field of an instruction holds the address of the instruction from which it is expecting its first operand. If the first operand is an immediate value then the address in this field is irrelevant.

Operand1 Required (D1R): D1R is a one-bit flag set at compile time that lets LU3 know if this instruction needs the first operand. When OPD1 is an immediate value, then this bit is set to 0.

Cell Section 4 (CS4): CS4 also contains two fields.

CS4 is identical to CS3. The fields are D2A (Operand2 Address) and D2R (Operand2 Required).

LOGIC UNITS:

Logic Unit1 (LU1): The LU1 in each intelligent cell controls CS1 and takes values sent to it by LU1, LU2 and LU3, inserts them in the appropriate fields and sets the appropriate

flags. After LU1 sets the flags, it fires the instruction if all the required operands and clauses have been obtained. Depending on the values of the LP, D1U and D2U fields, the values of the D1O, D2O and CAN bit may be reset.

Logic Unit2 (LU2): The LU2 in each intelligent cell controls CS2. It takes the result packet broadcasted on the result bus and compares the originating address (OA) in the packet to the CAD field in the CS2 it is controlling, if the CR flag is set.

Logic Unit3 (LU3): The LU3 in each intelligent cell controls CS3. It takes the result packet broadcasted on the result bus and compares the OA in the packet to the D1A field in the CS3 it is controlling, if the D1R flag is set.

Logic Unit4 (LU4): The LU4 in each intelligent cell controls CS4. It takes the result packet broadcasted on the result bus and compares the OA in the packet to the D1A field in the CS4 it is controlling, if the D2R flag is set.

The exact actions taken by each LU is presented in the implementation of the DFM. Note that no distinction is made when packets are picked by the three LUs. Each LU picks up and operates on every packet broadcasted.

QUEUE BUFFER (QB):

The queue buffer (QB) is a repository where result packets coming on the result bus are deposited in case the results are coming faster then they can be absorbed. If result packets arrive before the LUs have finished absorbing the packets that arrived earlier, then the new result packets are queued in the QB. As the LUs become free, the QB sends the queued result packets in the order that they were received. The order of dispatching the

result packets from the QB is not a requirement, but is a side effect of how the queue is implemented, that is a FIFO (First In First Out).

The next section covers how the DFM was implemented and the design decisions that were made that did not completely conform to the design presented in this section. The reasons for these decisions are also presented in the next section.

3.3.2 Dataflow Memory Implementation

The structure of the DFM single cell is shown in Figure 3.5. Each memory cell is 61 bits long, and is broken up into four sections CS1 – CS4, as described before. Each section is of varying length. All addresses, i.e. *operand1 address (D1A), operand2 address (D2A),* and *clause address (CAD)* are 11 bits long, thus giving a total addressable memory of 2K. All operands are 7 bits wide. This small operand size is not considered a limitation, because our main purpose is to prove the viability of our design.

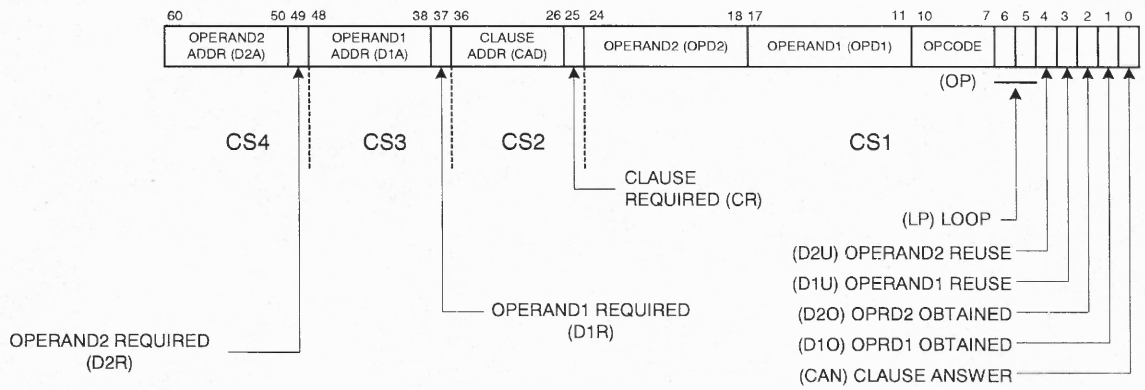


Figure 3.5 DFM Cell Structure

While implementing the DFM, some design decisions were made so that the DFM would fit in a single Altera device. One in particular was the implementation of the PE, i.e. LU1 – LU4. Ideally each intelligent cell should have its own set of LU1 – LU4, but

that required more logic than would fit in a single Altera device. So each LU was assigned to a group of cells called a sub-block, instead of a single cell, as shown in Figure 3.6.

Each LU controls a sub-block of 256 cells, for e.g. LU1 would control a sub-block of 256 CS1s, LU2 would control a sub-block of 256 CS2s and so on. The group of five sub-blocks, formed by LU1 – LU5 forms a block. There will be eight blocks in this design, thus giving a total space of 2K memory words. Notice that a new LU, LU5 is introduced in the figure above. This LU is used to control the QB in a block and is not part of the PE, which is made up of LU1 – LU4. LU5 takes no part in the manipulation of instructions.

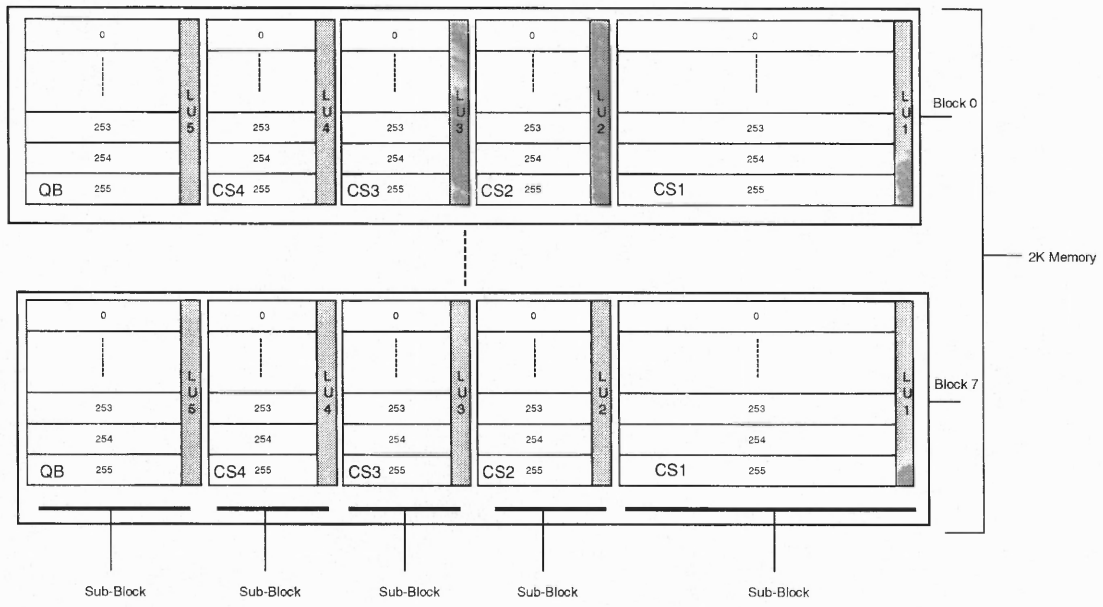


Figure 3.6 Internal Structure of the DFM.

BLOCK:

Figure 3.7 shows the internal structure of a block. The block, as mentioned before, consists of the 5 sub-blocks. Instructions are loaded into each block using the LOAD line,

along with the LOAD ADDR lines and LOAD DATA lines. Once the instructions are loaded, each block waits for the first initial result to arrive before an instruction is fired. The source of the result coming on the Result Bus is either the processors in the processor pool, or any of the LU1s from each block. The source may also be an external user input. To put a result on the Result Bus, the sender should get permission from the Result Bus Controller. This controller manages the information flow into LU5 of each block, which in turn determines if the result packet should be queued or sent straight through to LU2, LU3 or LU4.

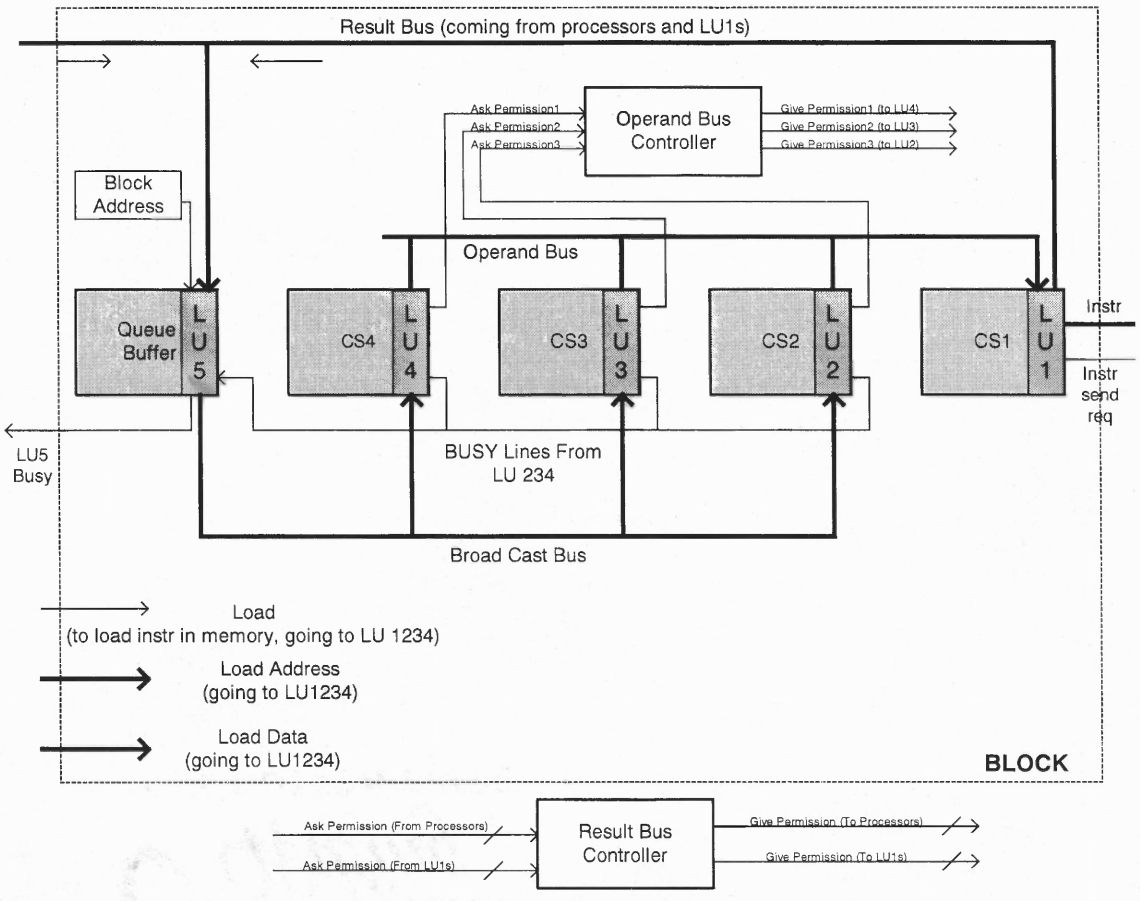


Figure 3.7 Internal structure of the Block

Once LU5 gets a result off the Result Bus, each of them broadcasts this information to the LU2, LU3 and LU4 units (to be collectively referred to in the future as LU234) in their respective blocks, if LU234 are not busy. If the LU234 are busy, then the incoming result is queued into the QB memory to be dispatched later when LU234 are free. The information flow is synchronous, i.e. permission to put result on the Result Bus is allowed only if LU5 in every block is not busy.

Upon reception of the result by LU234, each LU checks if any of the 256 instructions contained in its cell section requires the result. If an LU finds an instruction that needs the result, it dispatches the result to LU1. The Operand Bus Controller controls the dispatching of results by LU234 to LU1. On receiving an operand (result), LU1 dispatches the *executable* for execution if all the operands for the instruction are available. LU1 uses the Instr_Send_Req line to confirm if it is safe to send an instruction for execution.

The last component in the block is the Block_Addr, which is used for house keeping operations. The three most significant bits of the 11-bit address space determines the block address, thus giving a total of 8 blocks in the 2K DFM structure.

LOGIC UNITS:

The logic units are the heart of the intelligent cell architecture (or, in this case, intelligent sub-block architecture). The logic units fire instructions for execution when their operands become available, keep tabs on those instructions that have already fired and the ones that need to refire (like in loops). In addition, the LUs also determine if a result has arrived for a particular instruction (matching operation). The LUs use the flags present in each CS to perform their tasks properly.

The basic function of LU234 is to process anything that is sent by LU5 in the format shown in Figure 3.8. LU1 processes operands and clauses sent by LU234, and is responsible for the firing of instructions when all operands/clauses become available. LU1 dispatches ready instructions (executable) to the IQ via the 'Instruction Bus," in the format shown in Figure 3.9. LU5 looks out on the 'Result Bus' for results arriving from the 'Result Bus Controller' (these are values sent out from the processors and LU1s). The LU5 captures the 'result' and either dispatches it to LU234 or queues it based on the busy state of LU234. The result is forwarded to LU234 if it is not busy and queued if it is.

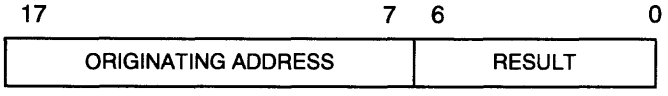


Figure 3.8 Messages on the Operand Bus / Result Arrival Format

Figure 3.9 Instruction Dispatch Format

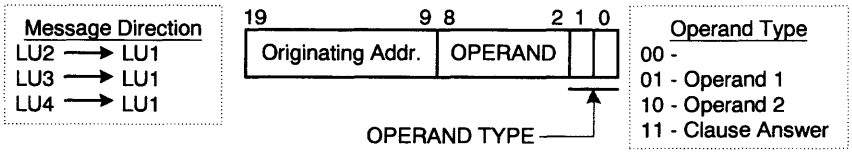


Figure 3.10 Message Communication format between LU2/LU3/LU4 and LU1

FUNCTIONAL DESCRIPTION OF THE LUs.

LU1: Upon receiving an operand in the format shown in Figure 3.10, the LU1 takes the following actions.

```
Goto to Cell (Originating Addr.)
If Operand Type == 01 then
   OPD1 = Operand
   D1O = True
Elseif Operand Type == 10 then
   OPD2 = Operand
   D2O = True
Else
   CAN = Operand
Endif

If (LP == 1 && CAN && (D1O # D2O)) then
   Dispatch Operand on the operand bus
   D1O = D2O = false
Elseif (LP == 3 && D1O && D2O && CAN) then
   Dispatch Operand1 on the operand bus in format shown in
                                                    Figure 3.8
   D1O = D2O = false
ElseIf (D1O && D2O && CAN) then
   Dispatch instruction on the instruction bus in the format
                                              shown in Figure 3.9
   If not (D1U) then
      D1O = false
   Endif
   If not (D2U) then
      D2O = false
   Endif
   If LP == 0 then
      CAN = false
   Endif
Endif
```

LU2: When a LU2 picks up a message from the operand bus, the following actions are performed on the CS2 it controls.

```
For i = 1 to 256
   If CR[i] then
      If CA[i] == Originating Address
         CO[i] = True
         Send CAN to LU1 in the format shown in Figure 3.10
      Endif
   Endif
Endfor
```

LU3: LU3's actions are similar to those of LU2, except that LU3 looks out for OPD1. The following actions are performed on the CS3s that the LU3s control when they pick up a message.

```
For i = 1 to 256
   If D1R[i] then
      If D1A[i] == Originating Address
         Send OPRD1 to LU1 in the format shown in Figure 3.10
      Endif
   Endif
Endfor
```

LU4: LU4's actions are exactly like those of LU3, except that LU4 looks for OPD2 instead of OPD1.

```
For i = 1 to 256
   If D2R[i] then
      If D2A[i] == Originating Address
         Send OPRD2 to LU1 in the format shown in Figure 3.10
      Endif
   Endif
Endfor
```

LU5: LU5 picks up results in the format shown in Figure 3.8. On the arrival of a message, LU5 performs the following actions.

```
If LU234 busy
   If queue not full
      Enqueue result
   Else
      Hold result in temp register and send busy signal
   Endif
Else
   Dispatch message on Operand bus in format shown in Figure 3.8
   If queue not empty
      Dequeue result
      Dispatch message on Operand bus as shown in Figure 3.8
   Endif
Endif
```

This completes the functional description of the DFM. The next section covers the detail design and implementation of the Instruction Queue (IQ).

3.3.3 Instruction Queue (IQ)

The instruction queue is composed of two major parts, the bank arbitrator and the memory banks, as shown in the shaded portion of Figure 3.11.

MEMORY BANKS:

A layout of the memory bank within the IQ is shown in Figure 3.12. Each bank has two pointers called the *New Instruction Load Pointer (NILP)* and *Next Instruction Execute Pointer (NIEP),* for forming a circular queue. The NILP holds the address of the next available location where the bank arbitrator can insert a new instruction coming from the DFM. Each bank in the IQ also has a NIEP, which holds the address of the next instruction that is ready to be executed. The processor associated with the bank uses this address to fetch the next instruction to be executed.

Once an instruction has been accessed for execution by a processor or a new instruction is loaded into the memory bank, both the NIEP and NILP pointers are incremented using modulo 2^m arithmetic.

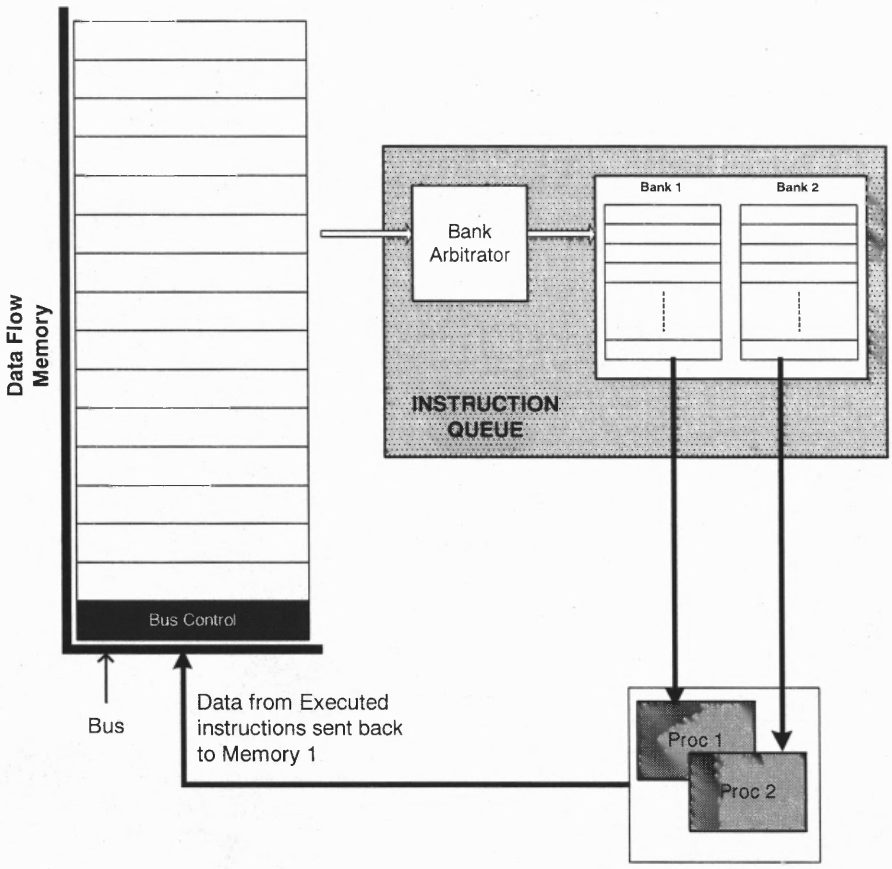


Figure 3.11 Structure of Dataflow Computer Implemented in this Thesis

Each memory bank contains two parts, a group of memory cells (that hold executables sent from the DFM) forming the circular queue, and a *Memory Controller* (MC) that arbitrates the values going in and out of the circular queue. The structure of a single cell in a bank is shown in Figure 3.13. The executable has already been described earlier (Section 3.1.1.).

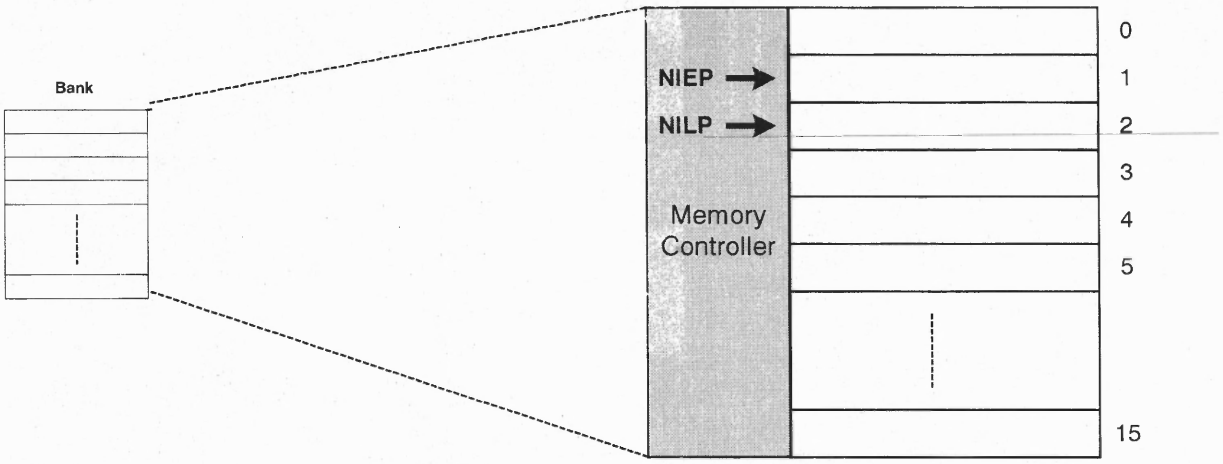


Figure 3.12 Layout of Memory Bank in IQ

When the BA informs the MC associated with a selected bank that a new instruction is put on the bus, the MC reads the instruction from the bus in the format shown in Figure 3.13 and puts it in the location pointed to by the NILP. The NILP is then incremented by the MC of the active bank to the next location (NILP+1), if it is available. The availability of the next location is determined by the value of the NIEP, which holds the address of the next instruction that is to be sent for execution. If the NILP equals NIEP – 1, the MC raises a flag which tells the BA that the buffer is full and not to send any more instructions.

Originating Address	OP1	OP2	INSTR

Figure 3.13 Structure of an IQ Memory Cell

The two pointers are used to manage the circular queue. When the values of the two pointers are the same, then the queue is considered empty, and when NILP = NIEP – 1 the queue is considered full. The MC in each bank uses two signals to communicate

with its processor; the Instr_RD signal is used by a bank to indicate to it's processor that a new instruction(s) is/are available for execution. In turn, when an instruction has finished executing the processor notifies it's bank (MC) via the Instr_done signal, upon which the corresponding MC increments NIEP, if NIEP ≠ NILP.

Each memory bank in the memory pool is implemented using a dual port memory, allowing the BA to send instructions to it while the processor associated with that memory can read from it. The BA, as described earlier, puts instructions into each bank using a round robin scheme. Each bank has sixteen memory words that are used to implement the circular queue, as shown in Figure 3.12. The number of memory words in each bank is not an optimized number, but an arbitrary number chosen to build the prototype. Messages are obtained and stored in the circular queue by the MC of each bank in the format shown in

Figure 3.14. Each memory cell in the IQ is 29 bits long and contains an executable. The algorithm followed by each MC is shown below.

Figure 3.14 Format of Instructions received by MC.

```
If NILP ≠ NIEP - 1
      Wait for INS to arrive
      Set busy high
      Store INS at location (NILP)
      Set busy low
      Increment NILP
Else
   Set busy high
End if
If NILP ≠ NIEP
   Set instr_avail high
   Wait for instr done signal
   Increment NIEP
```

```
End IF
```

BANK ARBITRATOR:

The BA's only job is to route the incoming instructions to the memory banks. For simplicity, it uses a round robin scheme to fill the banks. When an instruction arrives, the BA looks at a register called the Memory Pointer (MP). This MP tells the BA which bank it is supposed to write the next incoming instruction into. The address in a bank where the instruction should be written to is contained in the NILP. When the BA receives an instruction in the format shown in Figure 3.14, it simply forwards that instruction to the bank pointed to by the MP. The MC associated with the memory bank receiving the instruction then writes the instruction to the address pointed to by its NILP.

The process of how the BA handles instructions it gets from the DFM is shown in the algorithm below. When an instruction arrives, it is first stored in a local register, and the bank pointed by MP is notified. Next it checks if the memory pointed to by the MP is currently busy; if not busy, the Arbitrator puts the instruction on the bus and notifies the bank about it. If the bank pointed by the MP is busy (either due to a pending write from a previous insertion of an instruction or the bank is full), the MP is incremented and in the next internal cycle the BA tries to put the instruction in another bank. This procedure is explained in pseudo-code below:

```
MP = 0                          ; Initialize MP
Wait for INS to arrive          ; INS - instruction from DFM
Store INS                       ; Store INS in local register
Bank Arbitrator sets itself busy ; so no new instr arrive
Enable MP                       ; Enable memory pointed by MP
If (Bank(MP) not busy)
   Put instruction on bus
   Notify bank(MP)
```

```
      Wait for bank(MP) to send ack
      Increment MP
      Bank Arbitrator sets itself not busy
   Else
      Increment MP
   End if
```

3.3.4 Processor Pool

Each processor in the processor pool is associated with only one memory bank in the memory pool, and vice versa. The processors in the pool are specially designed and are more like execution units than general-purpose processors. When a processor is ready for execution, it looks for the Instr_RD signal from its corresponding memory bank. When it sees the signal go high, it reads the instruction from its memory bank in the format shown in Figure 3.14, and executes it. An instruction is always executed with OPD1 on the left side and OPD2 on the right, e.g. OPD1 – OPD2, OPD1 / OPD2, check if OPD1 > OPD2, etc. The processor then sends a request to the Result_Bus_Controller in DFM, asking permission to send the information back to the Data Flow Memory (DFM). Upon receiving acknowledgement from the controller, the requesting processor sends the result over to DFM in the format shown in Figure 3.15, and sends an Instr_done signal to its corresponding bank, which in turn readies a new instruction for execution. There are currently only two processors in the processor pool, but this is not a limitation, only two were used to expedite the implementation. Since all the dataflow work is relegated to the DFM, either processor in the processor pool is blindly executing instructions and sending signals, independent of the other. The instructions that the processor can currently execute are set to the bare minimum, just enough to make this data flow machine work. The instructions and their corresponding opcode are:

ADD	0001 →	Addition
SUB	0010 →	Subtraction
MUL	0011 →	Multiplication
DIV	0100 →	Division
CEQ	1000 →	Compare if Equal
CNE	1001 →	Compare if Not Equal
CGT	1010 →	Compare if Greater
CLT	1011 →	Compare if Less
CGE	1100 →	Compare if Greater or Equal
CLE	1101 →	Compare if Less or Equal

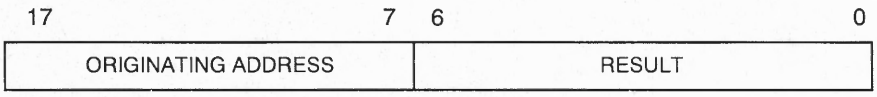


Figure 3.15 Result Dispatch format by a processor

The algorithm followed by each processor is shown below.

```
if instr_rd goes high
   read instr from instruction bus
   execute instruction
   ask permission from DFM to send result
   wait for ack from DFM
   send result along with originating address to DFM
   send instr_done signal to memory 2
end if
```

3.3.5 Programming on the Dataflow Computer

This section covers how the instructions presented in the previous section can be used to write programs that will execute on this dataflow computer. The complete instruction set is presented in the table below, along with their opcode.

