AN ABSTRACT OF THE THESIS OF

Chih-Ming Chang for the degree of Master of Science in Electrical and Computer Engineering presented on September 24, 1993.

Title: Micro Data Flow Processor Design

Redacted for Privacy

Abstract approved: _____
Shih-Lien Lu

Computer has evolved rapidly during the past several decades in terms of its implementation technology; it's architecture, however, has not changed dramatically since the von Neumann computer(control flow) model emerged in the 1940s. One main reason is that the performance for this kind of computers was able to satisfy the requirement of most users. Another reason maybe that the engineers who designed them are more familiar with this model. However, recent solutions to the problem of parallelizing sequential nature instructions on a von Neumann machine complicate both the compiler and the controller design. Therefore, another computer model, namely the data flow model, has regained attention since this model of computation exposes parallelism inherent in the program naturally.

In terms of implementation methodology, we currently use synchronous sequential logic, which is clock controlled for synchronization within circuits. This design philosophy becomes hard to follow due to the occurrence of clock skew as the clock frequency goes higher and higher. One way to eliminate these clock related problems is to use the self-timed(asynchronous) implementation methodology. It features advantages such as free of clock-skew, low power consumption, composibility and so forth.

Since data flow(data–driven) computation model provides the execution of instructions asynchronously, it is natural to implement a data flow processor using self–timed circuits.

In this thesis, micropipelines, one of the self–timed implementation methodology, is used to implement a preliminary version of general purpose static data flow processor. Some interesting observations will be addressed in this thesis. An example program of general difference recursive equation is given to test the correctness and performance of this processor. We hope to gain more insight on how to design and implement self–timed systems in the future.

MICRO DATA FLOW PROCESSOR DESIGN

by

Chih–Ming Chang

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 24, 1993

Commencement June, 1994

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented September 24, 1993

Typed by Shu–Hui Liu for Chih–Ming Chang

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

MICRO DATA FLOW PROCESSOR DESIGN

CHAPTER 1

INTRODUCTION

Computer has evolved rapidly during the past several decades in terms of its implementation technology, starting from relays, vacuum tubes, transistors, Integrated Circuit(IC), Large–Scaled IC(LSI), to today's Very Large–Scaled IC(VLSI) and even Ultra Large–Scaled IC(ULSI). It's architecture, however, has not dramatically changed since the von Neumann computer model emerged in the 1940s. This type of model, basically, is called the control flow model. Although the potential problem of the von Neumann model was reported by Backus in 1978[1], the von Neumann–type computers still dominate the market up to now. This is because the performance for this kind of computers was able to satisfy users' requirement. Moreover, the computer engineers who designed them are familiar with this model. Recently, the von Neumann–type computers have been reviewed and some hard–to–solve problems have been addressed(we will discuss these problems in more detail later)[5][6][7][8]. Therefore, another computer model, namely the data flow model, has regained attention from many researchers. This model of computation has many advantages. We will briefly describe its features in the later sections.

Between the implementation technology and the architecture of a computer design, there is another area namely the timing methodology which has been neglected. It's assumed the synchronous approach is the unmatchable method in designing computer systems. Only until recently, the asynchronous(self–timed) methodology was able to redeem its reputation. Good progress has been reported both in theory and practice. For example, lately, a self–timed implementation technique, called micropipelines, has been used to realize a Reduced Instruction Set Computer(RISC) microprocessor[13].

This technique, however, has not yet been employed in implementing a processor based on the data flow computer model. This research addresses the design of a self–timed static data flow processor. The rest of this chapter covers the problems that control flow computers have. Also, we examine how a data flow computer can potentially avoid these problems and increase program execution parallelism. Chapter 2 describes the basic data flow concept and a preliminary general–purpose data flow computer. The self–timed implementation technique, micropipelines, is recounted in chapter 3. Chapter 4 shows how a data flow computer is implemented using self–timed circuits based on micropipelines. In chapter 5, a logic simulation result is presented. The last chapter we draw some conclusions and suggest possible future works.

1.1 Control Flow Computers

The reason why the conventional von Neumann computer is called control flow computer is because the order of program execution is explicitly stated in the user's program from the software viewpoint, or governed by Program Counter(PC) from the hardware viewpoint. There are three general types of control flow computers.

(1) ***Uniprocessor***: A processor basically consists of a Central Processing Unit(CPU), a memory and a connecting tube that can transmit data and instructions between CPU and memory. A CPU can be subdivided into a centralized controller and an Arithmetic/Logic Unit(ALU). The von Neumann bottleneck occurs due to this connecting tube[1]. Moreover, the program is executed ***sequentially*** according to the PC, therefore, one instruction is fetched, executed and written back before the next instruction can be fetched. This kind of computer has its performance limit because of its architectural nature.

(2) ***Pipelined Processor***: A uniprocessor with pipelining allows multiple instruction execution in overlapped manner. This kind of computers potentially can

run K times faster than that of the uniprocessor without pipelining. Here K is the number of pipeline stages. However, allowing multiple instructions to overlap brings up the problems of hazards: structural hazards, data hazards and control hazards[2][3][4]. To obviate hazards, the hardware complexity has been increased greatly. This becomes obvious when out of order instruction issue(completion) is allowed. It is because the dynamic scheduling needs to trace all possible ***dependences*** among instructions in order to produce the correct result[5]. Interrupt will make the situation even worse especially when precise interrupt is required[6].