Table 3.1 Full Instruction Set

INSTR	OPCODE	INSTR	OPCODE
ADD	0001 → Addition	CGT	1010 → Compare if Greater than
SUB	0010 → Subtraction	CLT	1011 → Compare if Less than
MUL	0011 → Multiplication	CGE	1100 → Compare if Greater or Equal to
DIV	0100 → Division	CLE	1101 → Compare if Less or Equal
CEQ	1000 → Compare if Equal	SP	0000 → SPecial
CNE	1001 → Compare if not Equal	LK	0000 → LocK

All the instructions presented in Table 3.1 are commonly used instructions, except the SP and the LK instruction. This computer has six compare instructions, which is unlike most machines that usually have four. The two compare instructions not usually found in other computers are the CGE and the CLE, because these two instructions can normally by made up by combining the other compare instructions, CLT & CEQ for CGE, and CGT & CEQ for CLE. However, this is not suitable in this architecture because, when a combination of instructions, such as CLT & CEQ, is used to build CLE; instructions within the conditional have to fire either when CLT is true or CEQ is true, i.e. each instruction within the conditional should be capable of receiving "two clauses" and fire if either one is true. This ability to receive two clauses is not provided in this architecture, which if implemented would make the DFM design more complex. To avoid this problem, a conditional is provided for every possible scenario.

The instructions SP (SPecial) and LK (LocK) have already been presented before. This section reiterates their purpose here. The SP instruction is not really an instruction, and it is never sent to the processor pool for execution. Though its opcode presented in Table 3.1 is "0000", it is irrelevant. The SP instruction is a special directive for the LU1, instructing it to forward all values it receives to other cells in the DFM. What constitutes

an SP instruction is the value of the LP in an instruction; if the value of the LP is 1, the LU1 realizes that this is a directive and treats it accordingly. When the LP is set to 1, LU1 sends the operands it receives from either LU3 or LU4 out onto the operand bus if the CAN bit of the SP instruction is set to one. The SP instruction always has its D1R and D2R bits set, but unlike other instructions where LU1 will wait for both operands to arrive before dispatching them, the LU1 dispatches the arriving operands immediately.

The LK instruction is also a directive that has its LP set to 3 (opcode irrelevant). While setting the LP to 1 makes the SP instruction transmit any operand it receives (from either LU3 or LU4) onto the Operand Bus, setting the LP to 3 causes LU1 to transmit OPD1 (the value it receives from LU3), but only after the LK instruction has received both operands OPD1 and OPD2, and the CAN bit is set to one. This instruction is particularly useful in 'Loop' constructs where it is necessary to prevent instructions from different iterations of a loop from executing simultaneously. Concurrently executing multiple iterations of a loop is not always bad and is an important feature in parallel computing, but it is disastrous if there are dependencies between consecutive iterations.

An LP value of '1' (SP) and '3' (LK) act as directives for LU1, and these instructions are never sent for execution. But a LP value of '2' directive is given to an instruction that is sent for execution. An LP value of '2' is only given to a conditional that follows the SP instruction. The conditional instruction following a SP is the one that checks the loop control variable. An LP value of '2' for this conditional ensures that its clause bit is not reset after it executes, which in turn guarantees that this instruction will re-fire to check the loop control variable in following iterations.

Table 3.3 shows a segment of a program, which includes *conditional statements*, *reentrant code* and *reusable variables*. The code is presented in two formats in the table, a high level pseudo-code and a pseudo-assembly. The equivalent program in the dataflow language is presented in Table 3.4. Since a high level dataflow language and compiler was not developed for the dataflow machine, the equivalent code presented in Table 3.4 is not how one would write a program for this computer to be compiled, but is actually the code resident in the DFM. The alternately shaded areas represent the contents of each cell section, which are labeled at the top of the table. For comparison and better understanding, the flow graphs of the two codes (sequential and dataflow) are presented in Figure 3.16 and Figure 3.18.

Before presenting the flow diagram of the dataflow computer, it is important to learn to interpret the nodes of the flow graph of the dataflow code. Figure 3.16 shows the primitives that are used to represent each node in the dataflow graph. There are three basic primitives used to implement a flow graph, which are shown in Figure 3.16 (a), (b) and (c). Figure (d) is a variation of (a). Arrows going in and out of the node are divided into four categories based on *arc type, arrowhead, label type* and *tail.* Table 3.2 shows the classification of arrows associated with the dataflow graph nodes.

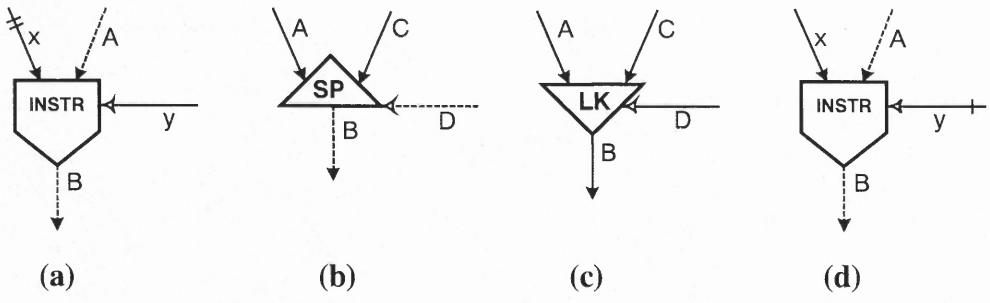


Figure 3.16 Primitives to implement a Program Flow Graph

As mentioned earlier, there are three kinds of nodes that can be used to implement a dataflow graph. The *instr* node is used to implement every kind of instruction this dataflow computer offers, except for the SP and the LK instructions, which have their own nodes presented in figures (b) and (c) above respectively. The two solid headed arrows coming into the top of each node represent the two operands required by the node. The operand arrow on the left is OPD1 and the operand arrow on the right is OPD2. The hollow headed arrow on the side of the nodes is the clause needed by the node and the solid headed arrow at the bottom of each node is the result pushed out by the node.

Based on the explanation given and Table 3.2, Figure 3.16 (a) can be described as a node with the following properties.

- The operand on the left is an immediate value, with value 'x' that has to be reused by the node.
- The operand on the right is obtained from the instruction at location 'A' and it has not arrived yet.
- The clause is also an immediate value (i.e. this instruction does not require a clause) and its value is 'y' (which of course is '1', else this instruction will never fire.)
- This node will perform the operation 'instr' on the operands coming into the node.
- This node is located in memory at 'B' (evident from the label on the result arc), which is sent out with the result. The result from this node has not yet been sent out.

Table 3.2 Classification of Arrows associated with Dataflow Graph Nodes

Arrowhead	Solid Head	Indicates an incoming *operand* or an outgoing *result.*
	Hollow Head	Indicates an incoming *clause*
Arc Type	Solid Arc	Value along that arc is not yet available.
	Dotted Arc	Value along that arc is available
Label Type	Lower Case	Value associated with arc (immediate value)
	Upper Case	Address associated with arc, (incoming or outgoing)
Tail	Single Tail	LP value of the instruction is 2
	Double Tail	Reuse value on the arc.

The style of representing a flow graph for interpreting a dataflow program is new. Though the nodes bear some similarity in form to the ones presented in Section 2.1, their interpretation is completely different. One may use only solid lines for the arcs to show less detail, but all other components are absolutely important to show *reusability* of an arc, the two different arrowheads to show the difference between *values* and *clauses* and the use of the single tail to indicate a LP value of '2'. Using dashed and solid lines shows what state a program is in when it is loaded into memory, giving a clue of the operands that are available to this program and those that it is waiting for. Labeling the arcs with memory addresses does have the drawback of not being able to draw the flow graph until the program has already been written. Eliminating all memory address labels (but leaving value labels in) overcomes this problem but reduces the amount of information the graph displays. Notice how some of the output arcs of the conditionals gets converted into a clause arcs for some nodes; this is allowed because the result from the execution of a conditional is either a '0' or a '1', thus it can be a clause arc for a node.

The high-level language code presented in Table 3.3 has two loops (a FOR loop and a WHILE loop) and three conditionals (one explicit, two implicit). The explicit conditional is the *if* statement, `"If x ≠ 0 then";` the two implicit conditionals are part of the two loop constructs. The conditional in the FOR loop checks if the value of 'i' (loop control variable) exceeds 'x', `"If i > x then."` The conditional in the WHILE loop checks to see if the loop control variable 'x' is greater than '1', `"If x > 1 then."`

The two loops and the three conditionals can be easily observed in Figure 3.17. The two loops execute concurrently and the values being used by either loop are not interdependent. In a Von Neumann machine, it would be quiet difficult to concurrently execute the two loops because the variable 'x' is needed by both the loops; while one (FOR) uses variable 'x', the other (WHILE) modifies the variable 'x'. This of course cannot be done concurrently, the FOR loop must finish executing before the WHILE loop starts as shown in Figure 3.18. However, in the dataflow machine, each loop is sent its own copy of 'x', thus allowing the loops to execute concurrently.

Table 3.3 Sample Code

Calculation of functions F1 and F2 in pseudo-high level language:		**Equivalent Pseudo-Assembly Code:**		
•			•	
•			Get (y)	
Get (y)	← Get y from user.		MOV X, Y	(X) ← (Y)
x = y * 9 - 15	← *Reuse variable(x)*		MUL X, #9	X ← X * 9
If x ≠ 0 then	← *Conditional*		SUB X, #15	
F1 = 0			CMP X, #0	
F2 = 1			BEQ DN2	
For i = 1 to x	← *Loop*		MOV R1,#0	
F1 = F1 + x * (i + 2)			MOV R2,#1	
End For			MOV R3,#0	
While x > 1	← *Loop*	LP1:	CMP R2, X	
F2 = F2 * x			BGT DN1	
x = x - 1			ADD R3,R2	For loop.
Endwhile			ADD R3,#2	R1 is F1
Endif			MUL R3, X	R2 is 'i'
•			ADD R1,R3	R3 is
•			CLR R3	temp
•			ADD R2,#1	
			BRA LP1	
		DN1:	MOV R2,#1	
		LP2:	CMP X, #1	
			BEQ DN2	
			BLT DN2	While
			MUL R2,X	loop.
			SUB X,#1	R2 is F2
			BRA LP2	
		DN2:	•	
			•	

Table 3.4 Equivalent Dataflow Program of the Sample Code

ADDRESS	CS4		CS3		CS2		CS1									
	OPERAND2 ADDR. (D2A)	OPERAND2 REQD. (D2R)	OPERAND1 ADDR. (D1A)	OPERAND1 REQD. (D1R)	CLAUSE ADDR. (CAD)	CLAUSE REQUIRED (CR)	OPERAND2 (OPD2)	OPERAND1 (OPD1)	OPCODE (OP)	LOOP (LP)	OPERAND2 REUSE (D2U)	OPERAND1 REUSE (D1U)	OPRD2 OBTAINED (D2O)	OPRD1 OBTAINED (D1O)	CLAUSE ANSWER (CAN)	
•																
A	GET (X)															
B	0	0	A	1	0	0	9	0	MUL	0	0	0	1	0	1	
C	0	0	B	1	0	0	15	0	SUB	0	0	0	1	0	1	Calculate **X**
D	0	0	C	1	0	0	0	0	CNE	0	0	0	1	0	1	If stmt.
E	0	0	0	0	D	1	1	0	ADD	0	0	0	1	1	0	Initialize **i**
F	L	1	E	1	D	1	0	0	SP	1	0	0	0	0	0	
G	C	1	F	1	D	1	0	0	CLE	2	1	0	0	0	0	For Loop **Start**
H	0	0	F	1	G	1	2	0	ADD	0	1	0	1	0	0	
I	C	1	H	1	G	1	0	0	MUL	0	1	0	0	0	0	
J	C	1	I	1	G	1	0	0	ADD	0	0	0	1	0	0	Calculate **F1**
K	J	1	F	1	G	1	0	0	LK	3	0	0	0	0	0	Lock instr
L	J	0	K	1	G	1	1	0	ADD	0	1	0	1	0	0	Increment **i**
M	0	1	C	1	D	1	0	0	SP	1	0	0	0	0	0	
N	P	0	M	1	D	1	1	0	CGT	2	1	0	1	0	0	While loop **Start**
O	0	1	M	1	N	1	1	0	MUL	0	0	0	1	0	0	Calculate **F2**
P	0	0	M	1	N	1	1	0	SUB	0	1	0	1	0	0	
•																

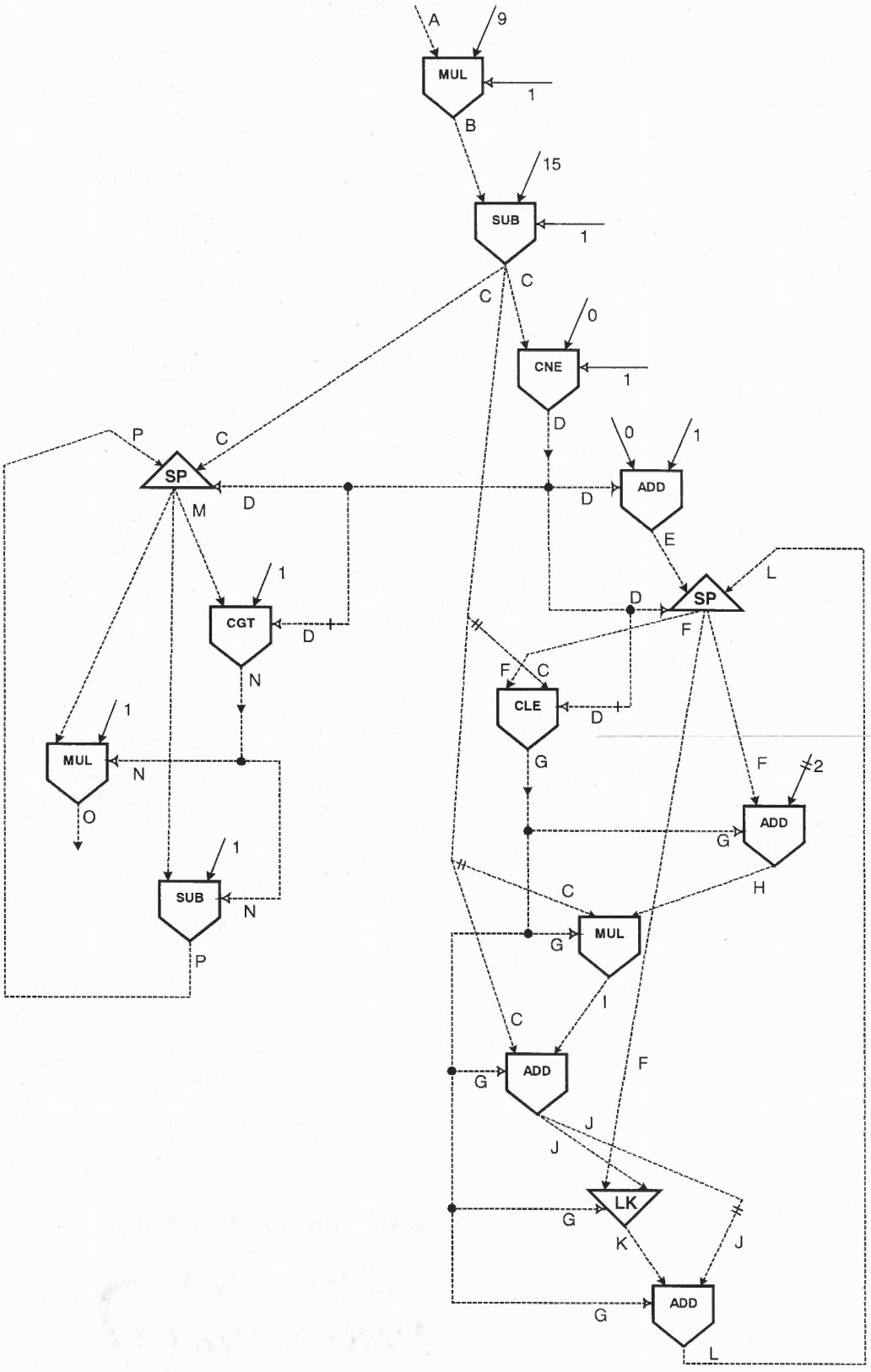


Figure 3.17 Flow graph of the code presented in Table 3.4

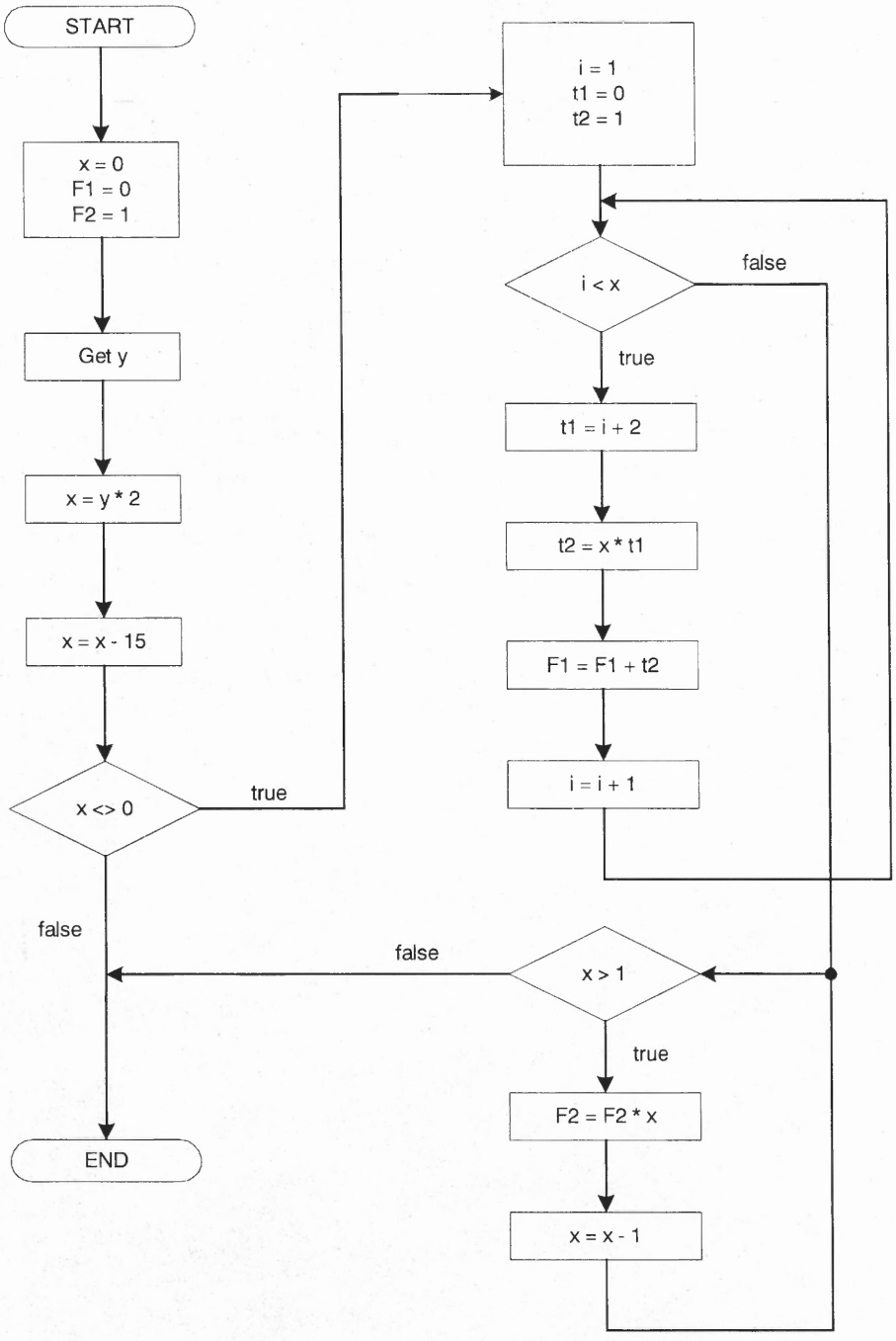


Figure 3.18 Flow graph of the sequential code in Table 3.3

The FOR loop is bounded by an LK instruction, because, there are dependencies between consecutive iterations of the loop. Since these dependencies do not exist between the consecutive iterations of the WHILE loop, no LK instruction is needed to bind the loop.

Writing a program for the dataflow machine is tricky and tedious, because the programmer has to be aware of the data dependencies in the program and address the issue. However, this issue can be overcome using a smart compiler that will automatically take care of the problem.

3.3.6 Altera Implementation

The MAX-PLUS II allows design entry in three forms, a *tdf* (text definition file), *gdf* (graphical definition file) and *wdf* (waveform definition file). Only the first two methods were used in implementing the dataflow computer.

The whole project was developed very modularly, so that each module can be tested and simulated individually. In addition, modularity reduces the complexity and makes debugging easier. The whole design can be categorized into two parts, main components and sub-components. The main components are made up of components that have been already discussed such as *DFM, Cell Sections, Processor Pool, Processors, Instruction Queue, Bank, etc.* The sub-components are additional modules built to make the main components. Each of these is briefly discussed and their relevance presented.

FULL DESIGN:

Figure 3.19 presents the whole design and is composed of a block in DFM, the IQ and the Processor Pool. Due to lack of time, a DFM with only one block was built, though it is

set up to have up to eight blocks (0 –7). This does not degrade the performance of the computer; it is simply a handicap for the program size that can be accommodated by the DFM. In the state presented, the DFM can accommodate a program whose size cannot exceed 254 instruction words. Address 0 and 255 are omitted to eliminate a timing problem that is encountered if used. Instead, these addresses are used as seeds for initiating execution. In this design, address 255 is used to initiate execution by sending a positive clause whose OA is 255 on the Result Bus. This will be seen in programs that were simulated on this machine. In Figure 3.19, *block_csa* and *block_csb* makes up the DFM. Attached to the DFM are the IQ and Processor Pool, as it was originally presented in Section 3.1. The IQ is made of *dfm2* and *proc_*pool makes up the processor pool. It may be noticed here that the DFM block is broken up into two parts *block_csa* and *block_csb.* This had to be done because the entire block could not be fit in a single Altera device.

HEIRARCHICAL TREE:

Appendix A.1 presents the hierarchical tree of the whole design, showing all the main components and some subcomponent. Each node of the tree is presented by the Altera Hierarchical Display in two ways, *component_type : item_number* or *component_type : component_name.* The component_type is the type of primitive or composite that is being used, the item_number is the index of a primitive added to a GDF (Graphic Design File). The component_name is the name of the variable, a primitive or composite is declared by, in a TDF. For example when using a composite called *counter* in GDF file, it will be assigned a index_number by MAX-PLUS II, and when using the *counter* in a TDF file, it is assigned a name by the programmer. So the composite *counter* used in a GDF may

appear in the graph as 'counter : 45', where '45' is the index_number assigned to the composite by MAX-PLUS II, and the composite *counter* used in a TDF may appear in the graph as 'counter : addr_counter', where 'addr_counter' is the name of the composite assigned by the programmer. Each tree grows from left to right, nodes on a vertical branch are all at the same level and nodes on a horizontal branch represent depth, with nodes to the left being higher in hierarchy then the nodes on the right.

The hierarchy and a brief explanation of each component (main and sub) follow.

MAIN COMPONENTS:

BLOCKB3_B: The hierarchy of blockb3_b is presented in Appendix A.2 and its layout is presented in Figure 3.20. Blockb3_b is made of five components, two of which are main components, the *block_cs5h* and *result_bus_controller,* and three sub-components, the *initialize* and two *bus_merge.*

block_cs5h: The block_cs5h is the result queue explained earlier, which queues incoming results when LU234 are busy to accept new results. Both the memory that holds the values and the logic which controls the memory is built into block_cs5h. The hierarchy of block_cs5h and its implementation is presented in Appendix A.3.

result_bus_controller: The result_bus_controller controls the arbitration of the results that are being fed into blockb3_b, by the processors and LU1. The controller is preferential, i.e. there is a priority assigned to everybody sending messages, with LU1 having the highest priority and processor 2 the lowest. . The hierarchy of result_bus_controller and its implementation is presented in Appendix A.4.

initialize: Initialize is a sub-component that is not found in any other modules, hence presented here. This module is used to send the seed clause to start firing instructions. The module sends a seed with a clause value one and OA of 255 to the result_bus_controller when it receives a trigger, which in turn sends it to block_cs5h. The implementation of *initialize* is presented in Appendix A.5.

Since most of the sub-components that are under the main components are common, i.e. the same sub-component appears under multiple main components, they are presented together after the main components.

BLOCKB3_A: Blockb3_a hierarchy is in presented Appendix A.6 and its layout is presented in Figure 3.21. Blockb3_a is made of six components, five of which are main components, the *block_cs1b, block_cs2b, block_cs3, block_cs4* and *lu234_bus_controller* and one sub-component, the *bus_merge.*

block_cs1b: The block_cs1b is the combination of cell section 1 (CS1) and LU1. The hierarchy of block_cs1b and its implementation is presented in Appendix A.7.

block_cs2b: The block_cs2b is the combination of CS2 and LU2. The hierarchy of block_cs2b and its implementation is presented in Appendix A.8.

block_cs3: The block_cs3 is the combination of CS3 and LU3. The hierarchy of block_cs3 and its implementation is presented in Appendix A.9.

block_cs4: The block_cs4 is the combination of CS4 and LU4. The hierarchy of block_cs4 and its implementation is presented in Appendix A.10.

lu234_bus_controller: The lu234_bus_controller controls the arbitration of messages sent by LU2, LU3 and LU4 to LU1. Like the result_bus_controller, the lu234_bus_controller is also preferential, i.e. there is an assigned priority to the LUs sending the message, with LU2 having the highest priority and LU3 the lowest. The hierarchy of result_bus_controller and its implementation is presented in Appendix A.11.

DFM2: The DFM2 is really the Instruction Queue (IQ) and is made up of three main components; the *dfm2_controller* also called the Bank Arbitrator and, two *buffs* also called memory banks. The layout of DFM2 is shown in Figure 3.22 and its hierarchy is presented in Appendix A.12.

dfm2_controller: The dfm2_controller distributes the instructions it receives from the DFM to the two memory banks using a round robin scheme. The hierarchy of dfm2_controller and its implementation is presented in Appendix A.13.

buff: The buff is the sixteen word deep circular queue that stores instructions sent by DFM and arbitrated by the dfm2_controller. Besides the memory implemented as a queue this module also has some logic used to control instructions going into and coming out of the queue. The hierarchy of buff and its implementation is presented in Appendix A.14

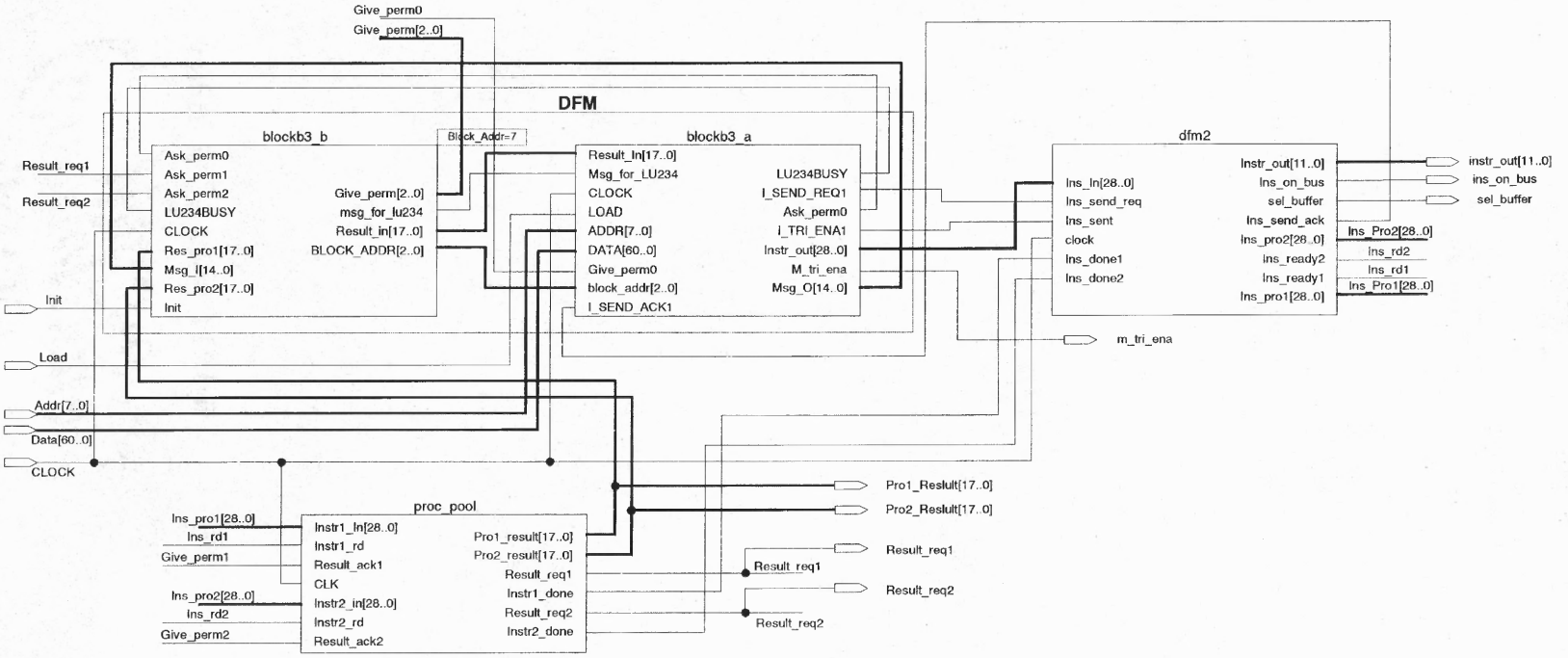


Figure 3.19 Altera produced Graphic Design File of the Full Dataflow Computer (redrawn)

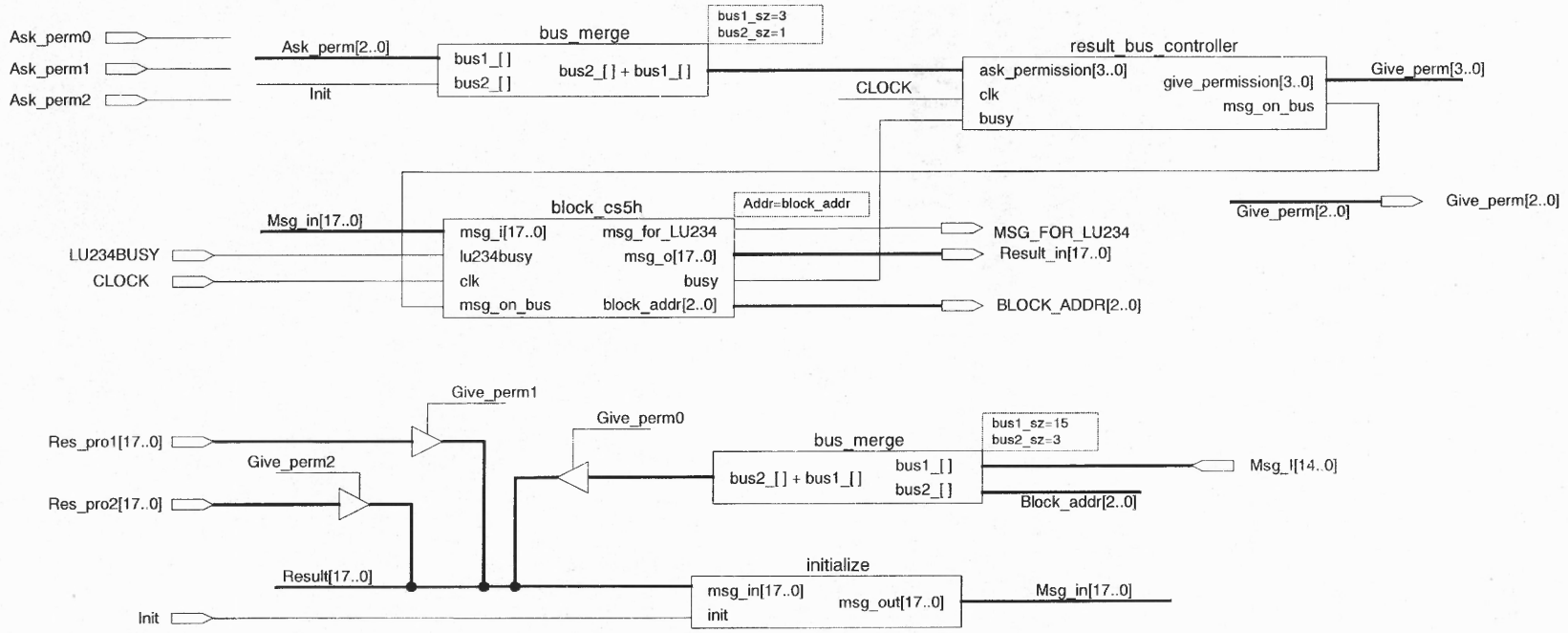


Figure 3.20 Altera produced Graphic Design File of Blockb3_b (redrawn)

PROC_POOL: The proc_pool is the processor pool that executes the instructions sent by the DFM. The proc_pool contains two main components, two *processors.* The hierarchy of the module is shown in Appendix A.15 and its layout is shown in Figure 3.23.

proc: The processors in the proc_pool are simple execution units capable of arithmetic operations; ADD, SUB, and MUL and logical operations; CEQ, CNE, CGE, CLE, CGT and CLT. Division is not implemented currently because there was a problem with the parameterized divide function (lpm_divide) provided by Altera. The adder is 6-bits wide, while the subtractor uses all 7-bits of the two operands. The multiplier uses a 4-bit multiplicand and a 3-bit multiplier; the 4-bits used come from the first operand, while the 3-bits come from the second operand. Each of the logic operations is performed on all 7-bits of both operands. All operations are of the form "OPD1 operation OPD2", and the results are padded to have a total of 7-bits. The proc is also responsible for sending the computed result back to DFM. The hierarchy of proc and its implementation is presented in Appendix A.16.

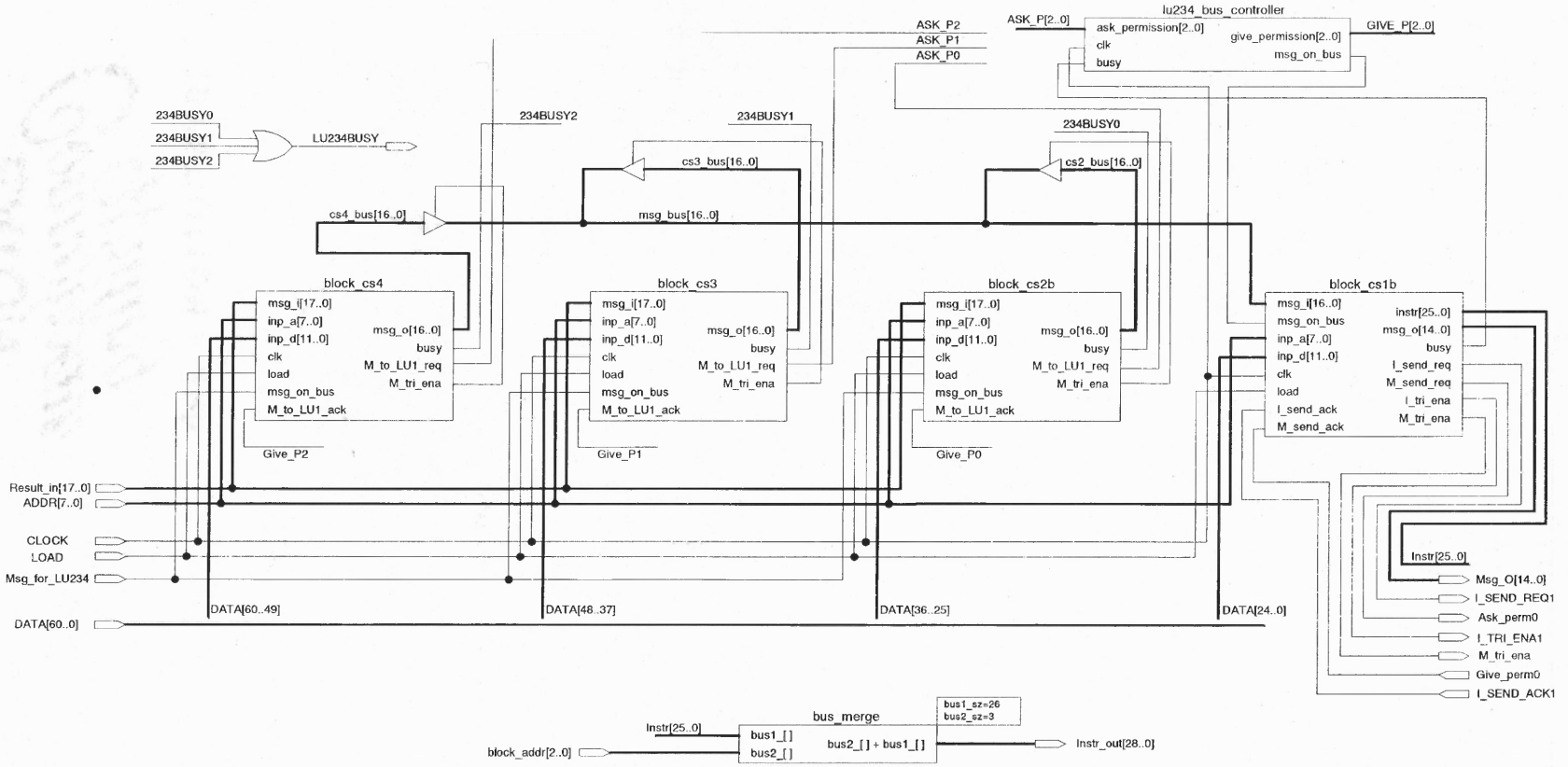


Figure 3.21 Altera produced Graphic Design File of Blockb3_a (redrawn)

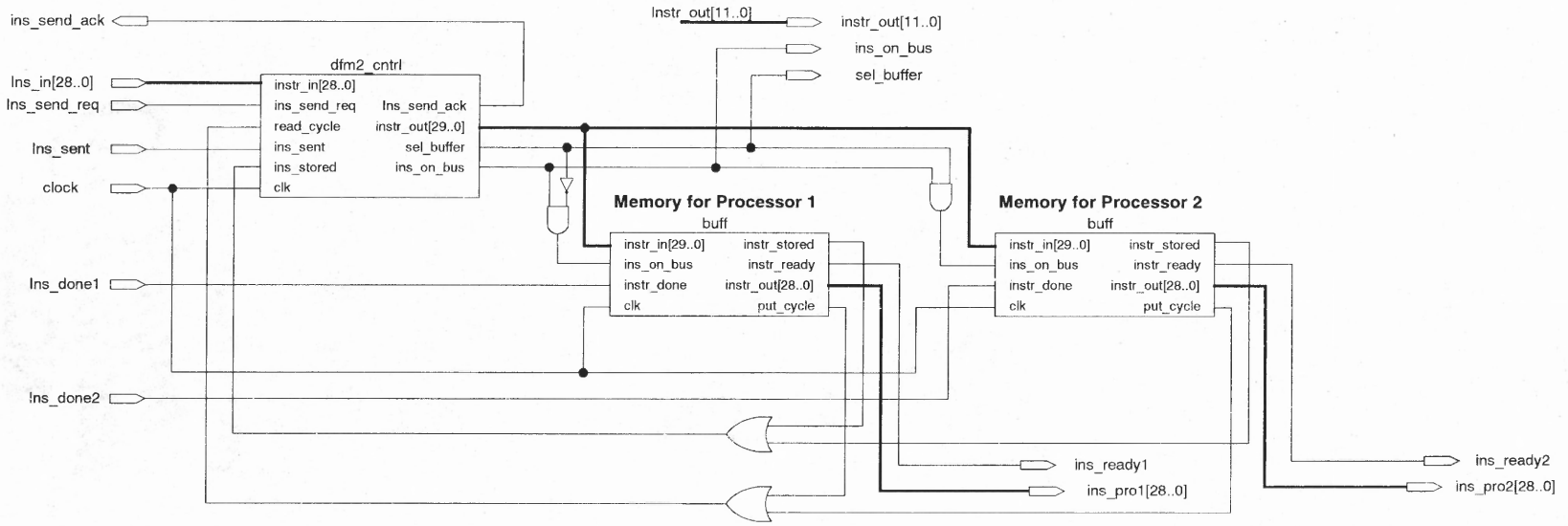


Figure 3.22 Altera produced Graphic Design File of dfm2 (redrawn)

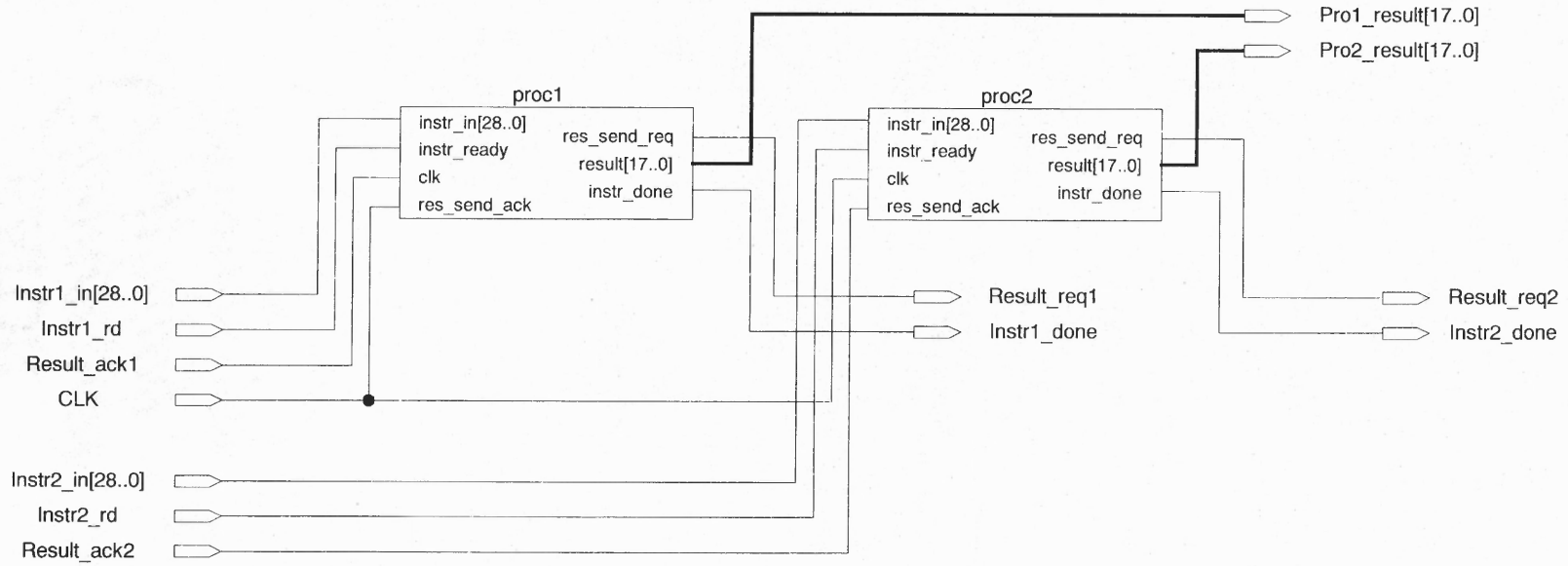


Figure 3.23 Altera produced Graphic Design File of Proc_Pool (redrawn)

SUB-COMPONENTS:

Most of the sub-components presented are *parameterized* functions. This is a feature allowed by Altera's design language. A parameterized function is a function that accepts parameters; it is method to generalize functions. For example, for some design two counters are needed, one that has maximum count of seven, another that counts till thirty-two and then resets. One-way to accomplish this is to write two counter modules, one that counts till seven and another which counts till thirty-two, and then include the two in the design. A more convenient and powerful way is to write one general-purpose counter that accepts a 'count' parameter. Now to implement the two counters, the general-purpose counter is declared twice in the design file, one is declared with a count parameter set to seven and the other is declared with a count parameter set to thirty-two [22]. Each sub-component used is presented below in alphabetical order.

1count: 1count is a 1-bit counter with a clear line, used particularly for signaling. The implementation is provided in Appendix A.17.

21mux: 21mux is an Altera provided 2-to-1 MUX.

4count: 4count is an Altera provided 4-bit counter.

bus_merge: Bus_merge is a sub-component that allows the merging of multiple lines together to form a bus, or merge two smaller buses to form a larger bus. The implementation is presented in Appendix A.18.

counter: Counter is a parameterized counter with enable and synchronous clear lines. The parameter provided is the bit-width 'sz' of the counter. The implementation is provided in Appendix A.19.

delaytimer: Delaytimer is parameterized timer with enable and synchronous clear lines, which takes in a 'delay' parameter. When the delaytimer is active, it continuously sends out a pulses spaced 'delay' cycles apart. The implementation is presented in Appendix A.20.

lpm_add_sub: lpm_add_sub is a parameterized Adder/Subtractor provided by Altera.

lpm_compare: lpm_compare is a parameterized comparator provided by Altera.

lpm_counter: lpm_counter is a parameterized counter provided by Altera.

lpm_mult: lpm_mult is a parameterized multiplier provided by Altera.

lpm_ram_dp: lpm_ram_dp is a parameterized function provided by Altera to implement dual-port memory. All the lpm_functions provided by Altera are extremely powerful functions with a lot of flexibility.

myclock2: Myclock2 is a parameterized clock, which takes two parameters 'LO' and 'HI' and outputs a clock signal that has a high time of 'HI' cycles and low time of 'LO' cycles. The implementation is presented in Appendix A.21.

mydffe: mydffe is edge triggered flipflop with enable and clear lines. The clear is an active high clear, instead of the active low clear DFFE provided by Altera. The implementation is presented in Appendix A.22.

mylatch: Mylatch is level triggered flipflop, which sets itself to '0' when power is applied to it. Mylatch was implemented instead of using the level triggered latch provided by Altera because when powered up, the Altera provided latch would have

an undefined value in it, until a value is latched into it. The implementation is presented in Appendix A.23.

pulsegen: pulsegen is a parameterized function which takes two parameters, 'PW' and 'DEL'. The pulsegen when activated by a trigger-signal, sends out a pulse 'PW' cycles wide, 'DEL' cycles after the trigger-signal came in. The implementation is presented in Appendix A.24.

queue: The queue is a circular queue used to implement the memory banks in the IQ and the QB. A module using the *queue* communicates with it using two signals in particular, they are the *empty* and the *full* signal, which indicate the status of the queue. The hierarchy and the text implementation is presented in Appendix A.25.

stoptimer: The stoptimer is a parameterized function which takes one parameter 'DEL'. When activated by a trigger-signal, the stoptimer sends out a high signal after a delay of 'DEL' cycles. The implementation is presented in Appendix A.26.

stoptimer2: This stoptimer is a modified version of the above stoptimer. They both perform the same task; only the implementation is a little different, which is presented in Appendix A.27.

toggle: Like the *1count* presented earlier, the toggle is also a 1-bit counter, except the implementation of toggle is different and is presented in Appendix A.28.

trans_detector: The trans_detector is a parameterized function which takes two parameters 'FV' and 'DEL'. This module is used to detect transitions on a line. When a line on which transitions have to be detected is connected to the trans_detector, FV determines whether the transition to be detected is high to low, or low to high. The

'DEL' value takes on two forms; if it is greater than zero, then the detector stops detecting transitions on the line (after one is detected) until 'DEL' cycles have expired. If 'DEL' is zero, then the detector stops detecting transitions on the line (after one is detected) until an external signal (not on the line on which transitions are being detected) reactivates the detector. The implementation is presented in appendix A.29.

3.4 Remarks

The design of the dataflow computer presented is simple, because the goal of this project was not to create a large, powerful computer, but to examine the feasibility of implementing a dataflow computer using the proposed 'dataflow memory' architecture. If categorized as per the tree presented in Figure 2.7, this design would be considered a *static* data flow machine using a *packet* based communication scheme. Higher primitives such as *code-copying* and *tagged tokens* are not implemented in this design, hence procedure invocation and indirect memory addressing is not currently possible on this machine. However, it is not impossible to add these features into the current architecture by making some modifications in the design to accommodate them.

There is one feature of this architecture that eliminates a problem faced by past dataflow designs, and that is the problem of *Data Fan Out*. Data Fan Out of an instruction 'A' is the number of instructions that need result from the execution of instruction 'A.' In past designs, the data fan out of an instruction was limited, usually to two. That means that after execution an instruction can send out the result to at most two or three destinations. So programs to be run on such machines had to be written adhering to this restriction. Of course, this was a major drawback and different schemes were

developed to overcome this restriction. Two ways that were devised to nullify this restriction were through hardware, as implemented in the epsilon dataflow processor [11], which used a *repeat* hardware unit that circulated the result value using a tagging scheme. The second method is using specialized instructions, which hold addresses of additional instructions (beyond the allowed limit) that need the result. The address of the special instruction is on the list addresses that executing instruction needs to send the result to. When the special instruction receives the result, it forwards this result to its list of destination addresses. This method can be chained so that the result can be sent to a large number of destinations.

However, no such means need to be employed in this architecture, since no instruction maintains a list of destination addresses to send the result to; instead each instruction has the addresses of the sources in its two operands. When a result packet is sent out, all instructions pick up the packet and compare the source addresses it has with the originating address in the result packet. All the instructions that make a positive match absorb the result. This scheme completely eliminates the data fan out problem.

The beauty of the design is that increasing the number of processors does not increase complexity drastically, because there is no need for synchronization between the processors. Since the order of execution is not important, each processor simply executes instructions as fast it can and sends the result back to the DFM. Even the cost of implementing is linear; adding N processors costs N times the cost of adding one processor.

Removing the processor from the dataflow tasks offers the advantage of simpler processors, and the ability to easily replace simpler execution units with powerful ones

having a compatible interface. Besides, the dataflow in DFM is performed using only the flag bits, thus offering opcode independence. Hence, a compatible (similar flag bits) language can be used to write the same program or new opcodes can be added to the existing language without affecting the dataflow; of course, a compatible processor needs to be used to execute the different or new instructions.

This architecture offers a number of advantages over previous implementations of dataflow computers. The performance of this machine is examined in the next chapter, which presents simulations that were run on this machine, along with the observed results.

4 TIMINGS, SIMULATIONS, AND PERFORMANCE

4.1 Timings

This section presents the timings of all the *main components* in the design in terms of clock cycles, irrespective of the clock speed. The main components are the bus controllers, the cell sections CS1 – CS4, the result queue also called CS5, the IQ, and the processor pool.

result_bus_controller: To reiterate the result_bus_controller controls the arbitration of results coming into the DFM. It is a priority-based controller, with the highest priority given to LU1; Processor1 and Processor2 bring up the rear respectively. Since the controller is priority based, the timing for each unit asking permission from the controller to send result is different. Though the result_bus_controller is priority based, the priority is only applicable when multiple units are asking permission to send data simultaneously; the priority has no effect if the controller is interrupted by a unit with a higher priority while it is servicing a unit of lower priority. It takes 6-cycles to service a request and on the seventh cycle, the controller looks for new requests. Based on this and the priority, the best and worst timing for different units requesting permission to send results are presented in Table 4.1.

lu234_bus_controller: The lu234_bus_controller is exactly like the result_bus_controller. It is a priority-based controller with the highest priority given to LU2; LU3 and LU4 bring up the rear respectively. The controller takes 7-cycles to service a request for sending a matched value from any CS. On the eighth cycle, the

controller accepts new requests. The best and worst timing for different units requesting permission to send matched values are presented in Table 4.2

Table 4.1 Timing of the result_bus_controller

Get Permission	Best Case (cycles)	Worst Case (cycles)
LU1	2	$6 + t_P + t_{LU5}$
Processor1	2	$6 + t_{LU1} + t_{LU5}$
Processor2	2	$6 + t_{P1} + t_{LU1} + t_{LU5}$
$t_P \rightarrow$ *time taken by the controller to process a request made by a processor (max 6)*		
$t_{LU5} \rightarrow$ *time taken by LU5 process a result sent to (5 – 11 cycles)*		
$t_{LU1} \rightarrow$ *time taken by the controller to process a request made by LU1 (max 6)*		
$t_{P1} \rightarrow$ *same as* t_P		

Table 4.2 Timing of the lu234_bus_controller

Get Permission	Best Case (cycles)	Worst Case (cycles)
CS2	2	$8 + t_{LU} + t_{LU1}$
CS3	2	$8 + t_{LU2} + t_{LU1}$
CS4	2	$8 + t_{LU2} + t_{LU3} + t_{LU1}$
$t_{LU} \rightarrow$ *time taken by the controller to process a request made by LU1 or LU2 (max 8)*		
$t_{LU1} \rightarrow$ *time taken by LU1 to process a token sent by cell sections (8 – 28 cycles)*		
$t_{LU2} \rightarrow$ *same as* t_{LU}		
$t_{LU3} \rightarrow$ *same as* t_{LU}		

CS2 – CS4: The three cell sections as explained earlier, simultaneously checks if there is a match between the OA of the incoming result and any of the operand/clause address fields of the 255 instructions that each of the cell sections are managing. A CS

accepts no new results, when it, or another CS is busy making matches, i.e. all cell sections must finish making matches, before a new result is accepted. Table 4.3 presents the best and the worst timing for a CS to make a single compare, and the time it takes if a match is made.

Table 4.3 Timing of CS2 – CS4

Matching (with one cell)	Best Case (cycles)	Worst Case (cycles)
No Match (CS2, CS3, CS4)	5	-
Match (CS2)	14	$20 + t_{LU1}$
Match (CS3)	14	$20 + t_{LU2} + t_{LU1}$
Match (CS4)	14	$20 + t_{LU3} + t_{LU2} + t_{LU1}$
$t_{LU1} \rightarrow$ *time taken by LU1 to process a token sent by cell sections (8 – 28 cycles)*		
$t_{LU2} \rightarrow$ *time taken processing a LU2 token sent to LU1 (max20)*		
$t_{LU3} \rightarrow$ *time taken processing a LU3 token sent to LU1 (max20)*		

CS1: CS1 takes the values sent to it by the other cell sections and either deposits it in the appropriate location, or deposits and dispatches either an executable to the IQ or a message to the result_bus_controller. The timing of these activities is presented in Table 4.4.

CS5: If LU234 are not busy, the CS5 takes results sent by the result_bus_controller and either sends it directly to LU234, else queues the result. If there are queued results in the buffer, then the new incoming results are queued even if LU234 are not busy, dispatching the queued results first. Queuing and dequeuing operations are independent of each other and are carried out in parallel. Hence dequeuing does not

affect the amount of time it takes to queue incoming results. The timing of these tasks performed by CS5 is presented in Table 4.5.

Table 4.4 Timing of CS1

Token Processing	Best Case (cycles)	Worst Case (cycles)
Depositing a token	8	-
Sending an Executable	8	$16 + t_{IQ}$
Sending a Message	8	$10 + t_{RBUS}$
$t_{IQ} \rightarrow$ *time taken by IQ to process (queue) a previously sent executable (max 12)*		
$t_{RBUS} \rightarrow$ *time taken to get permission from result_bus_controller (refer to Table 4.1)*		

Table 4.5 Timing of CS5

Result Processing	Best Case (cycles)	Worst Case (cycles)
Dispatching results to LU234 (LU234 not busy)	3	-
Queuing results	6	-
Dequeuing results	5	$11 + t_{OBUS}$
$t_{RBUS} \rightarrow$ *time taken to get permission from lu234_bus_controller (refer to Table 4.2)*		

IQ: The IQ is responsible for queuing executables streaming from the DFM to be executed by the processor pool. Only two timings are of concern in the IQ, the time taken to store an executable coming in and the time taken to remove an executable to be sent to the processor pool. Table 4.6 presents the best and the worst cases for the two timings.