(3) *Multiprocessors*: The maximum parallelism in a program execution can be achieved using von Neumann style multiprocessors. This approach, however, has two major problems. ***First***, it is inefficient to wait long latencies for memory requests from each processing element. ***Second***, it is hard to achieve unconstrained, yet synchronized, access to shared data for share memory style multiprocessors(tightly coupled multiprocessors)[7]. In order to explore the dependences in a program, a data dependence graph can be constructed to show the fine grain parallelism. This fine grain parallelism can be exploited with the price of very large synchronization overhead. In general, however, the multiprocessor system can not afford this large overhead so a ***coarse*** grain of parallelism is always used[8]. In addition to the critique, the program partitioning, scheduling, and the cache coherence problems complicate the implementation of compiler and controller and, of course, the hardware complexity.

1.2 Control Flow vs. Data Flow Computers

With the discussion of control flow computers above, some conclusions can be made.

(1) The control flow program is inherently sequential in execution. The "side effect" of updating a variable always confines the potential parallelism.

(2) Compiler will be complex if we are to explore the parallelism in a program written with procedure languages. To execute a sequential program in parallel and maintain correct dependency, a complicated controller hardware is expected.

(3) Only coarse grain parallelism can be effectively and efficiently executed when taking into account the synchronization overhead.

An alternative to avoid all the above problems is to use the data flow instead of the control flow model.

In a data flow computer, the execution of an instruction is determined by the data availability. Instead of executing instructions followed by a program counter, all instructions enabled are allowed to be executed simultaneously. Therefore, the instructions are not ordered in any way. All the instructions should be ready for execution in any time as long as the data(operands) become available. The results of previous instructions are directly passed to the "next" instructions without temporary storages. Here the "next" is in terms of dependence not the consecutive entry in the address space. Since there are no temporary storages for computational results, the results are gone forever once they are consumed by certain instruction(s)[9]. No result reuse later is possible. If more than one instruction needs data(results) from previous instruction, duplication of the result is necessary.

With the understanding of previous discussion, we summarize the main features of data flow computers.

(1) ***No program counter***: Since instruction execution is driven by data availability, no program counter is required. This feature releases the program from sequential execution. The execution parallelism can, therefore, be increased.

(2) ***No central controller***: A central controller is not suitable for data flow computers because the degree of parallelism depends on "local" data only. Here "local" means that the data come from previous instructions only. As long as the operands become available for an instruction, this instruction can be executed. No global depen-

dence analysis among instructions is necessary. It is, however, required in a control flow computer especially when out of order execution is allowed. Global controller complicates the design of a data flow computer. A special mechanism, however, is required to detect the data availability and to match data with needy instructions[10][11]. Local control is generally used for a data flow computer.

(3) ***No shared memory*****:** In a pure data flow computer, memory is not required since there is no need to save temporary results. There is a space for program storage, however, the way instructions are executed is different from the conventional way. An instruction is fetched by demand after all the operands are available. Without a shared data memory, many problems facing the control flow multiprocessors can be eliminated.

1.3 Implementation Advantages For Data Flow Computers

We have briefly mentioned the advantages of data flow computers in terms of architecture. In this section, we will introduce the advantages of self–timed implementation methodology. Since data–driven computation model provides the execution of instructions asynchronously, it is natural to implement a data flow processor using asynchronous(or self–timed) circuits.

The advantages of the self–timed circuits are as follows[12][13]:

(1) ***Clock skew free*****:** The clock skew problem becomes hard to handle for the computer engineers when the clock frequency is increased. In a conventional computer system design, clock is distributed all over the system through a long line(tree) on printed circuit board. Due to the parasitic capacitances and inductances, the high frequency clock signal will be distorted and therefore skewed with control signals. This will limit the number of logic stages which can be safely placed between two pipe–stages, if a clocked pipeline structure is considered. This problem comes from

the nature of conventional design philosophy — ***global*** clock synchronization. Instead of using global clock, self–timed machines use ***local*** communication scheme(handshaking) between successive stages so they are entirely free from the clock skew problem. By eliminating global clock, considerable amount of hardware can be saved. For example, more than a quarter of the silicon is devoted to clock logic for the Alpha chip from Digital Equipment Corporation(DEC)[13]. If clock can be totally removed, this quarter silicon area can be used for the other functions to upgrade the chip.

(2) ***Noise reduction*****:** In a conventional clock–driven processor, all the activating transistors in the system switch at the same time when the global clock arrives. This leads to steep source current variation and results in high–voltage inductive noise in the power line. In a self–timed processor, data (tokens) arrive distributively along with time, thus the number of simultaneously switching transistors may be reduced dramatically. This reduces the power noise.