Processor Pool: The timing of concern in the processor pool is the time it takes to execute an executable and the time it takes to send the result back to DFM. Table 4.7 presents the worst and best cases of these times.

Table 4.6 Timing of IQ

Instruction Operation	Best Case (cycles)	Worst Case (cycles)
Queuing Instruction	4	12
Dequeuing	4	12

Table 4.7 Timing of Processor Pool

Processor Operation	Best Case (cycles)	Worst Case (cycles)
Execution	8	-
Sending the Result	1	$1 + t_{RBUS}$
$t_{RBUS} \rightarrow$ *time taken to get permission from result_bus_controller (refer to Table 4.1)*		

Now that the timings of the different main components have been presented, it will be easier to understand the execution times of the three programs that were run on this dataflow machine.

4.2 Simulations

Three programs were run on this machine. They were designed, partially to test the performance of the machine, but more to check if the computer handled the kind of constructs that are usually found in a program, such as reentrant code, conditionals, and arithmetic operations. Every program was simulated at a speed of 100Mhz, i.e. all the

units DFM, IQ and Processor Pool were clocked at a frequency of 100Mhz. This was the fastest speed that the simulation could run at, without losing stability of the design.

4.2.1 Program 1

The first program run on this machine is a slight variation of the example presented in Section 3.3.5, and is presented in Table 4.8; the equivalent dataflow program is presented in Table 4.9. The shaded portions in Table 4.9 represent the cell sections; from the left they are CS4, CS3, CS2 and CS1.

Table 4.8 First Program Run on the Dataflow Machine in High Level Language

Calculation of a function F1 and F2:	
	Results:
x = 2 * 9 - 15	
If x ≠ 0 then	
F1 = 0	
F2 = 1	F1 = 0 + 3 * (1 + 2)
For i = 1 to x	F1 = 9 + 3 * (2 + 2)
F1 = F1 + x * (i + 2)	F1 = 21 + 3 * (3 + 2)
End For	F1 = 36**(final answer)**
While x > 1	
F2 = F2 * x	F2 = 1 * 3
x = x - 1	F2 = 3 * 2
Endwhile	F2 = 6**(final answer)**
Endif	

Program 1 computes two functions, F1 and F2, it contains two loops enclosed within a conditional *if* statement. The FOR loop is built with inter-iteration dependencies, on the other hand the WHILE loop is setup to have no inter-iteration dependencies. Since the dataflow machine is not setup to accept any direct input from the user, the program is setup to be self-contained, i.e. no values are needed from outside the program. In the

dataflow version of the program, the instruction at location 1 needs a clause from 7FF, which is the address 2047 (binary, 111-block 11111111-memory). As mentioned in section 3.3.6, this is the seed clause sent by the sub-component *initialize* when a trigger is sent to it. The fact that the seed comes from 7FF, has nothing to do with the decision to load the program in locations 701H – 70FH. These addresses simply signifies that the program was loaded in block 7, from addresses 01H – 0FH.

Table 4.9 First Program Run on the Dataflow Machine in Dataflow Language

ADDRESS	OPERAND2 ADDR. (D2A)	OPERAND2 REQD. (D2R)	OPERAND1 ADDR. (D1A)	OPERAND1 REQD. (D1R)	CLAUSE ADDR. (CA)	CLAUSE REQUIRED (CR)	OPERAND2 (OPD2)	OPERAND1 (OPD1)	OPCODE (OP)	LOOP (LP)	OPERAND2 REUSE (D2U)	OPERAND1 REUSE (DIU)	OPRD2 OBTAINED (D2O)	OPRD1 OBTAINED (D1O)	CLAUSE ANSWER (CAN)	
•																
•	WAIT FOR CLAUSE FORM (7FF) FOR THIS PROGRAM TO START															
701	0	0	0	0	7FF	1	9	0	MUL	0	0	0	1	1	0	Calculate **X**
702	0	0	701	1	0	0	15	0	SUB	0	0	0	1	0	1	
703	0	0	702	1	0	0	0	0	CNE	0	0	0	1	0	1	If stmt.
704	0	0	0	0	703	1	1	0	ADD	0	0	0	1	1	0	Initialize **i**
705	70B	1	704	1	703	1	0	0	SP	1	0	0	0	0	0	For Loop **Start**
706	702	1	705	1	703	1	0	0	CLE	2	1	0	0	0	0	
707	700	0	705	1	706	1	2	0	ADD	0	1	0	1	0	0	Calculate **F1**
708	702	1	707	1	706	1	0	0	MUL	0	1	0	0	0	0	
709	709	1	708	1	706	1	0	0	ADD	0	0	0	1	0	0	
70A	709	1	705	1	706	1	0	0	LK	3	0	0	0	0	0	Lock instr
70B	0	0	70A	1	703	1	1	0	ADD	0	1	0	1	0	0	Increment **i**
70C	70F	1	702	1	703	1	0	0	SP	1	0	0	0	0	0	While loop
70D	0	0	70C	1	703	1	1	0	CGT	2	1	0	1	0	0	
70E	70E	1	70C	1	70D	1	1	0	MUL	0	0	0	1	0	0	Calculate **F2**
70F	0	0	70C	1	70D	1	1	0	SUB	0	1	0	1	0	0	
•																

The simulation results for this program are presented in Section 4.3.1 along with the simulation run shown in Appendix B.2.

4.2.2 Program 2

The second program run on this machine is shown in Table 4.10, and the equivalent dataflow program is presented in Table 4.11. This program was setup such that there was higher level of concurrency. It consists of three arrays, A, B and X, each of them having 12 elements. In the pseudo-code presented in Table 4.10, array A = array X * 7 and array B = array A + 10. Since there are no dependencies between any iterations of the first FOR loop, the 12 elements in array A can be updated simultaneously, giving an effective concurrency of 12. But there are dependencies between the first and the second FOR loop, so the second FOR loop cannot execute until the first has finished executing. When the first FOR loop finishes its execution, all the 12 iterations of the second FOR loop can also execute concurrently, since there are no inter-iteration dependencies.

The amount of code for the equivalent dataflow program looks bloated because as mentioned before in Section 1.2.2, in a dataflow computer both the instruction and the data it works on reside in the same memory location. While in a Von Neumann architecture, the instruction and the data it operates on reside in separate memory locations. So in fact the two styles of programs use about the same amount of memory. Secondly, this architecture does not have indexed addressing mode, hence the entire loop is opened up and each element in the arrays is worked on independently. Though no indexed addressing seems like a problem here, it actually helps in achieving maximum concurrency without adding additional features that would be needed if indexed addressing was part of the design.

The first set of LK instructions function as a placeholder for the array X, the values of which are sent to array A (the set of MUL instructions) when the seed is introduced. The MUL instructions after execution pass the results to array B (the set of ADD instructions). The ADD instructions in turn pass the results of their execution to the set of LK instructions, which function as placeholders to store the final results.

The simulation results for this program are presented in Section 4.3.2. No simulation run is presented for this program run because the results are very similar to those obtained from the first program run. These similarities in spite of a higher level of concurrency are also discussed in Section 4.3.2.

Table 4.10 Second Program Run on the Dataflow Machine in High Level Language

Manipulating values in an array:	
	Comments:
	Three arrays A,B & X
For i = 1 to 12	
A[i] = X[i] * 7	Array A = Array X*7
End For	
For i = 1 to 12	Array B = Array A+10
B[i] = A[i] + 10	
End For	

Table 4.11 Second Program Run on the Dataflow Machine in Dataflow Language

ADDRESS	OPERAND2 ADDR. (D2A)	OPERAND2 REQD. (D2R)	OPERAND1 ADDR. (D1A)	OPERAND1 REQD. (D1R)	CLAUSE ADDR. (CA)	CLAUSE REQUIRED (CR)	OPERAND2 (OPD2)	OPERAND1 (OPD1)	OPCODE (OP)	LOOP (LP)	OPERAND2 REUSE (D2U)	OPERAND1 REUSE (D1U)	OPRD2 OBTAINED (D2O)	OPRD1 OBTAINED (D1O)	CLAUSE ANSWER (CAN)	
•																
•	WAIT FOR CLAUSE FORM (7FF) FOR THIS PROGRAM TO START															
701	0	0	0	0	7FF	1	0	2	LK	3	0	0	1	1	0	
702	0	0	0	0	7FF	1	0	3	LK	3	0	0	1	1	0	
703	0	0	0	0	7FF	1	0	4	LK	3	0	0	1	1	0	
704	0	0	0	0	7FF	1	0	5	LK	3	0	0	1	1	0	Place holder for
705	0	0	0	0	7FF	1	0	6	LK	3	0	0	1	1	0	the array. The
706	0	0	0	0	7FF	1	0	7	LK	3	0	0	1	1	0	array is waiting
707	0	0	0	0	7FF	1	0	8	LK	3	0	0	1	1	0	for the seed to
708	0	0	0	0	7FF	1	0	7	LK	3	0	0	1	1	0	come to send
709	0	0	0	0	7FF	1	0	6	LK	3	0	0	1	1	0	out the values
70A	0	0	0	0	7FF	1	0	5	LK	3	0	0	1	1	0	
70B	0	0	0	0	7FF	1	0	4	LK	3	0	0	1	1	0	
70C	0	0	0	0	7FF	1	0	3	LK	3	0	0	1	1	0	
70D	0	0	701	1	0	0	7	0	MUL	0	0	0	1	0	1	
70E	0	0	702	1	0	0	7	0	MUL	0	0	0	1	0	1	
70F	0	0	703	1	0	0	7	0	MUL	0	0	0	1	0	1	
710	0	0	704	1	0	0	7	0	MUL	0	0	0	1	0	1	
711	0	0	705	1	0	0	7	0	MUL	0	0	0	1	0	1	
712	0	0	706	1	0	0	7	0	MUL	0	0	0	1	0	1	Muliplying the
713	0	0	707	1	0	0	7	0	MUL	0	0	0	1	0	1	contents of the
714	0	0	708	1	0	0	7	0	MUL	0	0	0	1	0	1	array by 7.
715	0	0	709	1	0	0	7	0	MUL	0	0	0	1	0	1	
716	0	0	70A	1	0	0	7	0	MUL	0	0	0	1	0	1	
717	0	0	70B	1	0	0	7	0	MUL	0	0	0	1	0	1	
718	0	0	70C	1	0	0	7	0	MUL	0	0	0	1	0	1	
719	0	0	70D	1	0	0	A	0	ADD	0	0	0	1	0	1	Adding 10 to
71A	0	0	70E	1	0	0	A	0	ADD	0	0	0	1	0	1	the contents of
71B	0	0	70F	1	0	0	A	0	ADD	0	0	0	1	0	1	the array after
71C	0	0	710	1	0	0	A	0	ADD	0	0	0	1	0	1	they are
71D	0	0	711	1	0	0	A	0	ADD	0	0	0	1	0	1	multiplied by 7
71E	0	0	712	1	0	0	A	0	ADD	0	0	0	1	0	1	
71F	0	0	713	1	0	0	A	0	ADD	0	0	0	1	0	1	
720	0	0	714	1	0	0	A	0	ADD	0	0	0	1	0	1	

721	0	0	715	1	0	0	A	0	ADD	0	0	0	1	0	1	
722	0	0	716	1	0	0	A	0	ADD	0	0	0	1	0	1	
723	0	0	717	1	0	0	A	0	ADD	0	0	0	1	0	1	
724	0	0	718	1	0	0	A	0	ADD	0	0	0	1	0	1	
725	0	0	719	1	0	0	0	0	LK	3	0	0	0	0	0	
726	0	0	71A	1	0	0	0	0	LK	3	0	0	0	0	0	
727	0	0	71B	1	0	0	0	0	LK	3	0	0	0	0	0	
728	0	0	71C	1	0	0	0	0	LK	3	0	0	0	0	0	
729	0	0	71D	1	0	0	0	0	LK	3	0	0	0	0	0	Place holder for
72A	0	0	71E	1	0	0	0	0	LK	3	0	0	0	0	0	the computed
72B	0	0	71F	1	0	0	0	0	LK	3	0	0	0	0	0	values of the
72C	0	0	720	1	0	0	0	0	LK	3	0	0	0	0	0	array.
72D	0	0	721	1	0	0	0	0	LK	3	0	0	0	0	0	
72E	0	0	722	1	0	0	0	0	LK	3	0	0	0	0	0	
72F	0	0	723	1	0	0	0	0	LK	3	0	0	0	0	0	
730	0	0	724	1	0	0	0	0	LK	3	0	0	0	0	0	
•																

4.2.3 Program 3

The last program specifically catered to the way programs executed on this machine, and was designed so that maximum concurrency would be achieved on the dataflow prototype presented. The third program is presented in Table 4.12, and the equivalent dataflow code is presented in Table 4.13.

Table 4.12 Third Program Run on the Dataflow Machine

Manipulating values in an array:	
	Comments:
	Two arrays A & X
X[1] = 2 * 2	
For i = 2 to 12	Array X[2-12] =
X[i] = X[i] * X[1]	Array X[2-12] * X[1]
End For	
For i = 1 to 12	Array A = Array A +
A[i] = A[i] + X[2]	X[2]
End For	

Table 4.13 Third Program Run on the Dataflow Machine in Dataflow Language

ADDRESS	OPERAND2 ADDR. (D2A)	OPERAND2 REQD. (D2R)	OPERAND1 ADDR. (D1A)	OPERAND1 REQD. (D1R)	CLAUSE ADDR. (CA)	CLAUSE REQUIRED (CR)	OPERAND2 (OPD2)	OPERAND1 (OPD1)	OPCODE (OP)	LOOP (LP)	OPERAND2 REUSE (D2U)	OPERAND1 REUSE (D1U)	OPRD2 OBTAINED (D2O)	OPRD1 OBTAINED (D1O)	CLAUSE ANSWER (CAN)	
•																
•	WAIT FOR CLAUSE FORM (7FF) FOR THIS PROGRAM TO START															
701	0	0	0	0	7FF	1	2	2	MUL	0	0	0	1	1	0	Wait for seed
702	0	0	701	1	0	0	2	0	MUL	0	0	0	1	0	1	
703	0	0	701	1	0	0	3	0	MUL	0	0	0	1	0	1	
704	0	0	701	1	0	0	4	0	MUL	0	0	0	1	0	1	
705	0	0	701	1	0	0	5	0	MUL	0	0	0	1	0	1	
706	0	0	701	1	0	0	6	0	MUL	0	0	0	1	0	1	Wait for OPD1
707	0	0	701	1	0	0	7	0	MUL	0	0	0	1	0	1	from location
708	0	0	701	1	0	0	8	0	MUL	0	0	0	1	0	1	701H to do a
709	0	0	701	1	0	0	9	0	MUL	0	0	0	1	0	1	multiplication
70A	0	0	701	1	0	0	A	0	MUL	0	0	0	1	0	1	with OPD2.
70B	0	0	701	1	0	0	B	0	MUL	0	0	0	1	0	1	
70C	0	0	701	1	0	0	C	0	MUL	0	0	0	1	0	1	
70D	0	0	701	1	0	0	D	0	MUL	0	0	0	1	0	1	
70E	0	0	701	1	0	0	E	0	MUL	0	0	0	1	0	1	
70F	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
710	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
711	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
712	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
713	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	Wait for OPD1
714	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	from location
715	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	702H to add 10
716	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	to the incoming
717	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	value
718	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
719	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
71A	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
71B	0	0	702	1	0	0	A	0	ADD	0	0	0	1	0	1	
•																

As explained earlier in Section 3.3.2, when a result comes into the DFM, each LU (2-4) checks the OA of the result against the 256 cells that it is responsible for. If a match is made, the result is sent over to LU1, and if all the required operands/clause is available, the instruction is fired (sent for execution). As the number of matches made by each LU (during its iteration of checking 256 cells) increases, more results are sent to LU1, which in turn causes more instructions to fire. This program takes advantage of this feature to increase concurrency.

There are two arrays X and A. X[1] is set to 2 * 2, while all the other elements of the array X are computed to, X[i] * X[1]. Array A is computed to A[i] * X[2]. There are two dependencies in this program, first, X[1] has to be computed before any of the other X[i] can be evaluated. Secondly, X[2] has to be calculated before any A[i] can be calculated. How this is an advantage to increase concurrency may not be obvious, until the dataflow code is examined.

Instructions at locations 701H – 70EH are occupied by array X, and instructions at locations 70FH – 71BH are occupied by array A. Instruction at 701H has all its operands and waits for the seed clause. All the other members of array X require the result of this instruction. When this result arrives, the instructions 702H – 70EH fire during the first iteration that the LUs do the matching. The first instruction to fire in array X is the instruction at location 702H, the result of which is needed by all the elements of array A. When the result from the execution of instruction at 702H arrives, the instructions at 70FH – 71BH fire during the second iteration the LUs do the matching. Thus only in two iterations of the LUs, the entire program is executed.

This program maybe biased to show the best performance of the machine, but the goal of the program was to show the best possible result that can be obtained from this machine. This show that by reconfiguring the hardware to remove the time wasted doing irrelevant tasks, it is possible to extract a high level of performance consistently, provided there is sufficient concurrency. The simulation results for this program are discussed in Section 4.3.3 and the simulation run is presented in Appendix B.3.

4.3 Simulation Results

4.3.1 Program 1

As seen in the simulation graph of program 1 in Appendix B.2, the whole program took about 485μsec. This includes the time to load the program, which appears in the first two pages of the simulation. The second page also shows the signaling of the 'Init' signal that sends the seed clause that starts firing instructions. Subsequent pages show a lot of activity happening in the DFM memory cells 1 – 15. Activity is also seen in the IQ memory cells and the result lines coming out of the two processors. Note that beyond the second page, the horizontal scale of the graph is shrunk to show the activities occurring in the dataflow machine. If they were shown at the same scale as the first two pages, activities occurring wouldn't be as visible, since the events occur too far apart. This is the reason the clock signal is seen as a black band.

Unrolling all the loops in the program, gives a total of 37 instructions. Excluding the time taken to load the instruction, which is about 1μsec, equates to about to about 484μsec of execution time. This gives an average execution of 12μsec per instruction.

The speed of the clock that the machine is being executed at is 100 Mhz, giving a clock-cycle of 1ns. Hence the number of cycles needed to execute one instruction is 1200!!!

There is a plausible explanation for this despicable performance, and it lies within the cell sections doing the matching, CS2, CS3 and CS4. As mentioned in Section 4.1, the time taken by a cell section to perform a match is 5 cycles. Assuming that no match is made with an incoming result, each CS would take $5 \times 255 = 1275$ cycles, which equates to 12.75 μsec of wasted time. It is wasted time specially if the block (hence the cell sections) is not full, as is the case with program 1. Only 15 of the 254 locations available in the block are being used, thus each LU is wasting timing searching for a match in the other 240 locations, where there is no match to be made.

The second problem with program 1 is that its level of concurrency is low. In case of a few matches, say 10, the CS would take $5 \times 245 + 20 \times 10 = 1425$ cycles, which equates to a total matching time of 14.25 μsec and leads to average execution time of 1.425 μsec + T μsec per instruction, which is about $142 + \tau$ cycles. The T μsec (τ cycles) is the time required to do other tasks before an instruction finishes execution, such as spending time in the IQ, Processor Pool and back in the DFM, however this time is very small (< 80 cycles until it hits CS2, CS3 or CS4 again) compared to the matching time. The most amount of concurrency observed in program 1 is three.

Ideally, more the matches are made the better overall result will be obtained. For example, a block that is full and all the instructions in the block need the same result would take $20 \times 254 = 5080$ cycles, equating to a total matching time of 50.8 μsec and

leading to an average execution time of 0.2μsec + T μsec per instruction, which is about 20 + τ cycles.

4.3.2 Program 2

The total execution time for Program 2 was about 314μsec, which excludes the time needed to load the program. There were a total of forty-eight instructions in this program, which equates to an execution time of about 6μsec per instructions or 600 cycles. This is an improvement over the execution time of Program 1, but yet quite poor. The performance problem can again be attributed to the time wasted by the LUs checking unoccupied cells, and those times when they make no matches.

Though there is plenty of concurrency, they do not exploit the architecture of this dataflow machine. The concurrency of this program can be separated into four groups, the LK group of instructions, the MUL group of instructions, the ADD group of instructions, and the last LK group instructions. Each group is dependent on the execution of the previous group, but no dependencies exist within a group. So after the first groups of twelve LK instructions execute concurrently, the MUL group of twelve instructions (that depend on results from the first LK group) can execute concurrently. This is followed by the concurrent execution of the ADD group of twelve instructions that depends on results from the MUL group. Finally the LK group of twelve instructions executes upon receiving the results from ADD group. Thus at any given time there are twelve possible instructions that can concurrently execute.

The problem arises because there is a one-to-one correspondence between the instructions between each group; i.e. result from instruction no. x in the LK group is

needed by instruction no. x in the MUL group, in turn the result from the instruction no. x in the MUL group is needed by instruction no. x in the ADD group, whose execution then feeds the result to instruction no. x in the LK group. So when a result from any group arrives, it is needed by only one instruction, so each LU wastes time checking 253 cells for a match, amounting to a large execution time for each instruction. The only time that more than one match is made by each LU is when the initial seed needed by the first group of LK instructions comes in. It may be noticed that the explanation presented should actually give an execution time worse than the execution time obtained for the first program, since, except for the first matching iteration of the LUs, where multiple LK instructions are fired (when seed is obtained), all subsequent matching iterations made by the LUs result in the firing of only one instruction. This discrepancy arises because the execution time for an LK instruction is much shorter than then other instructions that are sent outside the DFM (since LK instructions are not actually sent for execution).

These results show that a block level implementation of the DFM, where each LU blindly checks the cells for a match is not a conducive way to implement this architecture. Alternatives to this method are presented in Chapter 5.

4.3.3 Program 3

The total execution time for Program 3 was about 31μsec, which excludes the time needed to load the program. There were a total of twenty-seven instructions in this program, which equates to an execution time of about 1.15μsec per instructions or 115 cycles. This is an incredible improvement over the execution time of Program 1 and 2. The performance enhancement can be attributed to program structure that was discussed in Section 4.2.3. Even though time is wasted by the LUs checking unoccupied cells, a

one-to-many correspondence exists, which allows a high degree of concurrency for this architecture.

The one-to-many correspondence exists because multiple instructions in the program need the result from the execution of a single instruction. All elements of array X, except the first, need the result from the execution of the instruction at location 701H. Secondly all the elements in array A, need the result from the execution of the instruction located at 702H. Thus in two iterations of the LUs, all the instructions are fired resulting in higher overall performance.

This proves that by reconfiguring the hardware to eliminate the time wasted in checking empty cells, and by using a program with a high degree of concurrency, it is possible to extract an acceptable amount of performance from this architecture. These possible enhancements to improve performance are presented in the next section.

5 ANALYSIS AND ENHANCEMENTS

Observing the performance of the three programs, it may be deduced that the entire performance bottleneck of an executing program lies in CS2, CS3 and CS4. About 90% of the execution time needed by an instruction is spent in these three units and most of that time is wasted time, if a block is not adequately full. It is obvious that, to reduce execution time, the time spent matching has to be reduced. Three enhancements are presented, each one requiring more change in the implementation (not design) then the previous.

5.1 Enhancement I

The first enhancement is based on limiting the matching to only those cells in a block that have instructions. This can be accomplished with considerable ease, by two means, either have a common register for CS2, CS3 and CS4 that informs each corresponding LU the number of locations in a block that are occupied, and hence matching will performed on only those locations.

The second option is embedding the presence of the instruction in the instruction itself. Since LU2, LU3 and LU4 operate independently and in parallel, each CS needs to be provided with this information. The presence of an instruction maybe indicated by adding an additional bit to each part of an instruction appearing in every CS. Hence each cell section will have three parts an *operand/clause address, operand/clause required bit* and *instruction bit.* The 'instruction' bit will let the LU doing the matching know when to stop matching and reset itself.

In such a setup, instructions in a block would need to occupy contiguous locations, because the LU would reset itself after matching the number of instructions specified by the register, or the first time the LU hits an instruction whose 'instruction' bit is zero. Though contiguity is important in such a setup, order of the instructions is not.

5.2 Enhancement II

The second enhancement is based on a dual headed attack on reducing execution time. It increases the concurrency of matching by reducing the block size, and decreases the time wasted matching unoccupied locations by using one of the methods mentioned in the previous section.

Reducing the block size from the current 256 cells to a small number, like 16 or 32 cells, would give 16 and 8 blocks respectively, thus increasing the concurrency of matching to 16 and 8 times respectively. This of course increases the complexity of the logic (since there would more LUs/blocks doing the work simultaneously) and introduces some extra communication overheads between the DFM - IQ, and DFM - DFM (since there are more LU1s that want to communicate with the IQ and with LU5s).

If in addition to implementing a smaller block, one of the methods presented in the previous section is used to reduce time wasted in matching empty cells, considerable performance improvement may be achieved.

5.3 Enhancement III

The last enhancement is the most complex of all, it is based on implementing the dataflow computer as per the theoretical approach presented in Section 3. Maximum matching concurrency is achieved and no time is wasted matching empty cells, even

though matching is performed on empty cells. Matching empty cells does not waste time because the time spent in matching these cells is not in addition to the time spent in matching non-empty cells, all the cells are matched simultaneously. The PE doing this work needs to be a simple structure to reduce complexity.

The structure of a possible implementation of such a cell with a PE is presented in Figure 5.1. Notice that the CS2, CS3 and CS4 don't have the operand/clause required bits; the addresses in the field automatically indicate if the operand/clause is required. Since address zero is not used, it can be used to indicate if the operand/clause is required.

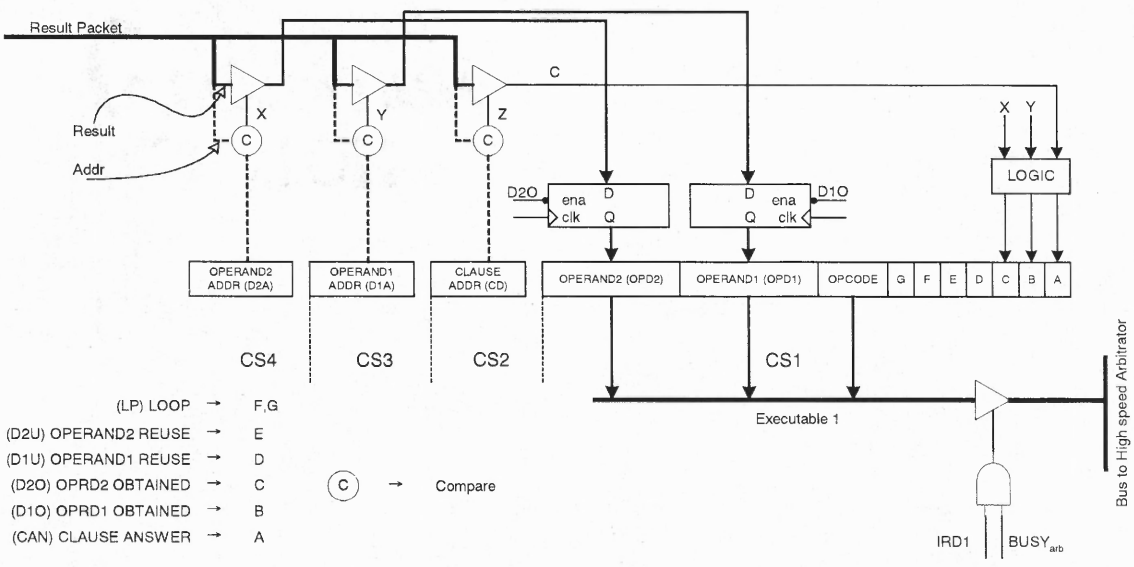


Figure 5.1 Possible Schematic of an Intelligent Cell

An incoming result packet is split into the result and OA. The result goes into a tri-state and the OA into a comparator of each CS. The OA is compared to operand/clause address in each cell and forwarded if a match is made. The logic shown in the Figure 5.2 configures the flag bits and sends the executable if appropriate flag bits are set.

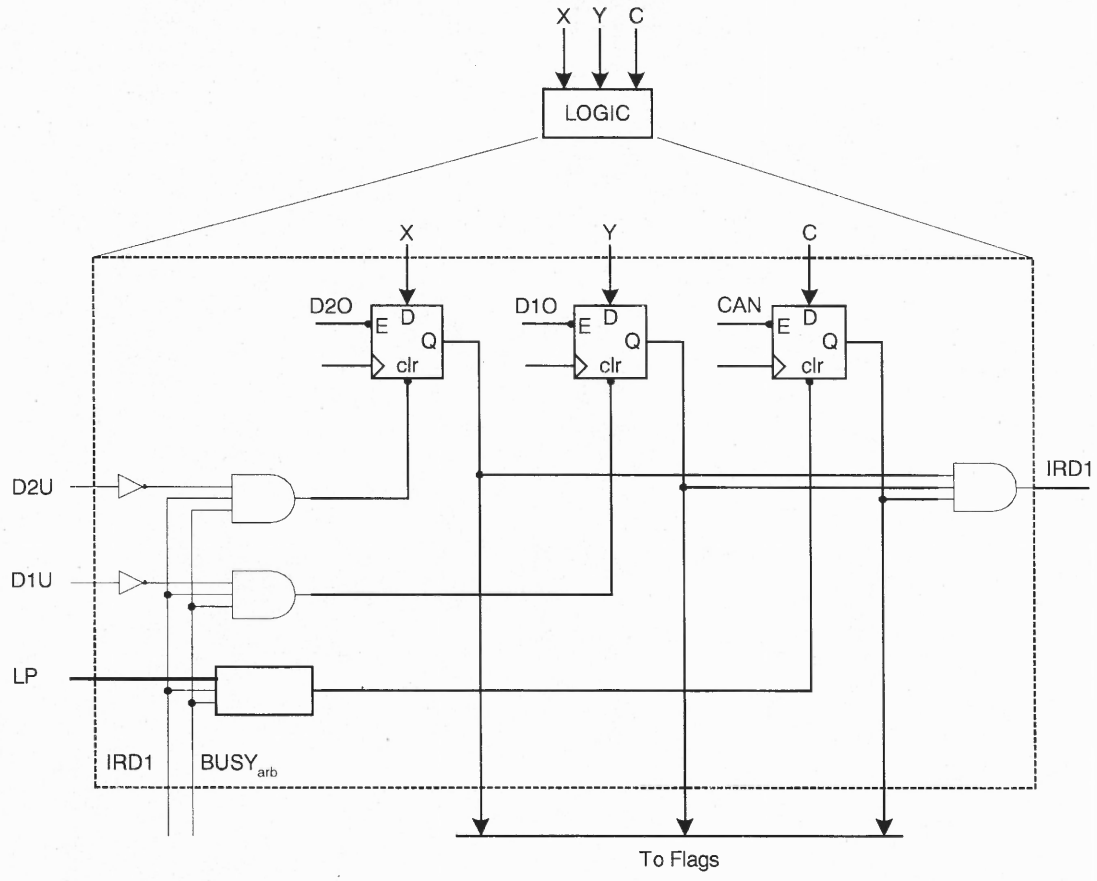


Figure 5.2 Logic used to configure Flag Bits

The concept of block takes a new meaning in this design; it is the group of cells that communicate with a high-speed arbitrator that queues (2 – 3 cycles) outgoing executables into the IQ. The concept of the IQ and Processor Pool remain unchanged. A possible high-level implementation of the DFM and a block is presented in Figure 5.3.

It may be a concern that since results from a block are sequentially queued on to a single bus, there may be performance degradation. It is a possibility, but unlikely to make a significant impact, since the number of cells allocated in a block is low, besides the arbitrator, if very fast, should be able to service all the requests very quickly. The idea

behind this design is to make the DFM pseudo-synchronous, i.e. mix both synchronous and asynchronous activities and minimize handshaking in the asynchronous activities.

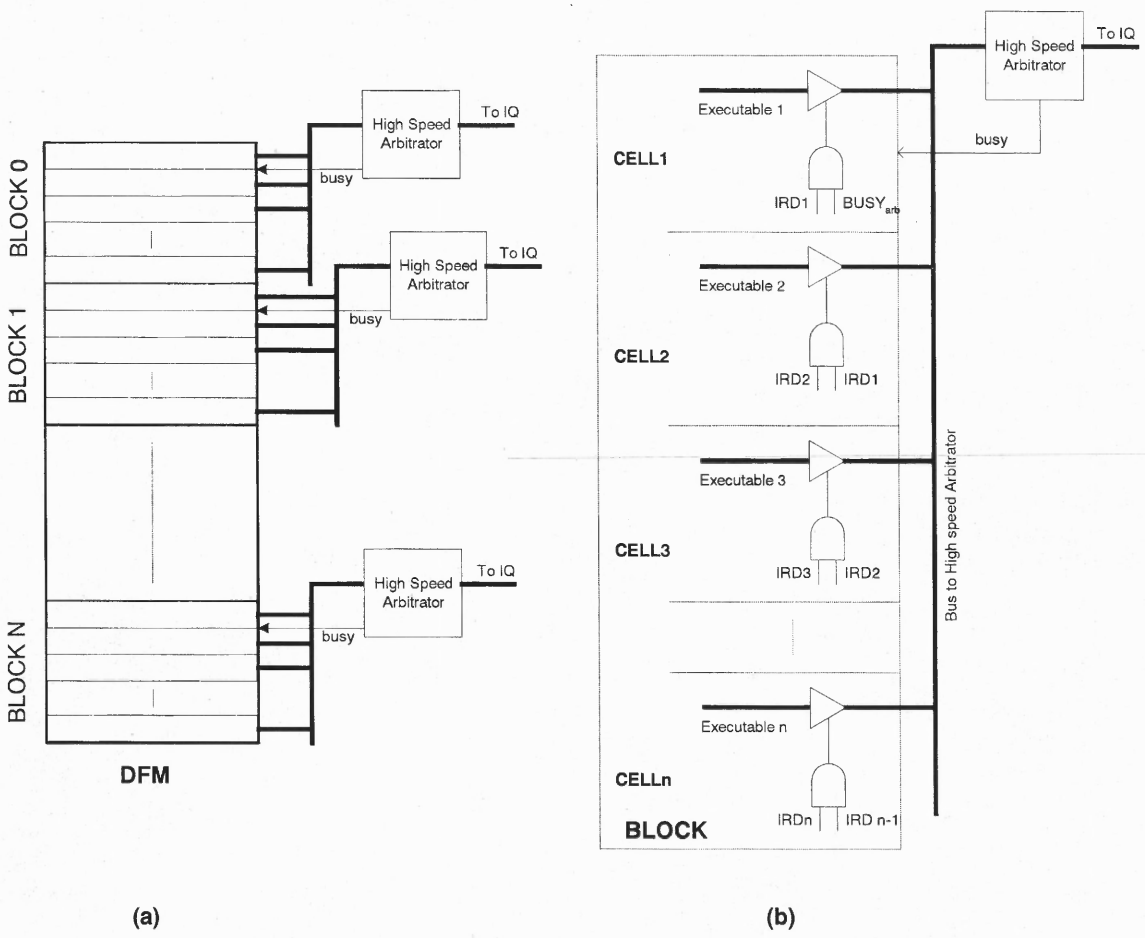


Figure 5.3 (a) High-Level implementation of DFM (b) Block Implementation

Making this new design needs a complete rework of the DFM design. Except for CS5, which queues incoming results, all the other cell sections need to be re-implemented. The processor pool and IQ can stay relatively unchanged, but a high-speed arbitrator needs to be implemented.

6 CONCLUSIONS

At the beginning of this thesis, the advantages of a dataflow computer were presented along with all the issues that a designer has to resolve to successfully implement one. This thesis presented the design and the prototype of a dataflow computer, addressing the issues of implementing a one at the memory level by using intelligent-memory, which was called Dataflow Memory.

The concept of intelligent-memory is very powerful, showing that the role of memory sub-systems as docile units is short-lived. It is the author's opinion that the computing systems of the future will involve active memory systems to further enhance concurrency and decentralize the processor's role.

Though the performance of the dataflow computer was not impressive, this idea is not a dead end. There is plenty of room for improvement and performance enhancement, using methods suggested in Section 5. Besides, the idea of implementing a dataflow computer using intelligent-memory systems, rather than using a modified PE is in itself novel and deserves further investigation.

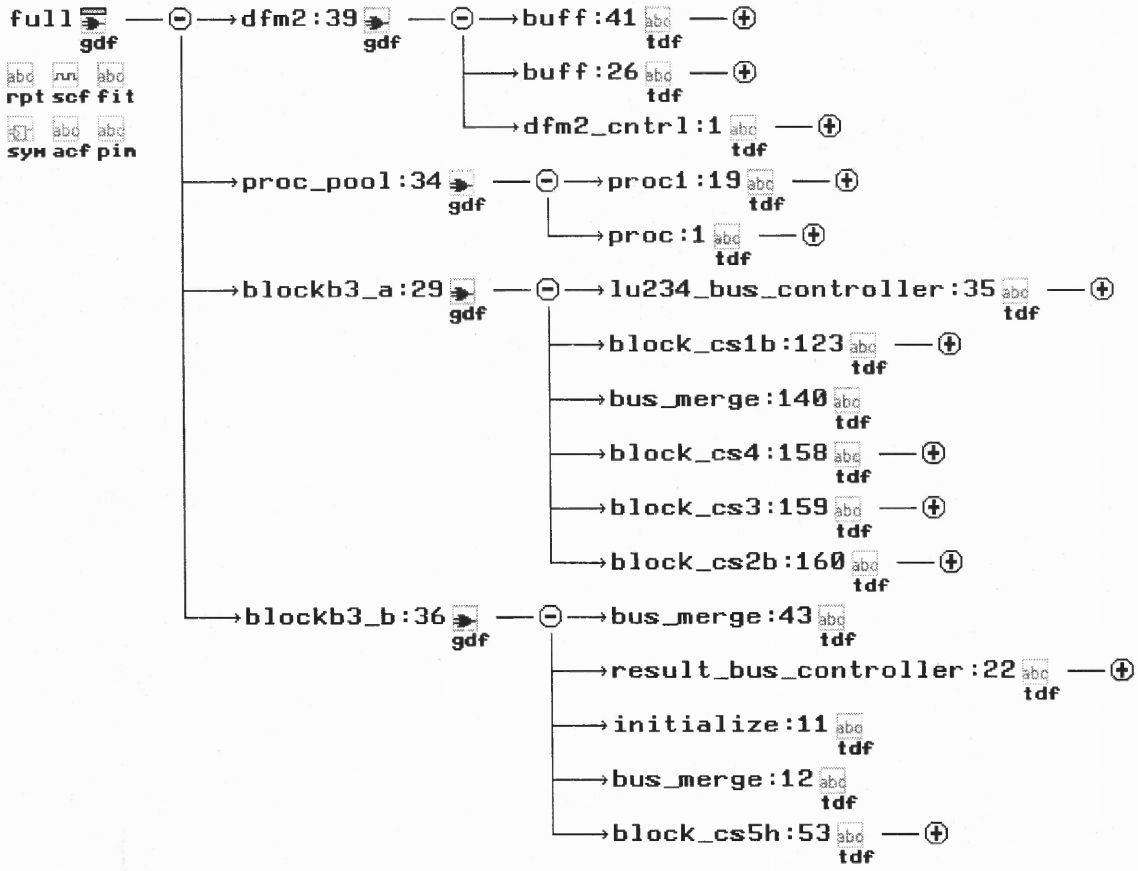
In addition to presenting a new way to implement a dataflow system, this thesis also showed how valuable FPGAs are in prototyping. Their ease of use, flexibility, power and cost make them an incredible asset for any kind of digital application.

This thesis provided me with an incredible opportunity to investigate and implement a dataflow computer. It has challenged my skill as a researcher, and tested my ability to overcome engineering hurdles. It has been a long and arduous nine months, but the journey has been rewarding and well worth the effort.

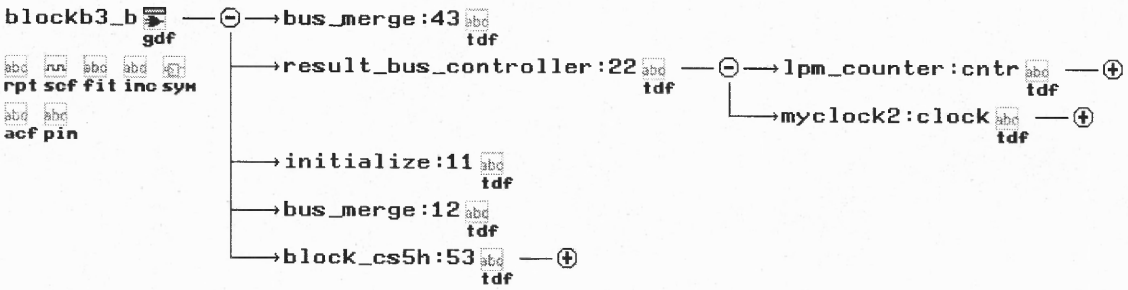
APPENDIX

Appendix A – Hierarchies and Programs

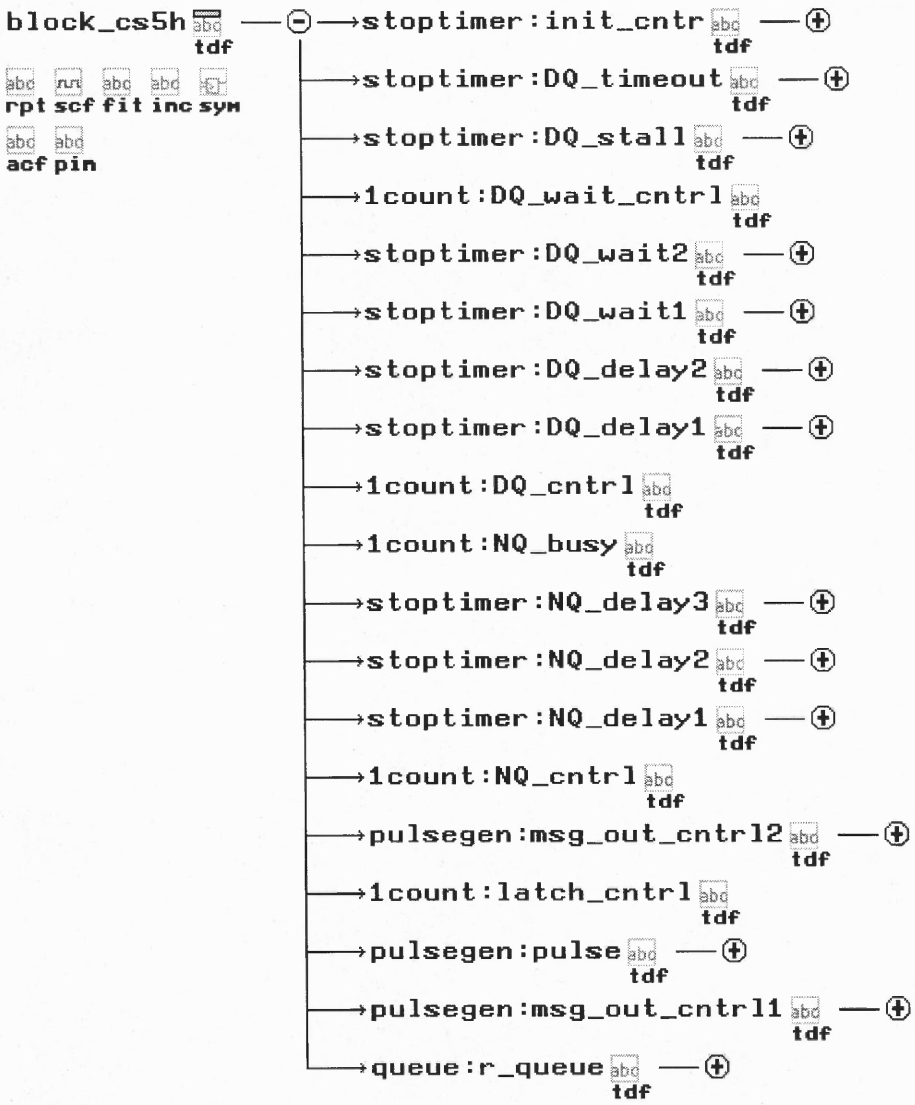
A.1 Hierarchy of the whole Dataflow Computer



A.2 Hierarchy of module – BLOCKB3_B



A.3 Hierarchy and TDF Implementation of module – BLOCK_CS5H




```
TITLE "Block of CellSection5";

INCLUDE "1count";
INCLUDE "myclock2";
INCLUDE "queue";
INCLUDE "counter";
INCLUDE "stoptimer";
INCLUDE "pulsegen";

PARAMETERS
(
   Addr = 7
);
```

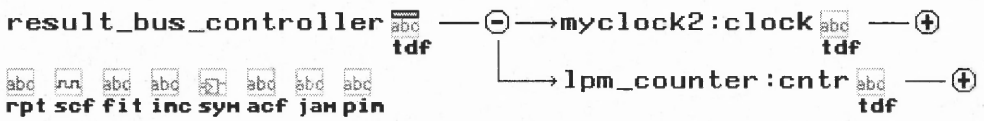
```
SUBDESIGN block_cs5h
(
   msg_in[17..0]      : INPUT;
   lu234busy          : INPUT;
   clk                : INPUT;
   msg_on_bus         : INPUT;
   msg_for_LU234      : OUTPUT;
   msg_o[17..0]       : OUTPUT;
   busy               : OUTPUT;
   block_addr[2..0]   : OUTPUT;
)

VARIABLE
   msg_in_reg[17..0]        : DFFE;
   msg_fromQ_reg[17..0]     : DFFE;
   msg_out[17..0]           : latch;
   r_queue                : queue WITH (D_Width = 18, A_Width = 8);

   msg_out_cntrl1     : pulsegen WITH (PW = 2, DEL = 2);
   pulse              : pulsegen WITH (PW = 1, DEL = 5);
   latch_cntrl        : 1count;

   msg_out_cntrl2     : pulsegen WITH (PW = 2, DEL = 1);

   NQ_cntrl           : 1count;
   NQ_delay1          : stoptimer WITH (DELAY = 1);
   NQ_delay2          : stoptimer WITH (DELAY = 2);
   NQ_delay3          : stoptimer WITH (DELAY = 3);
   protect_NQ         : DFFE;

   NQ_busy            : 1count;
   DQ_ON              : latch;

   DQ_cntrl           : 1count;
   DQ_delay1          : stoptimer WITH (DELAY = 1);
   DQ_delay2          : stoptimer WITH (DELAY = 2);
   DQ_wait1           : stoptimer WITH (DELAY = 2);
   DQ_wait2           : stoptimer WITH (DELAY = 3);
   DQ_wait_cntrl      : 1count;


   DQ_stall           : stoptimer WITH (DELAY = 4);

   DQ_timeout         : stoptimer WITH (DELAY = 7);
   From_RQ            : NODE;

   init_cntr          : stoptimer WITH (DELAY = 6);

BEGIN
   block_addr[] = addr;
   msg_in_reg[].clk = msg_on_bus;
   msg_fromQ_reg[].clk = clk;
```

```
msg_in_reg[].ena = VCC;

r_queue.clk = clk;
init_cntr.clock = clk;
init_cntr.clken = VCC;

msg_in_reg[].d = msg_in[];
msg_fromQ_reg[].d = r_queue.DQ_data[];


IF !r_queue.empty THEN
   msg_out[].d = msg_fromQ_reg[].q;
ELSE
   msg_out[].d = msg_in_reg[].q;
END IF;
msg_o[] = msg_out[].q;
msg_out[].ena = latch_cntrl.out # DQ_wait_cntrl.out;


latch_cntrl.clk = msg_on_bus # pulse.signal;
latch_cntrl.ena = !lu234busy  & init_cntr.timeup & !DQ_ON.q;

msg_out_cntrl1.clk = clk;
msg_out_cntrl1.trig = msg_on_bus & !lu234busy  & !DQ_ON.q;
msg_out_cntrl1.ena = !lu234busy  & !DQ_ON.q;

pulse.clk = clk;
pulse.trig = msg_out_cntrl1.signal;
pulse.ena = VCC;

NQ_delay1.clock = clk;
NQ_delay1.clken = VCC;
NQ_delay1.clr = msg_on_bus;

NQ_delay2.clock = clk;
NQ_delay2.clken = VCC;
NQ_delay2.clr = msg_on_bus;

NQ_cntrl.clk = NQ_delay1.timeup;
NQ_cntrl.clrn = NQ_delay2.timeup;

NQ_delay3.clock = clk;
NQ_delay3.clken = VCC;
NQ_delay3.clr = msg_on_bus;

protect_NQ.clk = clk;
protect_NQ.d = VCC;
protect_NQ.ena = lu234busy;
protect_NQ.clrn = lu234busy # !NQ_delay3.timeup # queue.full;

NQ_busy.clk = msg_on_bus # pulse.signal;
NQ_busy.clrn = r_queue.wr_busy;
```

```
DQ_ON.d = !r_queue.empty;
DQ_ON.ena = !NQ_busy.out;


IF (protect_NQ.q # DQ_ON.q) & !r_queue.wr_busy & !r_queue.full
                                                              THEN
   r_queue.NQ = NQ_cntrl.out;
END IF;
r_queue.NQ_data[] = msg_in_reg[].q;

busy = NQ_busy.out # r_queue.wr_busy # r_queue.full;

msg_out_cntrl2.clk = clk;
msg_out_cntrl2.trig = DQ_wait_cntrl.out;
msg_out_cntrl2.ena = !lu234busy;

DQ_wait1.clock = clk;
DQ_wait1.clken = VCC;
DQ_wait1.clr = DQ_cntrl.out;

DQ_wait2.clock = clk;
DQ_wait2.clken = VCC;
DQ_wait2.clr = DQ_cntrl.out;

DQ_wait_cntrl.clk = DQ_wait1.timeup & init_cntr.timeup;
DQ_wait_cntrl.clrn = DQ_wait2.timeup;

DQ_delay1.clock = clk;
DQ_delay1.clken = VCC;
DQ_delay1.clr = From_RQ;

DQ_delay2.clock = clk;
DQ_delay2.clken = VCC;
DQ_delay2.clr = From_RQ;

DQ_cntrl.clk = DQ_delay1.timeup;
DQ_cntrl.clrn = DQ_delay2.timeup;

DQ_stall.clock = clk;
DQ_stall.clken = VCC;
DQ_stall.clr = msg_for_LU234;

DQ_timeout.clock = clk;
DQ_timeout.clken = clk;
DQ_timeout.clr = From_RQ;

IF !lu234busy & !r_queue.empty & DQ_timeout.timeup &
                                        DQ_stall.timeup THEN
   From_RQ = VCC;
END IF;
r_queue.DQ = DQ_cntrl.out & init_cntr.timeup & !r_queue.empty;
```

```
   msg_for_LU234 = msg_out_cntrl1.signal #
msg_out_cntrl2.signal;
END;
```

A.4 Hierarchy and TDF Implementation of module – RESULT_BUS_CONTROLLER



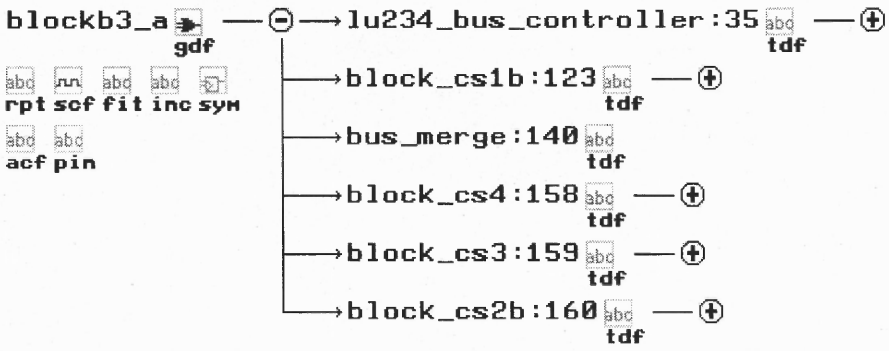

```
TITLE "Result Bus Controller";

INCLUDE "myclock2";
INCLUDE "lpm_counter";

SUBDESIGN result_bus_controller
(
   ask_permission[3..0]     : INPUT = GND;
   clk                      : INPUT;
   busy                     : INPUT = GND;
   give_permission[3..0]    : OUTPUT;
   msg_on_bus               : OUTPUT;
)

VARIABLE
   ask_permission_reg[3..0]    : DFF;
   give_permission[3..0]       : DFF;
   clock                       : myclock2 WITH (LO = 5, HI = 1);
   cntr                        : lpm_counter WITH (LPM_WIDTH = 3,
                          LPM_DIRECTION = "UP", LPM_MODULUS = 4);
   clr_ask_perm                : NODE;
   clr_give_perm               : NODE;
   msg_on_bus                  : DFFE;
   msg_on_bus_cntrl            : NODE;


BEGIN
   clock.inclock = clk;
   cntr.cnt_en = give_permission0 # give_permission1 #
                              give_permission2 # give_permission3;
   cntr.aclr = !(give_permission0 # give_permission1 #
                             give_permission2 # give_permission3);
   cntr.clock = clk;

   IF cntr.q[] == 3 THEN
      clr_ask_perm = clk;
      clr_give_perm = clk;
   ELSE
      clr_ask_perm = VCC;
      clr_give_perm = VCC;
   END IF;
```

```
   IF cntr.q[] > 1 THEN
      msg_on_bus_cntrl = VCC;
   END IF;

   msg_on_bus.d = VCC;
   msg_on_bus.clk = !clk;
   msg_on_bus.ena = msg_on_bus_cntrl;
   msg_on_bus.clrn = cntr.q0 # cntr.q1;

   ask_permission_reg[].clk = clock.signal;
   ask_permission_reg[].d = ask_permission[];
   ask_permission_reg[].clrn = clr_ask_perm;

   give_permission[].clk = !clock.signal;
   give_permission[].clrn = clr_give_perm;


   IF !busy THEN
      CASE ask_permission_reg[] IS
         WHEN B"XXX1" =>
            give_permission[].d = B"0001";
         WHEN B"XX10" =>
            give_permission[].d = B"0010";
         WHEN B"X100" =>
            give_permission[].d = B"0100";
         WHEN B"1000" =>
            give_permission[].d = B"1000";
         WHEN OTHERS =>
            give_permission[].d = GND;
      END CASE;
   END IF;

END;
```

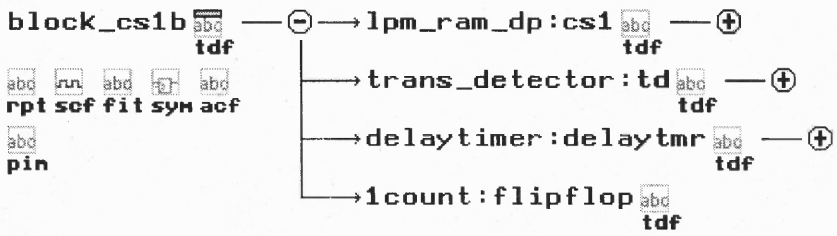
A.5 TDF Implementation of module – INITIALIZE

```
TITLE "Initialize Process";

SUBDESIGN initialize
(
   msg_in[17..0]   : INPUT;
   msg_out[17..0]  : OUTPUT;
   init            : INPUT;
)

BEGIN
   IF init THEN
      msg_out[] = B"111111111110000001";
   ELSE
      msg_out[] = msg_in[];
   END IF;
END;
```

A.6 Hierarchy of module – BLOCKB3_A



A.7 Hierarchy and TDF Implementation of module – BLOCK_CS1B



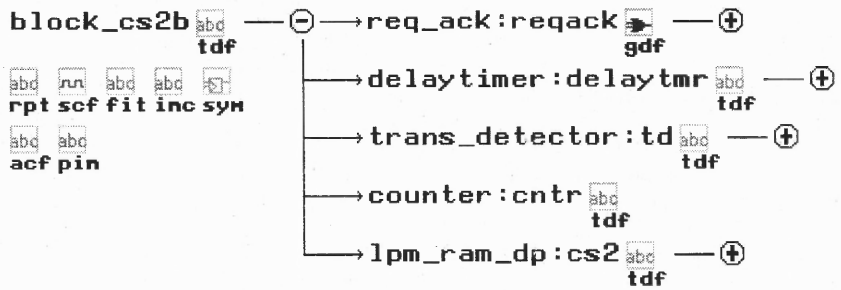

```
TITLE "Block of CellSection1";

INCLUDE "lpm_ram_dp";
INCLUDE "delaytimer";
INCLUDE "stoptimer2";
INCLUDE "trans_detector";
INCLUDE "1count";


SUBDESIGN block_cs1b
(
   msg_i[16..0]    : INPUT;
   msg_on_bus      : INPUT;
   inp_a[7..0]     : INPUT;
   inp_d[24..0]    : INPUT;
   clk, load       : INPUT;

   I_send_ack      : INPUT;
   M_send_ack      : INPUT;
   instr[25..0]    : OUTPUT;
   msg_o[14..0]    : OUTPUT;
   busy            : OUTPUT
   I_send_req      : OUTPUT
   M_send_req      : OUTPUT
   I_tri_ena       : OUTPUT
   M_tri_ena       : OUTPUT
)

VARIABLE
   cs1       : lpm_ram_dp WITH (LPM_WIDTH=25,
                        LPM_WIDTHAD=8);
   td        : trans_detector WITH (FV = 1,
                    DEL = 7);

   temp[24..0], tmp[24..0] : DFF;
   wr_reg[24..0]           : DFF;
   delaytmr                : delaytimer WITH (DELAY = 3);
   flipflop                : 1count;
   wena, rena              : NODE;
   wr                   : NODE;
```

```
    timeup                  : NODE;
    cc[2..0]                : DFF;
    msg_in_reg[16..0]       : DFF;

BEGIN

    -- MEMORY PIN ASSIGNMENTS
    cs1.rdaddress[] = msg_in_reg[16..9];
    cs1.wraddress[] = msg_in_reg[16..9] # inp_a[];
    cs1.data[] = wr_reg[].q # inp_d[];

    cs1.rdclken = VCC;
    cs1.rden = rena;
    cs1.wrclken = VCC;
    cs1.wren = wena;
    cs1.wrclock = clk;
    cs1.rdclock = clk;
    wena = load # wr;
    rena = !wena;
    -- MEMORY PIN ASSIGNMENTS

    msg_in_reg[].d = msg_i[];
    msg_in_reg[].clk = td.trans;

    %* Initializing the pins of Delaytimer *%
    delaytmr.clock = clk & ((M_send_req XNOR M_send_ack) XNOR
                                    (I_send_req XNOR I_send_ack));
    delaytmr.clken = td.trans;

    timeup = delaytmr.timeup;

    td.detect = msg_on_bus;
    td.clk = clk & ((M_send_req XNOR M_send_ack)
                                XNOR (I_send_req XNOR I_send_ack));
    cc[].clk = clk;

    IF td.trans # load THEN
       busy = VCC;
    ELSE
       busy = GND;
    END IF;

    temp[].clk = clk;
    temp[].clrn = wena;
    wr_reg[].clk = !clk;
    wr_reg[].clrn = wena;
    wr_reg[24..3].d = temp[24..3].q;
    tmp[].d = cs1.q[];
    tmp[].clk = clk;

    flipflop.clk = timeup;
    flipflop.clrn = !td.trans;
```

```
IF flipflop.out & td.trans THEN
   wr = VCC;
END IF;

IF td.trans & flipflop.out THEN
   CASE msg_in_reg[1..0] IS
      WHEN B"01" =>
         temp[].d = (tmp[24..18].q, msg_in_reg[8..2],
                              tmp[10..2].q, B"1", tmp[0].q);
      WHEN B"10" =>
         temp[].d = (msg_in_reg[8..2], tmp[17..3].q, B"1",
                                                tmp[1..0].q);
      WHEN OTHERS =>
         temp[].d = (tmp[24..1].q, msg_in_reg[2]);
   END CASE;

   CASE temp[2..0].q IS
      WHEN B"011", B"101" =>
         IF temp[6..5].q == B"01" & msg_in_reg[1..0] != B"11"
                                                        THEN
            msg_o[] = msg_in_reg[16..2];
            M_send_req = VCC;
         ELSIF temp[6..5].q == B"01" & msg_in_reg[1..0] ==
                                                   B"11" THEN
            IF temp1 THEN
               msg_o[] = (msg_in_reg[16..9], temp[17..11]);
               M_send_req = VCC;
            ELSE
               msg_o[] = (msg_in_reg[16..9], temp[24..18]);
               M_send_req = VCC;
            END IF;
         END IF;

      WHEN B"111" =>
         IF temp[6..5].q == B"11" THEN
            msg_o[] = (msg_in_reg[16..9], temp[17..11].q);
            M_send_req = VCC;
         ELSE
            instr[] = (msg_in_reg[16..9], temp[24..7].q);
            I_send_req = VCC;
         END IF;
   END CASE;

I_tri_ena = I_send_ack;
M_tri_ena = M_send_ack;


CASE temp[2..0].q IS
   WHEN B"111" =>
         IF !temp[3].q & !temp[4].q & temp[6..5].q == B"00"
                                                        THEN
         cc[].d = B"000";
      ELSIF !temp[3].q & temp[4].q & temp[6..5].q == B"00"
```

```
THEN
            cc[].d = B"100";
         ELSIF temp[3].q & !temp[4].q & temp[6..5].q == B"00"
                                                              THEN
            cc[].d = B"010";
         ELSIF !temp[3].q & !temp[4].q & temp[6..5].q != B"00"
                                                              THEN
            cc[].d = B"001";
         ELSIF !temp[3].q & temp[4].q & temp[6..5].q != B"00"
                                                              THEN
            cc[].d = B"101";
         ELSIF temp[3].q & !temp[4].q & temp[6..5].q != B"00"
                                                              THEN
            cc[].d = B"011";
         END IF;
      WHEN B"011", B"101" =>
         IF temp[6..5].q == B"01" THEN
            cc[].d = B"001";
         ELSE
            cc[].d = temp[2..0].q;
         END IF;
      WHEN OTHERS =>
         cc[].d = temp[2..0].q;
      END CASE;
   END IF;

   wr_reg[2..0].d = cc[].q;

END;
```

A.8 Hierarchy and TDF Implementation of module – BLOCK_CS2B

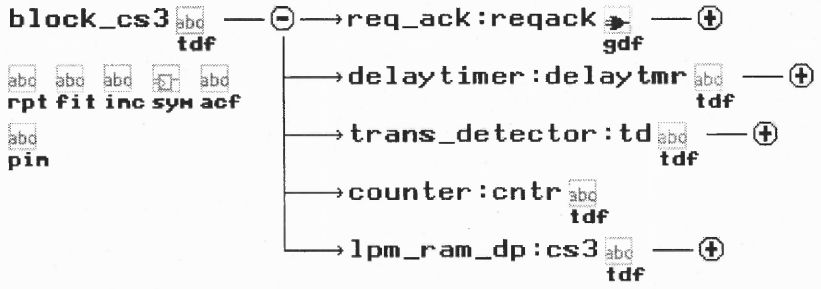



```
TITLE "Block of CellSection2";

INCLUDE "lpm_ram_dp.inc";
INCLUDE "delaytimer.inc";
INCLUDE "stoptimer.inc";
INCLUDE "counter";
INCLUDE "trans_detector";
INCLUDE "1count";
INCLUDE "req_ack";


SUBDESIGN block_cs2b
(
   msg_i[17..0]    : INPUT;
   inp_a[7..0]     : INPUT;
   inp_d[11..0]    : INPUT;
   clk, load       : INPUT;
   msg_on_bus      : INPUT;
   msg_o[16..0]    : OUTPUT;
   busy          : OUTPUT;

   M_to_LU1_req    : OUTPUT;
   M_to_LU1_ack    : INPUT;
   M_tri_ena       : OUTPUT;

)

VARIABLE
   cs2           : lpm_ram_dp WITH (LPM_WIDTH=12,
                                    LPM_WIDTHAD=8);
   cntr          : counter WITH (SZ = 9);
   msg_reg[14..0]     : DFF;
   msg_in_reg[17..0]  : DFF;
   w_addr[7..0]       : DFF;
   r_addr[7..0]       : NODE;
   td                 : trans_detector WITH (FV = 1);
   cr                 : DFF;
   addr[10..0]     : DFF;
```

```
    delaytmr        : delaytimer WITH (DELAY = 4);
    timeup          : NODE;
    rena, wena      : NODE;
    wr              : NODE;
    td_reset        : NODE;
    reqack          : Req_Ack;

BEGIN
    msg_in_reg[].clk = td.trans;
    msg_in_reg[].d = msg_i[];

    msg_o[] = (msg_reg[], B"11");
    M_tri_ena = M_to_LU1_ack;

    td.detect = msg_on_bus;
    td.clk = clk;
    td_reset = cntr.q8;
    td.in = td_reset;

    cntr.clk = timeup;
    cntr.ena = td.trans;
    cntr.clr = !td.trans;

    delaytmr.clock = clk;
    delaytmr.clken = td.trans & !M_to_LU1_req;
    timeup = delaytmr.timeup;

    r_addr[] = cntr.q[7..0];
    w_addr[].d = r_addr[];
    w_addr[].clk = timeup;
    rena = VCC;
    wena = load # wr;
    cs2.rdaddress[] = r_addr[];
    cs2.wraddress[] = w_addr[] # inp_a[];
    cs2.data[] = (msg_in_reg[17..7].q, cr) # inp_d[];
    cs2.rden = rena;

    cs2.wren = wena;
    cs2.wrclock = clk;
    cs2.rdclock = clk;

    cr.d = cs2.q0;
    cr.clk = timeup;
    addr[].d = cs2.q[11..1];
    addr[].clk = timeup;
    addr[].clrn = reqack.out;
    w_addr[].clrn = td.trans;

    busy = td.trans # load;


    reqack.clk = clk;
    reqack.req = M_to_LU1_req;
```

```
   reqack.ack = M_to_LU1_ack;

   IF td.trans & cr & addr[] == msg_in_reg[17..7].q  THEN
      M_to_LU1_req = VCC;
   ELSE
      M_to_LU1_req = GND;
   END IF;

   msg_reg[].clk = clk & td.trans;
   msg_reg[].d = (w_addr[], msg_in_reg[6..0].q);

   IF M_to_LU1_ack THEN
      IF r_addr[] != w_addr[] THEN
         wr = VCC;
      ELSE
         wr = GND;
      END IF;
   END IF;

END;
```

A.9 Hierarchy and TDF Implementation of module – BLOCK_CS3

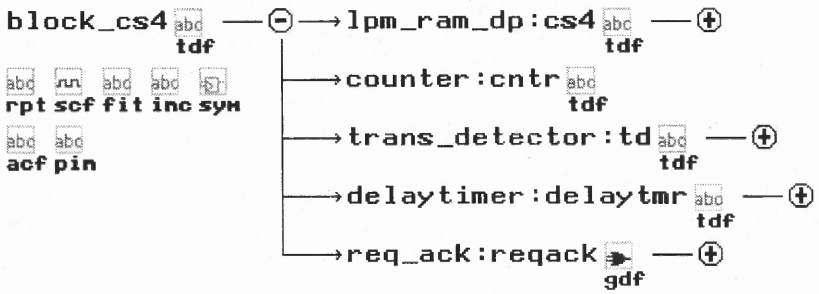



```
TITLE "Block of CellSection3";

INCLUDE "lpm_ram_dp.inc";
INCLUDE "delaytimer.inc";
INCLUDE "stoptimer.inc";
INCLUDE "counter";
INCLUDE "trans_detector";
INCLUDE "1count";
INCLUDE "req_ack";


SUBDESIGN block_cs3
(
   msg_i[17..0]    : INPUT;
   inp_a[7..0]     : INPUT;
   inp_d[11..0]    : INPUT;
   clk, load       : INPUT;
   msg_on_bus      : INPUT;
   msg_o[16..0]    : OUTPUT;
   busy            : OUTPUT;

   M_to_LU1_req    : OUTPUT;
   M_to_LU1_ack    : INPUT;
   M_tri_ena       : OUTPUT;
)

VARIABLE
   cs3          : lpm_ram_dp WITH (LPM_WIDTH=12,
                                   LPM_WIDTHAD=8);
   cntr                  : counter WITH (SZ = 9);
   msg_reg[14..0]        : DFF;
   msg_in_reg[17..0]     : DFF;
   w_addr[7..0]          : DFF;
   r_addr[7..0]          : NODE;
   td                    : trans_detector WITH (FV = 1);
   d1r                   : DFF;
   addr[10..0]     : DFF;
```

```
    delaytmr        : delaytimer WITH (DELAY = 4);
    timeup          : NODE;
    rena, wena      : NODE;
    wr              : NODE;
    td_reset        : NODE;
    reqack          : Req_Ack;


BEGIN
    msg_in_reg[].clk = td.trans;
    msg_in_reg[].d = msg_i[];

    msg_o[] = (msg_reg[], B"01");
    M_tri_ena = M_to_LU1_ack;

    td.detect = msg_on_bus;
    td.clk = clk;
    td_reset = cntr.q8;
    td.in = td_reset;

    cntr.clk = timeup;
    cntr.ena = td.trans;
    cntr.clr = !td.trans;

    delaytmr.clock = clk;
    delaytmr.clken = td.trans & !M_to_LU1_req;
    timeup = delaytmr.timeup;

    r_addr[] = cntr.q[7..0];
    w_addr[].d = r_addr[];
    w_addr[].clk = timeup;
    rena = VCC;
    wena = load # wr;
    cs3.rdaddress[] = r_addr[];
    cs3.wraddress[] = w_addr[] # inp_a[];
    cs3.data[] = (msg_in_reg[17..7], d1r) # inp_d[];
    cs3.rden = rena;
    cs3.wren = wena;
    cs3.wrclock = clk;
    cs3.rdclock = clk;

    d1r.d = cs3.q0;
    d1r.clk = timeup;
    addr[].d = cs3.q[11..1];
    addr[].clk = timeup;
    addr[].clrn = reqack.out;
    w_addr[].clrn = td.trans;

    busy = td.trans # load;

    reqack.clk = clk;
    reqack.req = M_to_LU1_req;
    reqack.ack = M_to_LU1_ack;
```

```
   IF td.trans & d1r & addr[] == msg_in_reg[17..7]  THEN
      M_to_LU1_req = VCC;
   ELSE
      M_to_LU1_req = GND;
   END IF;

   msg_reg[].clk = clk & td.trans;
   msg_reg[].d = (w_addr[], msg_in_reg[6..0]);

   IF M_to_LU1_ack THEN
      IF r_addr[] != w_addr[] THEN
         wr = VCC;
      ELSE
         wr = GND;
      END IF;
   END IF;

END;
```

A.10 Hierarchy and TDF Implementation of module – BLOCK_CS4

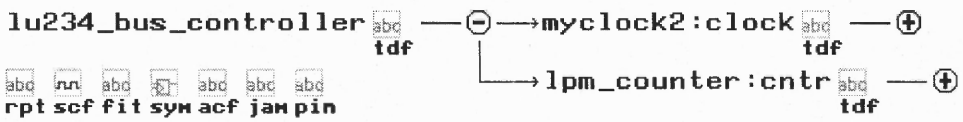



```
TITLE "Block of CellSection4";

INCLUDE "lpm_ram_dp.inc";
INCLUDE "delaytimer.inc";
INCLUDE "stoptimer.inc";
INCLUDE "counter";
INCLUDE "trans_detector";
INCLUDE "lcount";
INCLUDE "req_ack";


SUBDESIGN block_cs4
(
   msg_i[17..0]    : INPUT;
   inp_a[7..0]     : INPUT;
   inp_d[11..0]    : INPUT;
   clk, load       : INPUT;
   msg_on_bus      : INPUT;
   msg_o[16..0]    : OUTPUT;
   busy          : OUTPUT;

   M_to_LU1_req    : OUTPUT;
   M_to_LU1_ack    : INPUT;
   M_tri_ena       : OUTPUT;
)

VARIABLE
   cs4           : lpm_ram_dp WITH (LPM_WIDTH=12, LPM_WIDTHAD=8);
   cntr                : counter WITH (SZ = 9);
   msg_reg[14..0]      : DFF;
   msg_in_reg[17..0]   : DFF;
   w_addr[7..0]        : DFF;
   r_addr[7..0]        : NODE;
   td                  : trans_detector WITH (FV = 1);
   d2r                 : DFF;
   addr[10..0]         : DFF;
   delaytmr            : delaytimer WITH (DELAY = 4);
   timeup          : NODE;
```

```
    rena, wena      : NODE;
    wr              : NODE;
    td_reset        : NODE;
    reqack          : Req_Ack;

BEGIN
    msg_in_reg[].clk = td.trans;
    msg_in_reg[].d = msg_i[];

    msg_o[] = (msg_reg[], B"10");
    M_tri_ena = M_to_LU1_ack;

    td.detect = msg_on_bus;
    td.clk = clk;
    td_reset = cntr.q8;
    td.in = td_reset;

    cntr.clk = timeup;
    cntr.ena = td.trans;
    cntr.clr = !td.trans;

    delaytmr.clock = clk;
    delaytmr.clken = td.trans & !M_to_LU1_req;
    timeup = delaytmr.timeup;

    r_addr[] = cntr.q[7..0];
    w_addr[].d = r_addr[];
    w_addr[].clk = timeup;
    rena = VCC;
    wena = load # wr;
    cs4.rdaddress[] = r_addr[];
    cs4.wraddress[] = w_addr[] # inp_a[];
    cs4.data[] = (msg_in_reg[17..7], d2r) # inp_d[];
    cs4.rden = rena;
    cs4.wren = wena;
    cs4.wrclock = clk;
    cs4.rdclock = clk;

    d2r.d = cs4.q0;
    d2r.clk = timeup;
    addr[].d = cs4.q[11..1];
    addr[].clk = timeup;
    addr[].clrn = reqack.out;
    w_addr[].clrn = td.trans;

    busy = td.trans # load;

    reqack.clk = clk;
    reqack.req = M_to_LU1_req;
    reqack.ack = M_to_LU1_ack;

    IF td.trans & d2r & addr[] == msg_in_reg[17..7]  THEN
       M_to_LU1_req = VCC;
```

```
ELSE
   M_to_LU1_req = GND;
END IF;

msg_reg[].clk = clk & td.trans;
msg_reg[].d = (w_addr[], msg_in_reg[6..0]);

IF M_to_LU1_ack THEN
   IF r_addr[] != w_addr[] THEN
      wr = VCC;
   ELSE
      wr = GND;
   END IF;
END IF;
```

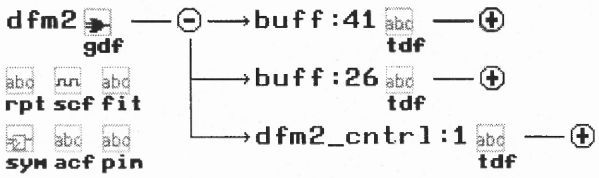
A.11 Hierarchy and TDF Implementation of module – LU234_BUS_CONTROLLER



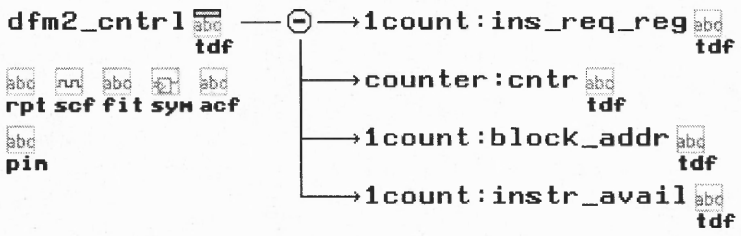

```
TITLE "Bus Controller";

INCLUDE "myclock2";
INCLUDE "lpm_counter";

SUBDESIGN LU234_bus_controller
(
   ask_permission[2..0]     : INPUT = GND;
   clk                      : INPUT;
   busy                     : INPUT = GND;
   give_permission[2..0]    : OUTPUT;
   msg_on_bus               : OUTPUT;
)

VARIABLE
   ask_permission_reg[2..0]      : DFF;
   give_permission[2..0]         : DFF;
   clock          : myclock2 WITH (LO = 5, HI = 1);
   cntr           : lpm_counter WITH (LPM_WIDTH = 3,
                        LPM_DIRECTION = "UP", LPM_MODULUS = 5);
   clr_ask_perm                  : NODE;
   clr_give_perm                 : NODE;
   msg_on_bus                    : DFFE;
   msg_on_bus_cntrl              : NODE;


BEGIN
   clock.inclock = clk;
   cntr.cnt_en = give_permission0 # give_permission1 #
                                          give_permission2;
   cntr.aclr = !(give_permission0 # give_permission1 #
                                         give_permission2);
   cntr.clock = clk;

   IF cntr.q[] == 3 THEN
      clr_ask_perm = clk;
      clr_give_perm = clk;
   ELSE
      clr_ask_perm = VCC;
```

```
      clr_give_perm = VCC;
   END IF;

   IF cntr.q[] > 1 THEN
      msg_on_bus_cntrl = VCC;
   END IF;

   msg_on_bus.d = VCC;
   msg_on_bus.clk = !clk;
   msg_on_bus.ena = msg_on_bus_cntrl;
   msg_on_bus.clrn = cntr.q0 # cntr.q1;


   ask_permission_reg[].clk = clock.signal;
   ask_permission_reg[].d = ask_permission[];
   ask_permission_reg[].clrn = clr_ask_perm;

   give_permission[].clk = !clock.signal;
   give_permission[].clrn = clr_give_perm;

   IF !busy THEN
      CASE ask_permission_reg[] IS
         WHEN B"XX1" =>
            give_permission[] = B"001";
         WHEN B"X10" =>
            give_permission[] = B"010";
         WHEN B"100" =>
            give_permission[] = B"100";
         WHEN OTHERS =>
            give_permission[] = GND;
      END CASE;
   END IF;

END;
```

A.12 Hierarchy of module – DFM2



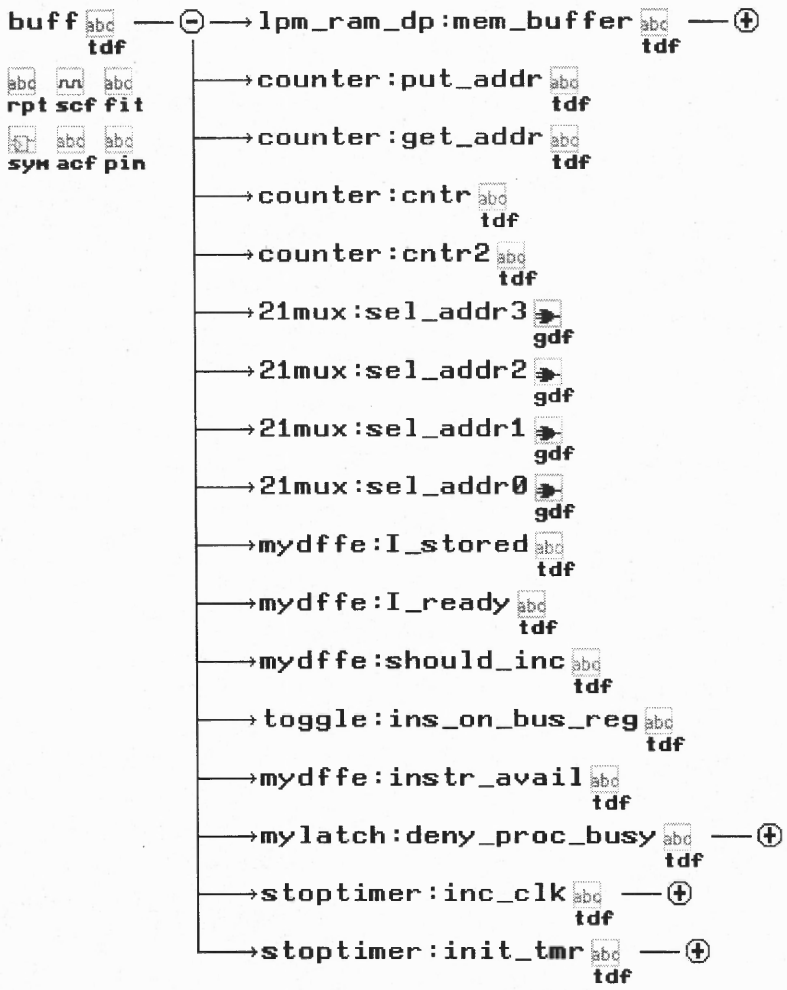
A.13 Hierarchy and TDF Implementation of module – DFM2_CNTRL




```
TITLE "DFM2 Controller";

INCLUDE "1count";
INCLUDE "counter";
INCLUDE "21mux";

SUBDESIGN DFM2_cntrl
(
   instr_in[28..0]       : INPUT;
   ins_send_req          : INPUT;
   read_cycle            : INPUT;
   ins_sent              : INPUT;
   ins_stored            : INPUT;
   clk                   : INPUT;

   ins_send_ack          : OUTPUT;
   instr_out[29..0]      : OUTPUT;
   sel_buffer            : OUTPUT;
   ins_on_bus            : OUTPUT;
)

VARIABLE
   instr_in_reg[28..0]     : DFF;
   instr_avail           : 1count;
   block_addr            : 1count;
   ack_cntr              : DFFE;
   cntr                  : counter WITH (sz = 2);
   ins_req_reg           : 1count;
   ena_ack_cntr          : NODE;
   set_ack_cntr          : NODE;

BEGIN
   instr_in_reg[].d = instr_in[];
   instr_out[] = (B"0",instr_in_reg[].q);

   ack_cntr.d = VCC;
   ack_cntr.ena = ena_ack_cntr;
   ack_cntr.clrn = !set_ack_cntr;
   ack_cntr.clk = !clk;
```

```
    cntr.clk = !clk;
    cntr.clr = !ack_cntr.q;

    ins_req_reg.clk = ins_send_req & read_cycle;
    ins_req_reg.clrn = ack_cntr.q;

    instr_avail.clk = ins_sent;
    instr_avail.clrn = ins_stored;

    IF !read_cycle THEN
       cntr.ena = ack_cntr.q;

       IF ins_req_reg.out & !instr_avail.out THEN
          ena_ack_cntr = VCC;
       END IF;

       IF  !cntr.q0 & cntr.q1 THEN
          set_ack_cntr = VCC;
       END IF;

       ins_send_ack = ack_cntr.q;
       instr_in_reg[].clk = ins_sent;
    END IF;

    IF read_cycle & instr_avail.out THEN
       ins_on_bus = VCC;
    END IF;

    block_addr.clk = ins_send_req;
    sel_buffer = block_addr.out;

END;
```

A.14 Hierarchy and TDF Implementation of module – BUFF




```
TITLE "Memory Buffer";

INCLUDE "lpm_ram_dp";
INCLUDE "lpm_counter";
INCLUDE "toggle";
INCLUDE "counter";
INCLUDE "21mux";
INCLUDE "mylatch";
INCLUDE "mydffe";
INCLUDE "stoptimer";


SUBDESIGN buff
(
   instr_in[29..0]      : INPUT;
   ins_on_bus           : INPUT;
```

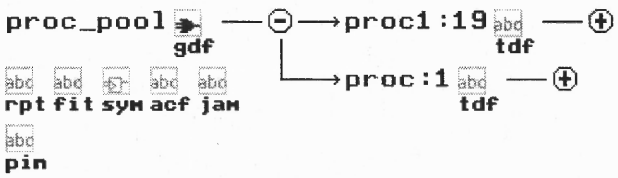
```
    instr_done          : INPUT;
    clk                 : INPUT;

    instr_stored        : OUTPUT;
    instr_ready         : OUTPUT;
    instr_out[28..0]    : OUTPUT;
    put_cycle           : OUTPUT;
)

VARIABLE
    mem_buffer      : lpm_ram_dp WITH (LPM_WIDTH=30,
                                       LPM_WIDTHAD=4);
    put_addr        : counter WITH (sz = 4);
    get_addr        : counter WITH (sz = 4);
    rdwrclk         : NODE;
    cntr            : counter WITH (sz = 3);
    cntr2           : counter WITH (sz = 4);
    addr[3..0]      : NODE;
    data[29..0]     : NODE;
    sel_addr[3..0]     : 21mux;
    wena               : NODE;
    inc_put, inc_get   : NODE;
    I_stored           : myDFFE;
    inc_istored        : NODE;
    I_ready            : myDFFE;
    inc_iready         : NODE;
    should_inc         : myDFFE;
    inc_should         : NODE;
    ins_on_bus_reg     : toggle;
    clr_ins_on_bus     : NODE;
    instr_out[28..0]   : latch;
    latch_Iout         : NODE;
    instr_avail        : myDFFE;
    inc_instr_avail    : NODE;
    deny_proc_busy     : mylatch;

    empty              : NODE;
    equal              : NODE;
    clr_should_inc     : NODE;
    clr_istored        : NODE;
    clr_iready         : NODE;
    dirty              : NODE;
    inc_clk            : stoptimer WITH (DELAY = 2);
    init_tmr           : stoptimer WITH (DELAY = 6);

BEGIN

    -- MEMORY PIN ASSIGNMENTS
    mem_buffer.rdaddress[] = addr[];
    mem_buffer.wraddress[] = addr[];
    mem_buffer.data[] = data[];
    mem_buffer.rdclken = VCC;
    mem_buffer.rden = !wena;
```

```
mem_buffer.wrclken = VCC;
mem_buffer.wren = wena;
mem_buffer.wrclock = clk;
mem_buffer.rdclock = clk;
-- MEMORY PIN ASSIGNMENTS

-- COUNTERs/RDWRCLK initialization
cntr.clk = clk;
cntr.ena = VCC;
cntr2.clk = clk;
cntr2.ena = VCC;
rdwrclk = cntr2.q3;
put_cycle = rdwrclk;


ins_on_bus_reg.clk = ins_on_bus;
ins_on_bus_reg.clrn = clr_ins_on_bus;

sel_addr[].a = put_addr.q[];
sel_addr[].b = get_addr.q[];
addr[] = sel_addr[].y;
sel_addr[].s = rdwrclk;

I_stored.clk = clk;
I_stored.ena = inc_istored;
I_stored.clrn = clr_istored;
I_stored.d = VCC;
instr_stored = I_stored.q;
should_inc.clk = clk;
should_inc.ena = inc_should;
should_inc.clrn = clr_should_inc;
should_inc.d = VCC;

wena = I_stored.q # I_ready.q;


clr_should_inc = cntr.q2 & cntr.q1 & !cntr.q0;
clr_ins_on_bus = cntr.q2 & cntr.q1 & cntr.q0 & rdwrclk;

IF ins_on_bus_reg.q & rdwrclk THEN
   data[] = instr_in[];
   CASE cntr.q[] IS
      WHEN 2 =>
         IF empty # dirty THEN
            inc_istored = VCC;
         END IF;
      WHEN 3 =>
         IF I_stored.q THEN
            inc_should = VCC;
         END IF;
      WHEN 4 =>
         clr_istored = VCC;
         IF should_inc.q THEN
```

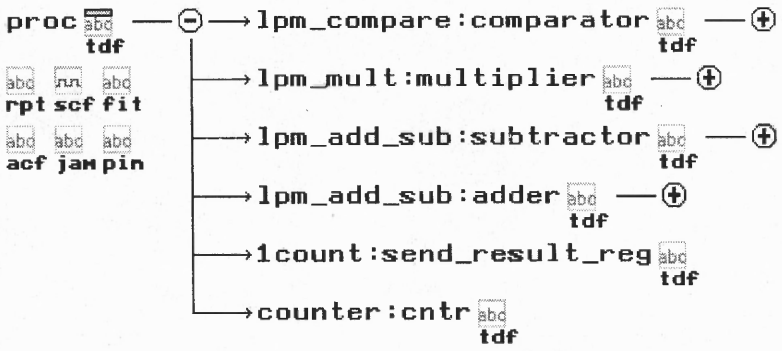
```
            inc_put = VCC;
         END IF;
   END CASE;
END IF;

I_ready.clk = clk;
I_ready.ena = inc_iready;
I_ready.clrn = clr_iready;
I_ready.d = VCC;
instr_out[].d = mem_buffer.q[28..0];
instr_avail.clk = clk;
instr_avail.ena = inc_instr_avail;
instr_avail.clrn = instr_done;
instr_avail.d = VCC;
instr_out[].ena = latch_Iout;
instr_ready = instr_avail.q;
deny_proc_busy.clk = clk;
deny_proc_busy.d = instr_avail.q;
deny_proc_busy.ena = rdwrclk;

IF put_addr.q[] == get_addr.q[] THEN
   equal = VCC;
END IF;

empty = !mem_buffer.q[3] & !mem_buffer.q[2] & !mem_buffer.q[1]
                                            & !mem_buffer.q[0];
dirty = mem_buffer.q[29];

IF !rdwrclk & !deny_proc_busy.q THEN
   data[] =  (B"1", mem_buffer.q[28..0]);
   CASE cntr.q[] IS
      WHEN 2 =>
         IF !empty & !dirty THEN
            latch_Iout = VCC;
            inc_instr_avail = VCC;
            inc_iready = VCC;
         END IF;
      WHEN 3 =>
         IF instr_avail.q & !equal THEN
            inc_should = VCC;
         END IF;
      WHEN 4 =>
         clr_iready = VCC;
         IF should_inc.q THEN
            inc_get = VCC;
         END IF;
   END CASE;
END IF;

put_addr.clk = inc_clk.timeup;
get_addr.clk = inc_clk.timeup;

put_addr.ena = rdwrclk & init_tmr.timeup;
```

```
    get_addr.ena = !rdwrclk & init_tmr.timeup;

    inc_clk.clock = clk;
    inc_clk.clken = VCC;
    inc_clk.clr = inc_get # inc_put;

    init_tmr.clock = clk;
    init_tmr.clken = VCC;

END;
```

A.15 Hierarchy of module – PROC_POOL



A.16 Hierarchy and TDF Implementation of module – PROC




```
TITLE "processor";

INCLUDE "1count";
INCLUDE "counter";
INCLUDE "lpm_add_sub";
INCLUDE "lpm_mult";
%INCLUDE "lpm_divide";%
INCLUDE "lpm_compare";

%
   ADD      0001
   SUB      0010
   MUL      0011
   DIV      0100
   CEQ      1000
   CNE      1001
   CGT      1010
   CLT      1011
   CGE      1100
   CLE      1101
%


SUBDESIGN proc
(
   instr_in[28..0]    : INPUT;
   instr_ready        : INPUT;
   clk                : INPUT;
   res_send_ack       : INPUT;

   res_send_req       : OUTPUT;
   result[17..0]      : OUTPUT;
   instr_done         : OUTPUT;
)


VARIABLE
```

```
    cntr                    : counter WITH (SZ = 3);
    result_reg[17..0]       : DFFE;
    send_result_reg         : lcount;

    --ALU Components
    adder                   : lpm_add_sub WITH (LPM_WIDTH = 6,
                                  LPM_REPRESENTATION = "unsigned");
    subtractor              : lpm_add_sub WITH (LPM_WIDTH = 7,
                                  LPM_REPRESENTATION = "unsigned");
    multiplier              : lpm_mult WITH (LPM_WIDTHA = 4,
                        LPM_WIDTHB = 3, LPM_WIDTHS = 7, LPM_WIDTHP = 7,
                                  LPM_REPRESENTATION = "unsigned");
%   divider                 : lpm_divide WITH (LPM_WIDTHN = 7,
                    LPM_WIDTHD = 7, LPM_NREPRESENTATION = "unsigned",
                                  LPM_DREPRESENTATION = "unsigned");%
    comparator              : lpm_compare WITH (LPM_WIDTH = 7,
                                  LPM_REPRESENTATION = "unsigned");

BEGIN
    cntr.clk = clk;
    cntr.ena = instr_ready & !send_result_reg.out;
    cntr.clr = send_result_reg.out;

    result_reg[17..7].d = instr_in[28..18];
    result_reg[].clk = !clk;
    result_reg[].ena = cntr.q2 & cntr.q1 & !cntr.q0;
    send_result_reg.clk = !clk;
    send_result_reg.ena = cntr.q2 & cntr.q1 & cntr.q0;
    send_result_reg.clrn = res_send_ack;
    res_send_req = send_result_reg.out;
    result[] = result_reg[];

    adder.dataa[] = instr_in[9..4];
    adder.datab[] = instr_in[16..11];
    adder.add_sub = VCC;

    subtractor.dataa[] = instr_in[10..4];
    subtractor.datab[] = instr_in[17..11];
    subtractor.add_sub = GND;

    multiplier.dataa[] = instr_in[7..4];
    multiplier.datab[] = instr_in[13..11];

%   divider.numer[] = instr_in[10..4];
    divider.denom[] = instr_in[17..11];%

    comparator.dataa[] = instr_in[10..4];
    comparator.datab[] = instr_in[17..11];

    instr_done = !clk & cntr.q2 & cntr.q1 & cntr.q0;

    IF !send_result_reg.out THEN
       CASE instr_in[3..0] IS
```

```
            WHEN B"0001" =>              % ADD %
               result_reg[6..0].d = (adder.cout, adder.result[]);
            WHEN B"0010" =>              % SUB %
               result_reg[6..0].d = subtractor.result[];
            WHEN B"0011" =>              % MUL %
               result_reg[6..0].d = multiplier.result[];
%           WHEN B"0100" =>               DIV
               result_reg[6..0].d = divider.quotient[];%
            WHEN B"1000" =>              % CEQ %
               result_reg[6..0].d = (B"000000", comparator.aeb);
            WHEN B"1001" =>              % CNE %
               result_reg[6..0].d = (B"000000", comparator.aneb);
            WHEN B"1010" =>              % CGT %
               result_reg[6..0].d = (B"000000", comparator.agb);
            WHEN B"1011" =>              % CLT %
               result_reg[6..0].d = (B"000000", comparator.alb);
            WHEN B"1100" =>              % CGE %
               result_reg[6..0].d = (B"000000", comparator.ageb);
            WHEN B"1101" =>              % CLE %
               result_reg[6..0].d = (B"000000", comparator.aleb);
         END CASE;
      END IF;

END;
```

A.17 TDF Implementation of module – 1COUNT

```
TITLE "1 Bit counter";

SUBDESIGN 1count
(
   clk          : INPUT;
   out          : OUTPUT;
   clrn         : INPUT = GND;
   ena          : INPUT = VCC;
)

VARIABLE
    ff          : DFF;
    inc         : NODE;
BEGIN
   inc = VCC;
   ff.clk = clk;

   IF ena THEN
      ff.d = inc XOR ff.q;
   ELSE
      ff.d = ff.q;
   END IF;
   ff.clrn = !clrn;
   out = ff.q;
END;
```

A.18 TDF Implementation of module – BUS_MERGE

```
TITLE "bus_merge";

PARAMETERS
(
   bus1_sz = 2,
   bus2_sz = 2
);

SUBDESIGN bus_merge
(
   bus1_[bus1_sz-1..0]      : INPUT;
   bus2_[bus2_sz-1..0]      : INPUT;
   bus[bus1_sz+bus2_sz-1..0]  : OUTPUT;
)

BEGIN
   bus[] = (bus2_[], bus1_[]);

END;
```

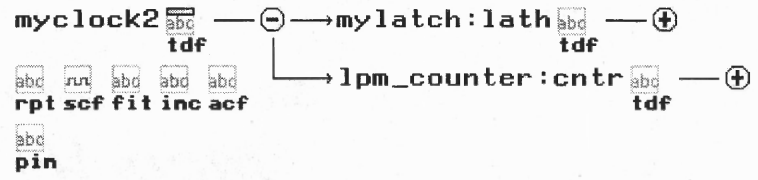
A.19 TDF Implementation of module – COUNTER

```
TITLE "counter";

PARAMETERS
(
   sz = 8
);

SUBDESIGN counter
(
   clk, ena          : INPUT;
   clr               : INPUT = GND;
   q[sz-1..0]        : OUTPUT;
)

VARIABLE
   count[sz-1..0]    : DFF;

BEGIN
   count[].clk = clk;
   count[].clrn = !clr;
   count[].d = GND;

   IF ena THEN
      count[].d = count[].q + 1;
   ELSE
      count[].d = count[].q;
   END IF;

   q[] = count[];
END;
```

A.20 TDF Implementation of module – DELAYTIMER

```
TITLE "Delay Timer";

INCLUDE "4count";
INCLUDE "1count";

%* HOW IT WORKS: This function is used in conjunction with delay
requirement in clock cycles. The user provides the clock and
clken signals. The clken signal normally low, and because the
clken is connected to the clrn line, the '8count' counter has
been stays cleared. When the value of the counter is equal to the
required delay, the timer sets the timeup signal high (else it is
low). This in turn tells the calling    function that it should
turn of the clken signal, which will clear and stop the timer *%

PARAMETERS
(
   DELAY = 1      %* delay is  in clock cycles *%
);

SUBDESIGN delaytimer
(
   clock, clken    : INPUT;
   timeup          : OUTPUT;
   ec              : INPUT = VCC; %* External Clear Signal *%
)

VARIABLE
   timer           : 4count;
   timeup          : DFF;
   count[3..0]     : NODE;

BEGIN
   timeup.clk = clock;
   IF count[] == 0 & !clock THEN
      timeup.clrn = GND;
   ELSE
      timeup.clrn = VCC;
   END IF;

   timer.clk = clock & clken;
   timer.dnup = GND;
   timer.clrn = !timeup & ec;
   count[] = (timer.qd, timer.qc, timer.qb, timer.qa);

   IF count[] >= DELAY THEN
      timeup = VCC;
   ELSE
      timeup = GND;
   END IF;
END;
```

A.21 Hierarchy and TDF Implementation of module – MYCLOCK2




```
TITLE "My Clock";

INCLUDE "lpm_counter";
INCLUDE "mylatch";

PARAMETERS
(
   LO = 6,   -- No of clock cycles that myclock o/p has to stay LO
   HI = 1    -- No of clock cycles that myclock o/p has to stay HI
);

SUBDESIGN myclock2
(
   inclock      : INPUT;        -- Input clock which drives myclock
   ena          : INPUT = VCC;
   signal       : OUTPUT;       -- Output clock signal
)

VARIABLE
   cntr         : lpm_counter WITH (LPM_WIDTH = 4, LPM_DIRECTION =
                                    "UP", LPM_MODULUS = LO+HI);
   outclock     : DFF;
   cntrl        : NODE;
   IF LO > 1 GENERATE
      lath         : mylatch;
      lath_cntrl      : NODE;
   END GENERATE;

BEGIN
   cntr.clock = inclock;          -- clock input to cntr
   cntr.cnt_en = ena;
   cntr.clk_en = ena;
   outclock.d = VCC;

   outclock.clk = !inclock;

   IF LO > 1 GENERATE
      outclock.clrn = lath.q;
   ELSE GENERATE
      outclock.clrn = cntrl # inclock;
   END GENERATE;
```

```
   signal = outclock.q;

   IF cntr.q[] < LO THEN
      cntrl = GND;
   ELSE
      cntrl = VCC;
   END IF;

   IF LO > 1 GENERATE
      lath.d = cntrl;

      IF cntr.q[] == LO or cntr.q[] == 0 THEN
         lath_cntrl = VCC;
      END IF;

      lath.clk = inclock;
      lath.ena = lath_cntrl;
   END GENERATE;

END;
```

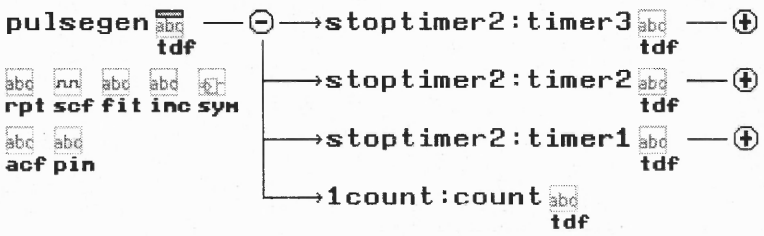
A.22 TDF Implementation of module – MYDFFE

```
TITLE "My DFFE";

SUBDESIGN mydffe
(
   d      : INPUT;
   ena       : INPUT;
   clk       : INPUT;
   clrn   : INPUT;

   q      : OUTPUT;
)

VARIABLE
   flipflop : DFFE;

BEGIN
   flipflop.d = d;
   flipflop.ena = ena;
   flipflop.clk = clk;
   flipflop.clrn = !clrn;

   q = flipflop.q;

END;
```

A.23 TDF Implementation of module – MYLATCH

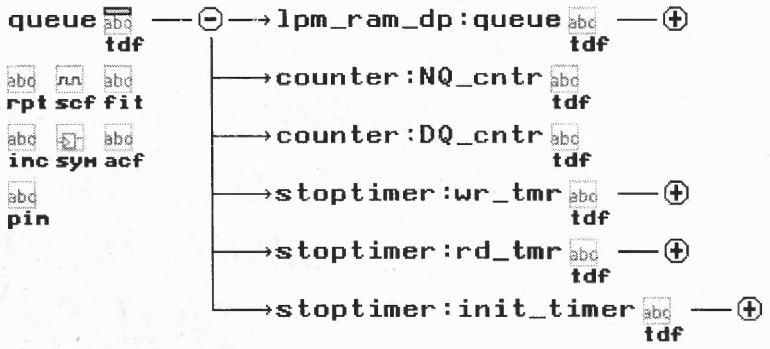
```
TITLE "My latch";

INCLUDE "delaytimer";
INCLUDE "1count";

SUBDESIGN mylatch
(
   D, clk, ena : INPUT;
   Q           : OUTPUT;
)
VARIABLE
   cnt1, cnt2          : 1count;
   lath              : latch;

BEGIN
   cnt1.clk = clk;
   cnt2.clk = cnt1.out & !clk;
   cnt1.clrn = cnt2.out;

   lath.ena =cnt1.out # ena;
   lath.d = D;
   Q = lath.q;

END;
```

A.24 Hierarchy and TDF Implementation of module – PULSEGEN




```
TITLE "Pulsegen";

INCLUDE "1count";
INCLUDE "stoptimer2";

PARAMETERS
(
   PW = 1,
   DEL = 1
);

SUBDESIGN pulsegen
(
   trig, clk      : INPUT;    -- Input clock which drives myclock
   ena            : INPUT = VCC;
   signal         : OUTPUT;   -- Output clock signal
)

VARIABLE
   count       : 1count;
   timer1      : stoptimer2 WITH (DELAY = DEL);
   timer2      : stoptimer2 WITH (DELAY = DEL+PW);
   timer3      : stoptimer2 WITH (DELAY = DEL+2+PW);

BEGIN
   count.clk = timer1.timeup;
   count.clrn = timer2.timeup;
   signal = count.out & timer3.timeup;

   timer1.clk = clk;
   timer1.clken = ena;
   timer1.clr = trig;

   timer2.clk = clk;
   timer2.clken = ena;
   timer2.clr = trig;

   timer3.clk = clk;
   timer3.clken = VCC;

END;
```

A.25 Hierarchy and TDF Implementation of module – QUEUE




```
TITLE "Queue in CellSection5";

INCLUDE "counter";
INCLUDE "stoptimer";
INCLUDE "lpm_ram_dp";

PARAMETERS
(
   D_Width = 18,
   A_Width = 8
);

SUBDESIGN queue
(
   NQ                          : INPUT;
   DQ                          : INPUT;
   clk                         : INPUT;
   NQ_Data[D_Width-1..0]       : INPUT;

   DQ_Data[D_Width-1..0]       : OUTPUT;
   full                        : OUTPUT;
   empty                       : OUTPUT;
   rd_busy                     : OUTPUT;
   wr_busy                     : OUTPUT;
)

VARIABLE
   queue           : lpm_ram_dp WITH (LPM_WIDTH=D_Width,
                      LPM_WIDTHAD=A_Width);
   NQ_cntr         : counter WITH (SZ = A_Width);
   DQ_cntr         : counter WITH (SZ = A_Width);
   wr_tmr          : stoptimer WITH (DELAY = 3);
   rd_tmr          : stoptimer WITH (DELAY = 3);
   init_timer      : stoptimer WITH (DELAY = 5);
   init            : NODE;
```

```
   in_reg[17..0]  : DFFE;
   out_reg[17..0] : DFFE;

BEGIN
   queue.rdaddress[] = DQ_cntr.q[];
   queue.wraddress[] = NQ_cntr.q[];
   queue.data[] = in_reg[].q;
   out_reg[] = queue.q[];
   queue.rden = !rd_tmr.timeup;
   queue.wren = !wr_tmr.timeup;
   queue.wrclock = clk;
   queue.rdclock = clk;

   in_reg[].d = NQ_data[];
   DQ_data[] = out_reg[].q;

   in_reg[].clk = clk;
   out_reg[].clk = !clk;

   wr_tmr.clock = clk;
   wr_tmr.clken = VCC;
   wr_tmr.clr = NQ;

   rd_tmr.clock = clk;
   rd_tmr.clken = VCC;
   rd_tmr.clr = DQ;

   NQ_cntr.clk = wr_tmr.timeup;
   DQ_cntr.clk = rd_tmr.timeup;

   NQ_cntr.ena = VCC;
   DQ_cntr.ena = VCC;

   rd_busy = !rd_tmr.timeup;
   wr_busy = !wr_tmr.timeup;

   init_timer.clock = clk;
   init_timer.clken = VCC;
   init = !init_timer.timeup;
   IF (DQ_cntr.q[] == NQ_cntr.q[]) # init THEN
      empty = VCC;
   END IF;

   IF DQ_cntr.q[] - NQ_cntr.q[]  == 1 THEN
      full = VCC;
   END IF;

END;
```

A.26 TDF Implementation of module – STOPTIMER

```
TITLE "Stop Timer";

INCLUDE "4count";

PARAMETERS
(
   DELAY = 6      -- delay is  in clock cycles
);

SUBDESIGN stoptimer
(
   clock, clken    : INPUT;
   timeup          : OUTPUT;
   clr             : INPUT = GND; %* External Clear Signal *%
)

VARIABLE
   timer        : 4count;
   count[3..0]     : NODE;

BEGIN
   timer.clk = clock & clken & !timeup;
   timer.dnup = GND;
   timer.clrn = !clr;
   count[] = (timer.qd, timer.qc, timer.qb, timer.qa);

   IF count[] >= DELAY THEN
      timeup = VCC;
   ELSE
      timeup = GND;
   END IF;
END;
```

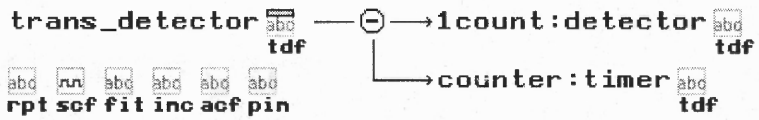
A.27 TDF Implementation of module – STOPTIMER2

```
TITLE "Stop Timer2";

INCLUDE "4count";
INCLUDE "1count";


PARAMETERS
(
   DELAY = 6      -- delay is  in clock cycles
);

SUBDESIGN stoptimer2
(
   clk, clken     : INPUT;
   timeup         : OUTPUT;
   clr            : INPUT = GND; -- External Clear Signal
)

VARIABLE
   timer          : 4count;
   count[3..0]    : NODE;
   clear          : dff;

BEGIN
   timer.clk = clk & clken & !timeup;
   timer.dnup = GND;
   timer.clrn = !clear.q;
   count[] = (timer.qd, timer.qc, timer.qb, timer.qa);
   clear.d = vcc;
   clear.clk = clr;
   clear.clrn = !clear.q;

   IF count[] >= DELAY THEN
      timeup = VCC;
   ELSE
      timeup = GND;
   END IF;
END;
```

A.28 TDF Implementation of module – TOGGLE

```
TITLE "Toggle";

SUBDESIGN toggle
(
   clk          : INPUT;
   q            : OUTPUT;
   clrn         : INPUT = GND;
   ena          : INPUT = VCC;

)

VARIABLE
    ff          : DFF;
    inc         : NODE;

BEGIN
   inc = VCC;
   ff.clk = clk & ena;

   ff.d = inc XOR ff.q;
   ff.clrn = !(!clk & clrn);
   q = ff.q;

END;
```

A.29 Hierarchy and TDF Implementation of module – TRANS_DETECTOR




```
TITLE "Transition Detector";

INCLUDE "1count";
INCLUDE "counter";

PARAMETERS
(
   FV = 1,
   DEL = 0
);

SUBDESIGN trans_detector
(
   detect      : INPUT;        -- Line to detect transition on
   in          : INPUT = GND;
   trans       : OUTPUT;
   NI          : OUTPUT;
   clk         : INPUT;
)

VARIABLE
   detector    : 1count;

   IF DEL != 0 GENERATE
      timer    : counter WITH (sz = 8);
      clear    : NODE;
      clr         : NODE;
   ELSE GENERATE
      timer    : counter WITH (sz = 2);
   END GENERATE;
BEGIN

   IF FV == 1 THEN
      detector.clk = detect # detector.out;
   ELSE
      detector.clk = !detect # detector.out;
   END IF;

   NI = in;
   trans = detector.out;
```

```
   IF DEL != 0 GENERATE
      clear = GND # clr;
      timer.clk = clk;
      timer.clr = clear;
      timer.ena = detector.out;

      detector.clrn = clear;
      IF timer.q[] == DEL THEN
         clr = VCC;
      ELSE
         clr = GND;
      END IF;
   ELSE GENERATE
      timer.clk = clk & in;
      timer.ena = detector.out;
      timer.clr = !in;

      IF timer.q[] == B"10" THEN
         detector.clrn = VCC;
      ELSE
         detector.clrn = GND;
      END IF;
   END GENERATE;
END;
```

Appendix B – Simulation Results

B.1 Fields of the Simulation Results

The fields that appear in the simulation results shown in the next section are:

clock: The clock signal of 100Mhz applied to all the units.

Load: The signal used to load a program into memory, the load signal is accompanied by an 'address' on the address line and an 'instruction word' on the data bus.

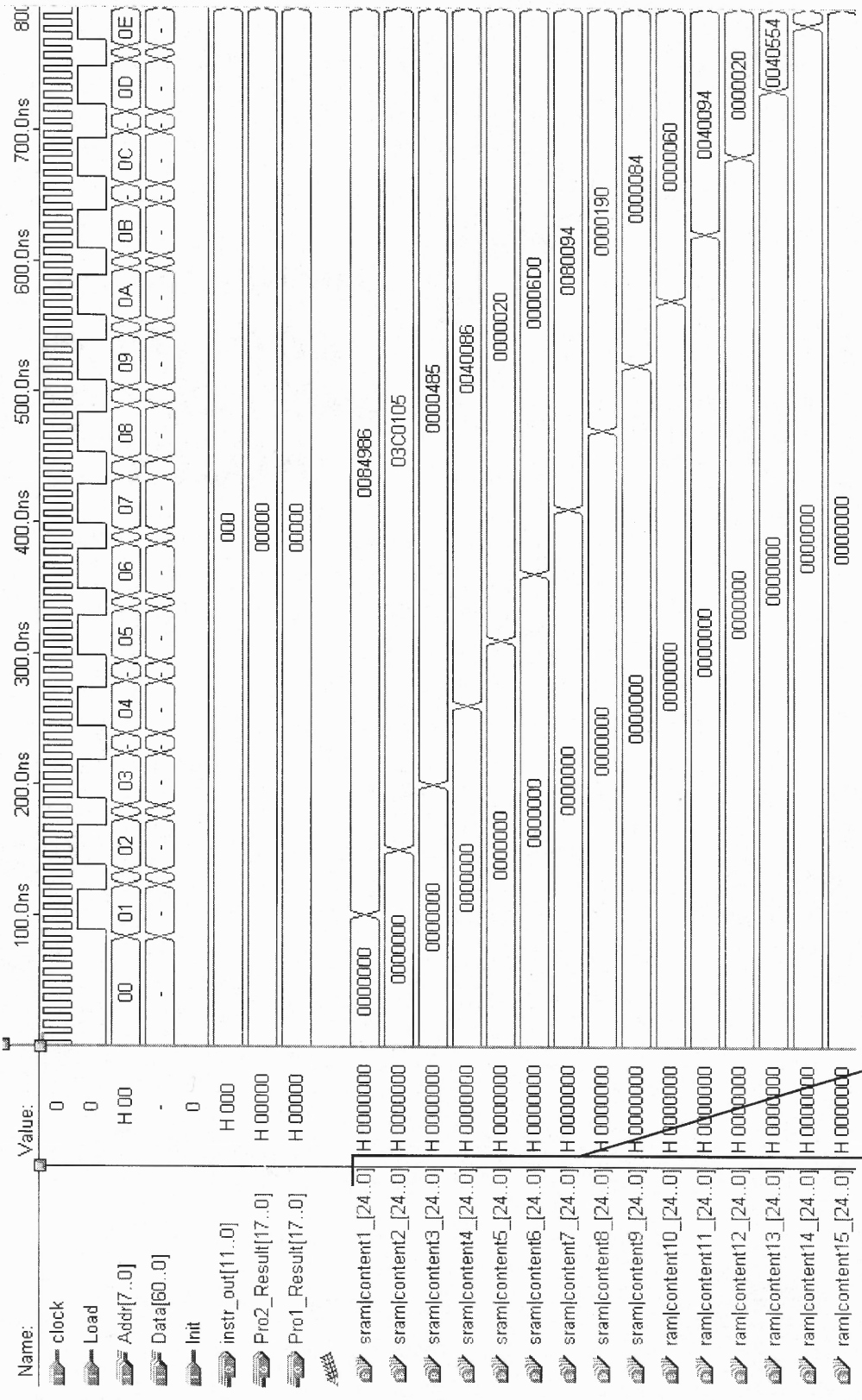
Addr[7..0]: Address lines used to specify the address, where 'instruction words' are loaded into memory. There are eight address lines, which allow the addressing the 256 locations in a block.

Data[60..0]: Data lines used to load 'instruction words' into memory. There are 61 data lines which is the same as the width of an 'instruction word', i.e. CS1(25-bits), CS2(12-bits), CS3(12-bits), and CS4 (12-bits).

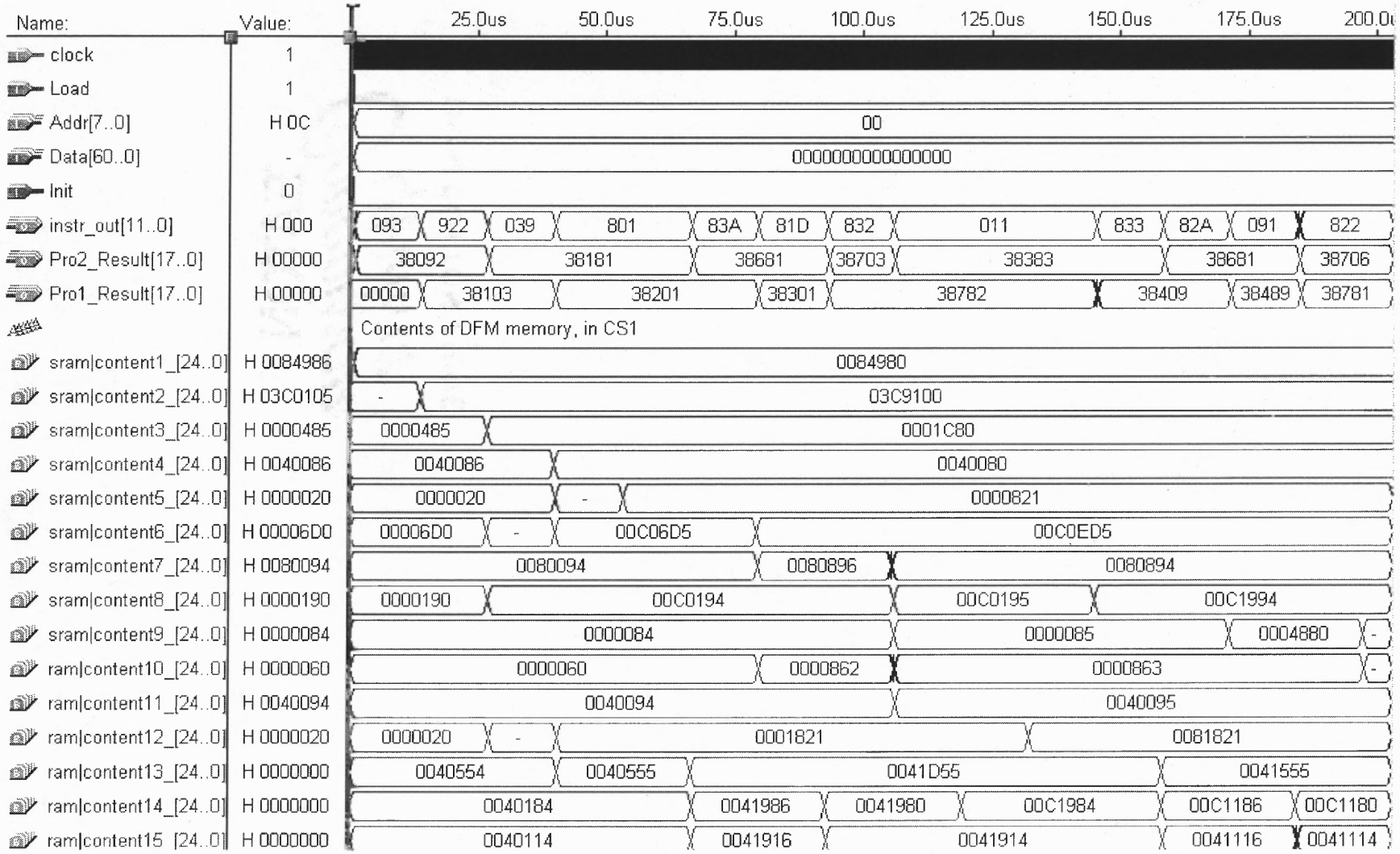
Pro2_Result[17..0] and Pro1_Result[17..0]: The results sent out by each processor after it finishes executing the executable it picks up from its memory bank. Each result sent out is 18-bits wide, 11-bits of OA and 7-bits of result.

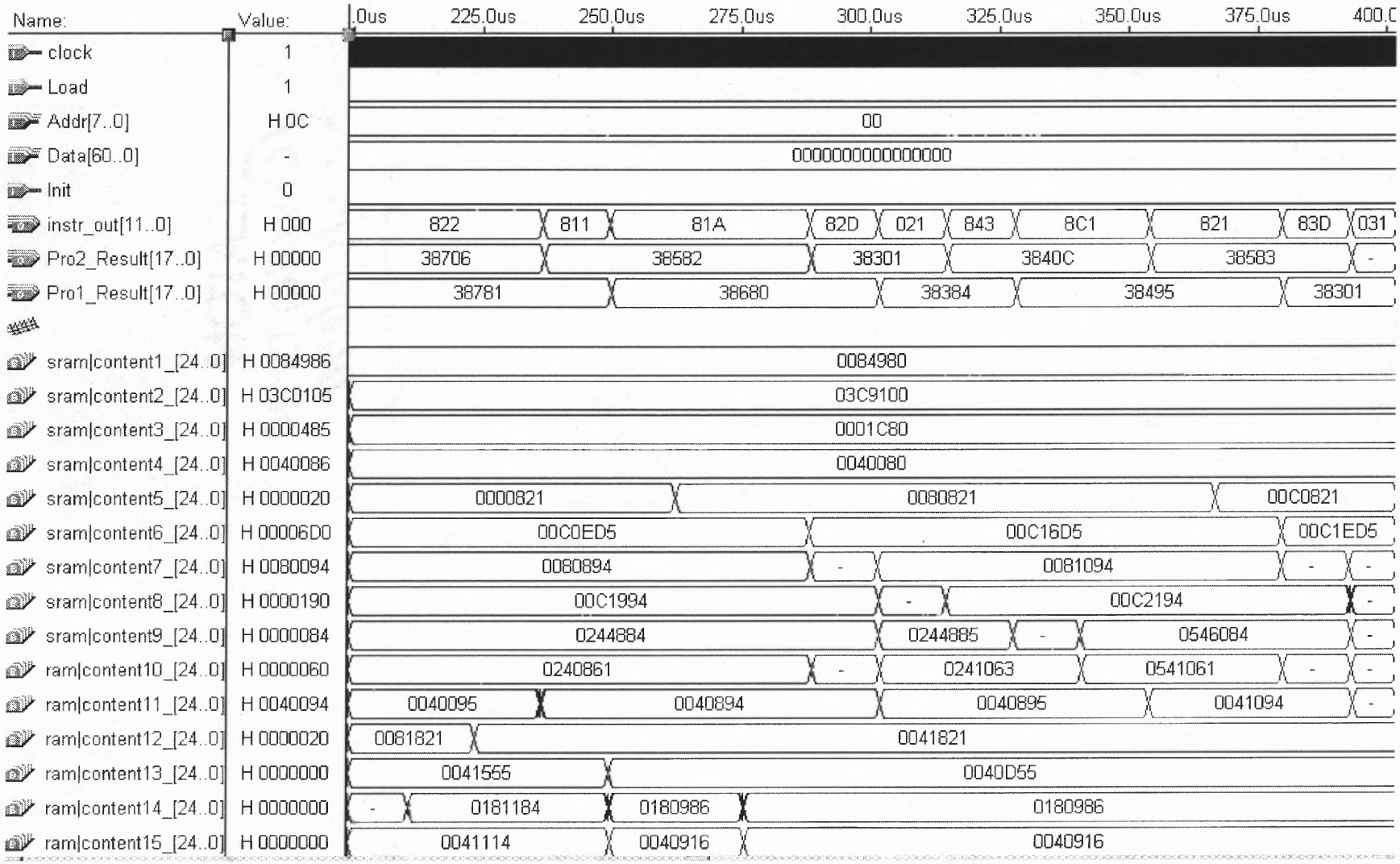
|block_cs1b:123|lpm_ram_dp:cs1|altdpram:sram|content1_[24..0] - |block_cs1b:123|lpm_ram_dp:cs1|altdpram:sram|content15_[24..0]: The memory contents of CS1 from location 1 to 15. This set of words allows viewing the changes occurring in CS1 memory as the program is executed.

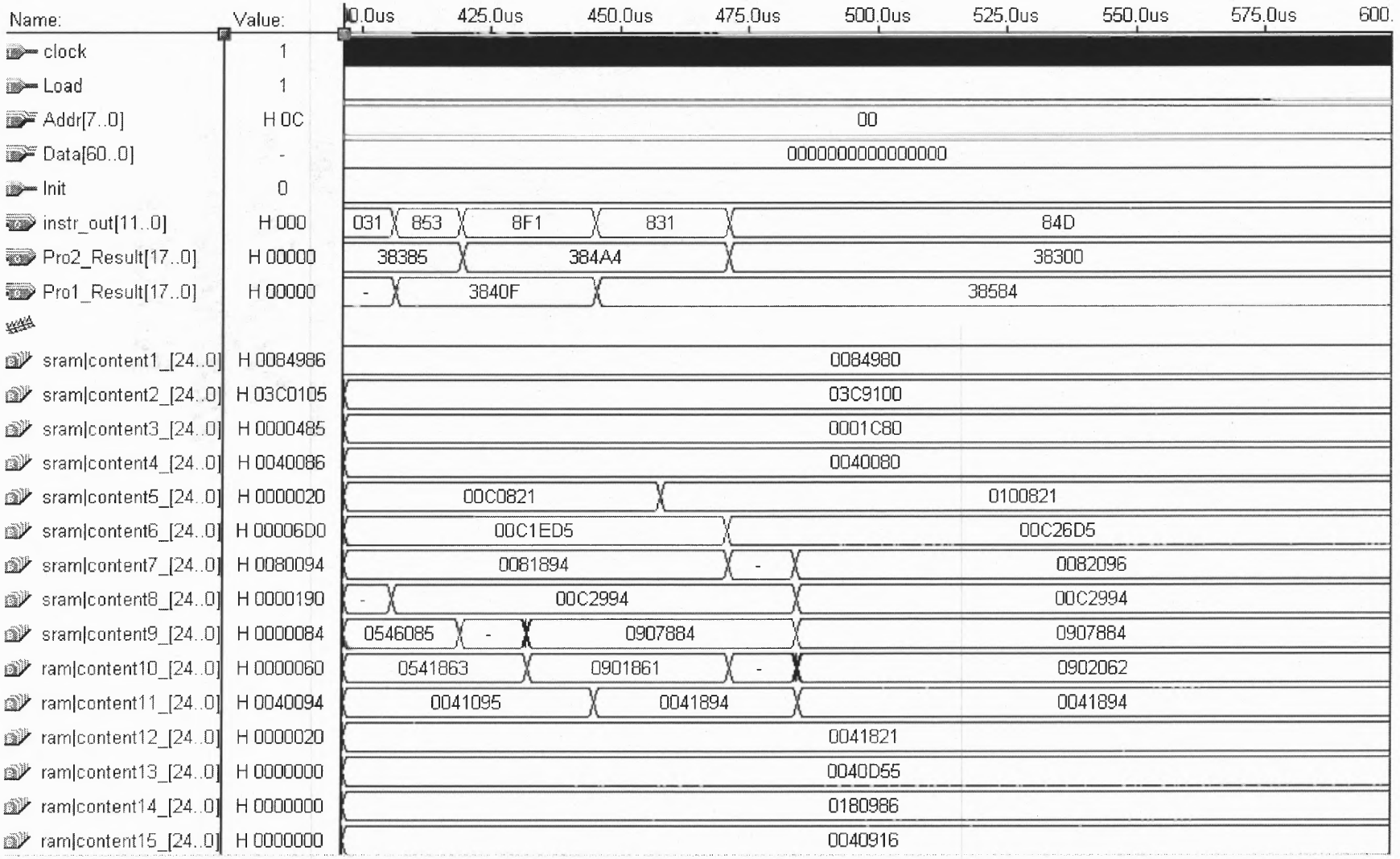
B.2 Program 1 Simulation Results



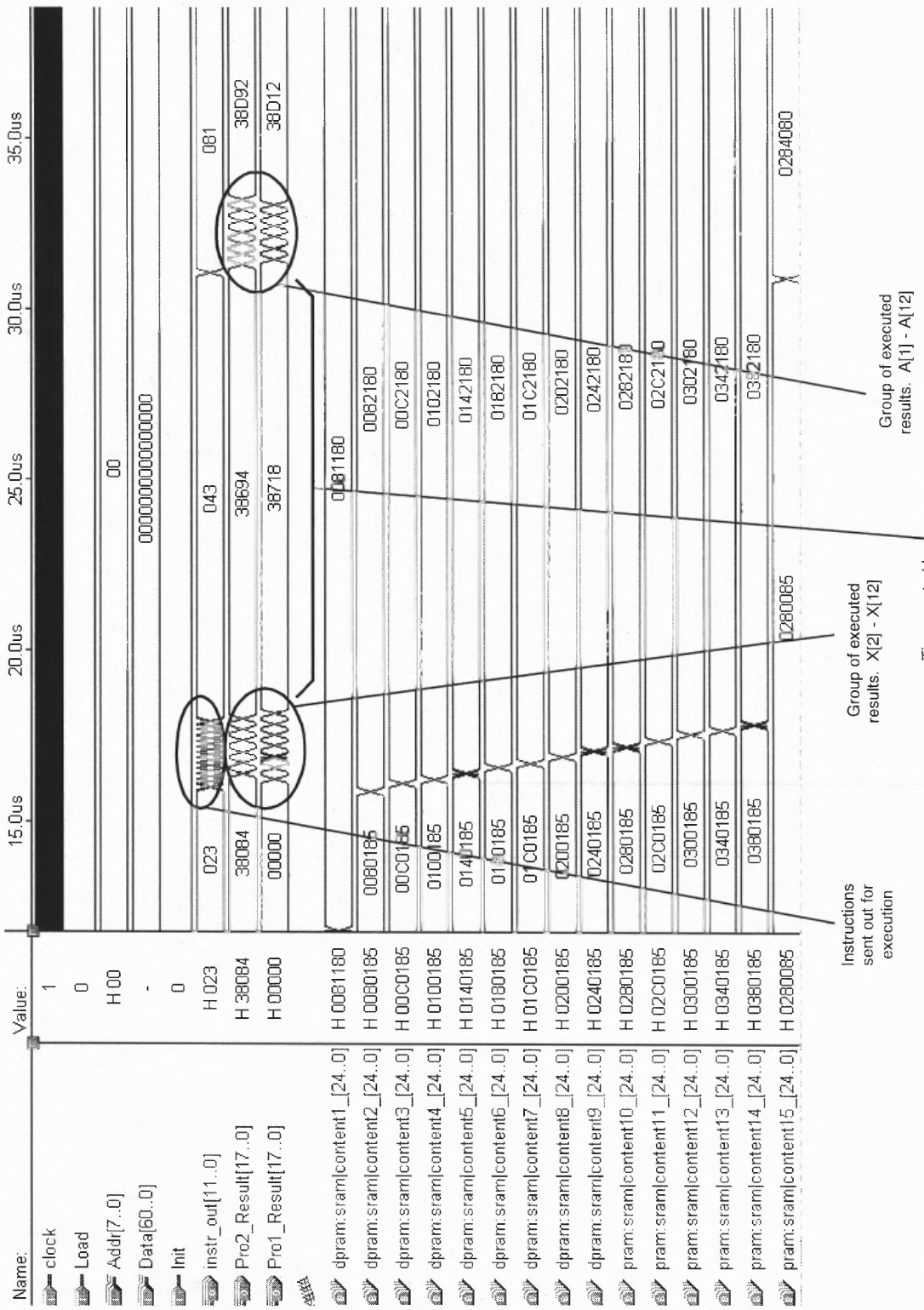

Name:
Value:
800.0ns
900.0ns
1.0us
1.1us
1.2us
1.3us
1.4us
1.5us
clock
1
Load
1
Addr[7..0]
H 0C
0F
00
Data[60..0]
-
0000000000000000
Init
0
instr_out[11..0]
H 000
000
093
Pro2_Result[17..0]
H 00000
00000
Pro1_Result[17..0]
H 00000
00000
sram|content1_[24..0]
H 0084986
0084986
0084980
sram|content2_[24..0]
H 03C0105
03C0105
sram|content3_[24..0]
H 0000485
0000485
sram|content4_[24..0]
H 0040086
0040086
sram|content5_[24..0]
H 0000020
0000020
sram|content6_[24..0]
H 00006D0
00006D0
sram|content7_[24..0]
H 0080094
0080094
sram|content8_[24..0]
H 0000190
0000190
sram|content9_[24..0]
H 0000084
0000084
ram|content10_[24..0]
H 0000060
0000060
ram|content11_[24..0]
H 0040094
0040094
ram|content12_[24..0]
H 0000020
0000020
ram|content13_[24..0]
H 0000000
0040554
ram|content14_[24..0]
H 0000000
0040184
ram|content15_[24..0]
H 0000000
0040114


Name:
Value:
25.0us
50.0us
75.0us
100.0us
125.0us
150.0us
175.0us
clock
1
Load
1
Addr[7..0]
H 0C
00
Data[60..0]
-
0000000000000000
Init
0
instr_out[11..0]
H 000
093
922
039
801
83A
81D
832
011
833
82A
091
822
Pro2_Result[17..0]
H 00000
38092
38181
38681
38703
38383
38681
38706
Pro1_Result[17..0]
H 00000
00000
38103
38201
38301
38782
38409
38489
38781
Contents of DFM memory, in CS1
sram|content1_[24..0]
H 0084986
0084980
sram|content2_[24..0]
H 03C0105
-
03C9100
sram|content3_[24..0]
H 0000485
0000485
0001C80
sram|content4_[24..0]
H 0040086
0040086
0040080
sram|content5_[24..0]
H 0000020
0000020
-
0000821
sram|content6_[24..0]
H 00006D0
00006D0
-
00C06D5
00C0ED5
sram|content7_[24..0]
H 0080094
0080094
0080896
0080894
sram|content8_[24..0]
H 0000190
0000190
00C0194
00C0195
00C1994
sram|content9_[24..0]
H 0000084
0000084
0000085
0004880
-
ram|content10_[24..0]
H 0000060
0000060
0000862
0000863
-
ram|content11_[24..0]
H 0040094
0040094
0040095
ram|content12_[24..0]
H 0000020
0000020
-
0001821
0081821
ram|content13_[24..0]
H 0000000
0040554
0040555
0041D55
0041555
ram|content14_[24..0]
H 0000000
0040184
0041986
0041980
00C1984
00C1186
00C1180
ram|content15_[24..0]
H 0000000
0040114
0041916
0041914
0041116
0041114


Name:
Value:
0us
225.0us
250.0us
275.0us
300.0us
325.0us
350.0us
375.0us
400.0
clock
1
Load
1
Addr[7..0]
H 0C
00
Data[60..0]
-
0000000000000000
Init
0
instr_out[11..0]
H 000
822
811
81A
82D
021
843
8C1
821
83D
031
Pro2_Result[17..0]
H 00000
38706
38582
38301
3840C
38583
Pro1_Result[17..0]
H 00000
38781
38680
38384
38495
38301
sram|content1_[24..0]
H 0084986
0084980
sram|content2_[24..0]
H 03C0105
03C9100
sram|content3_[24..0]
H 0000485
0001C80
sram|content4_[24..0]
H 0040086
0040080
sram|content5_[24..0]
H 0000020
0000821
0080821
00C0821
sram|content6_[24..0]
H 00006D0
00C0ED5
00C16D5
00C1ED5
sram|content7_[24..0]
H 0080094
0080894
0081094
sram|content8_[24..0]
H 0000190
00C1994
00C2194
sram|content9_[24..0]
H 0000084
0244884
0244885
0546084
ram|content10_[24..0]
H 0000060
0240861
0241063
0541061
ram|content11_[24..0]
H 0040094
0040095
0040894
0040895
0041094
ram|content12_[24..0]
H 0000020
0081821
0041821
ram|content13_[24..0]
H 0000000
0041555
0040D55
ram|content14_[24..0]
H 0000000
0181184
0180986
0180986
ram|content15_[24..0]
H 0000000
0041114
0040916
0040916


Name:
Value:
0.0us
425.0us
450.0us
475.0us
500.0us
525.0us
550.0us
575.0us
600.
clock
1
Load
1
Addr[7..0]
H 0C
00
Data[60..0]
-
0000000000000000
Init
0
instr_out[11..0]
H 000
031
853
8F1
831
84D
Pro2_Result[17..0]
H 00000
38385
384A4
38300
Pro1_Result[17..0]
H 00000
-
3840F
38584
sram|content1_[24..0]
H 0084986
0084980
sram|content2_[24..0]
H 03C0105
03C9100
sram|content3_[24..0]
H 0000485
0001C80
sram|content4_[24..0]
H 0040086
0040080
sram|content5_[24..0]
H 0000020
00C0821
0100821
sram|content6_[24..0]
H 00006D0
00C1ED5
00C26D5
sram|content7_[24..0]
H 0080094
0081894
-
0082096
sram|content8_[24..0]
H 0000190
-
00C2994
00C2994
sram|content9_[24..0]
H 0000084
0546085
-
0907884
0907884
ram|content10_[24..0]
H 0000060
0541863
0901861
-
0902062
ram|content11_[24..0]
H 0040094
0041095
0041894
0041894
ram|content12_[24..0]
H 0000020
0041821
ram|content13_[24..0]
H 0000000
0040D55
ram|content14_[24..0]
H 0000000
0180986
ram|content15_[24..0]
H 0000000
0040916

B.3 Program 3 Simulation Results



REFERENCES

[1] Veen, Arthur H. "Dataflow Machine Architecture." ACM Computing Survey 18.4 (1986): 365-396.

[2] Rau, B. Ramakrishna. "Cydra 5 Directed Dataflow Architecture." Thirty-Third IEEE Computer Society International Conference, Digest of Papers. (1988): 106-113.

[3] Dennis, Jack B., and Guang R. Gao. "An Efficient Pipelined Dataflow Processor Architecture." Proceedings of Supercomputing '88. [Vol.1]. (1988): 368-373.

[4] Sakai, Shuichi., Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. "An Architecture of a Dataflow Single Chip Processor." Proceedings of the 16th annual international symposium on Computer architecture. (1989): 46-53.

[5] Deshmukh, R. G., and Tariq Jamil, "A Novel Technique for Parallel Computations Using Associative Dataflow Processor." Proceedings of Southeastcon. Visualize the Future, IEEE. (1995): 322-328.

[6] Lu, Shih-Lin., and Chi-Ming Chang. "Modelling of a Selftimed DataFlow Processor in VHDL." Proceedings of the Sixth Annual IEEE International ASIC Conference and Exhibit. (1993): 228-231.

[7] Dadda, Luigi. "The Evolution of Computer Architecture." Proceedings of the 5th Annual European Computer Conference. (1991): 9-16.

[8] Robic, Borut, and Jurij Sile. "MADAME – Macro-Dataflow Machine." Proceedings of the 6th Mediterranean Electrotechnical Conference. (1991): 985-988.

[9] Grafe, V. G., and J. E. Hoch. "Implementation of the εpsilon Dataflow Processor." Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences. (1990): 19-29.

[10] Traub, Kenneth R., Gregory M. Papadopoulos, Michael J. Beckerle, and James E. Hicks. "Overview of the Monsoon Project." Proceedings of the IEEE International Conference on VLSI in Computers and Processors. (1991): 150-155.

[11] Grafe, V. G., G. S. Davidson, J. E. Hoch, and V. P. Holmes. "The εpsilon Dataflow Processor." Proceedings of th 16th Annual International Symposium on Computer Architecture. (1989): 36-45.

[12] Ang, Boon Seong. "Efficient Implementation of Sequential Loops in Dataflow Computation." Proceedings of the Conference on Functional Programming Languages and Computer Architecture. (1993): 169-178.

[13] Dennis, Jack B., and David P. Misunas. "A Preliminary Architecture for a Basic Dataflow Processor." 25 years of the International Symposia on Computer Architecture (Selected Papers). (1998): 125-131

[14] Ingersoll, Segreen, and Sotirios Ziavras. "Dataflow Computation With Intelligent Memories Emulated on Field-Programmable Gate Arrays (FPGAs)." VLSI Design Journal, submitted for publication, December 2000.

[15] Altera Corporation. MAX + PLUS II Getting Started. California: Altera Corp., 1997.

[16] Altera Corporation. MAX + PLUS II AHDL. California: Altera Corp., 1997.

[17] Altera Corporation. FLEX 10KE Embedded Programmable Logic Family Data Sheet. Sep. 2000. Oct. 2000 <http://www.altera.com/document/ds/dsf10ke.pdf>.

[18] Altera Corporation. ACEX 1K Programmable Logic Family Data Sheet. April 2000. Oct. 2000 <http://www.altera.com/document/ds/acex.pdf>.

[19] Altera Corporation. MAX 9000 Programmable Logic Family Data Sheet. July 1999. Oct. 2000 <http://www.altera.com/document/ds/m9000.pdf>.

[20] Hwang, Kai. Advanced Computer Architecture: Parallelism, Scalability, Programmability. San Francisco: Mc Graw Hill, 1993. 475-539.

[21] Hatano, Daryl. "Statement of Daryl Hatano." Hearing Archives 26 Jan. 2000. 18 Nov. 2000 <http://www.workforce21.org/archive_ca_hatano.htm>.

[22] Baeverrud, Rune. Parameterized Functions Made Simple. 17 June 2000 <http://www.freecore.com/nosupport/param1.htm>.