(3) ***Zero stand–by power*****:** If made of Complementary Metal–Oxide Semiconductor(CMOS), a self–timed processor consumes no power theoretically(in reality, small leakage current exists) when it is idling(no tokens are in a position to fire). On the contrary, a clock–driven processor consumes power even no tasks are being executed since the clock continues to operate. If a self–timed microprocessor is running under full(maximum) load(speed), the power consumption will be compatible to a synchronous microprocessor. In reality, however, the microprocessor is rarely running at the maximum speed in general applications, leading to less total power consumption. This characteristic is especially useful for portable devices.

(4) ***Low heat generation*****:** More power is consumed implying more heat is generated. In VLSI design, when the density is getting higher and higher or more specific the size reduces to 0.5 micro level and below, tens of millions to hundreds of millions of transistors could be contained in a single chip. With this kind of density, the heat dissipation is a serious problem. Although a static CMOS is employed

to save the power, a clocked microprocessor still intrinsically wastes the power and therefore dissipates the "extra" heat. As mentioned before, self–timed microprocessors consumes less power hence, less heat is generated.

(5) ***Modularity and composibility***: Self–timed circuit is also called delay–insensitive circuit because the signal delays between elements(modules) do not affect the correct operations. This feature enables self–timed circuits for ease of hierarchical design. A larger module can be constructed easily and efficiently by interconnecting smaller modules appropriately using a self–timed signalling protocol. This larger module will function correctly as long as the smaller modules are correct. No additional attention needs to be considered. In conventional clocking circuit design, composing two modules directly is disallowed in general since this might destroy the synchronization due to additional and unexpected delay. This kind of circuit is called delay–sensitive circuit, and it is not suitable for individual module design.

(6) ***Performance improvement***: In conventional clocking circuit design(eg. clocked pipeline structure), the clock period must be greater than the worst stage delay in order not to override some data. In self–timed circuit design, the current stage result can be transmitted to next stage as long as the next stage is ready to accept data. From the overall viewpoint, only average delay is seen instead of worst delay. This will speed up the throughput. As technology improves, some old–version module(s) can be replaced by new and fast version without redesigning the whole system by the thanks to composibility characteristics. This improves the whole system performance and hence enlongs the product market life with least price.

1.4 Summary

Control flow computers have three major problems. First, the side effect of updating a variable always confines the potential parallelism. Second, complex compiler and controller are expected if a sequential program would like to be executed in parallel and maintain correct dependency. Third, only coarse grain parallelism can be effectively and efficiently executed in taking into account the synchronization overhead. An alternative to avoid all the above problems is to use the data flow instead of control flow model. Due to the nature of data flow model, it can show all the potential parallelism in a program directly. Since data–driven computation model provides the execution of instructions asynchronously, it is natural to implement a data flow processor using asynchronous(or self–timed) circuits. Self–timed circuits provide the following advantages: clock skew free, noise reduction, zero stand–by power, low heat generation, modularity/composibility and performance improvement. With these advantages of self–timed circuits, this thesis is devoted to implement a data flow processor using self–timed methodology namely micropipelines.

CHAPTER 2
DATA FLOW PROCESSORS

Data flow processors provide an alternative to avoid the problems to which conventional von Neumann processors must face as stated in the previous chapter. In this chapter, basic concepts of data flow processors including the data flow graphs(languages) and the inherent parallelism within them are described. We also present the detail description of a particular static data flow processor. This processor is later implemented using micropipelines.

2.1 Primitives Of Data Flow Graphs

As mentioned in the last chapter, conventional control flow program is sequential in nature and has side effect such that the potential parallelism is confined. If we are to take the limit of parallelism off, both complex compiler and complicated hardware are expected. An alternative is to use functional language or called applicative language, which is suitable for programming data flow processors. A specific functional language for a data flow processor is called the data flow language.

There are properties of the data flow language, such as: freedom from side effect, locality of effect, single assignment rule and so on. In addition, they also feature modularity and composibility, which are nice characteristics for program expansion[9]. In terms of hardware implementation, because of the modularity and composibility resemblance between the data flow languages and self–timed circuits a special purpose self–timed processor(eg. Digital Signal Processing processor) can be readily obtained by direct mapping from data flow language to hardware. Data flow languages describe the data flowing from one function entity to another to activate the next instruction. This data flow concept in language suggests that the program can be

represented by graphs, called data flow graphs[14]. Graph representation let programmers efficiently and correctly visualize the unavoidable dependences and maximum potential parallelism.

Before a data flow graph is defined, eight basic elements(primitives) are described[15]. Of the eight primitives(two links and six actors) in Figure 1, ***data–link*** and ***control–link*** function as data copy and control copy, respectively. ***Operator*** is the arithmetic computation actor. ***Boolean operator*** is the logic computation actor(a complex logic decision can be built up with simpler decisions). In order to generate a control signal, ***decider*** is provided. Moreover, ***merge, true gate*** **and** ***false gate*** are included for the instruction control (loop, branch and iteration) purpose.

Two types of tokens — data tokens and control tokens — are fed to or generated by these primitives. A token, which conveys a value of either true or false and functions as control purpose, is called control token, such as a token fed to true/false gate from horizontal arc. The other type of token, which functions as data purpose, is called data token, such as tokens consumed by operator. In Figure 1, the solid arrows carry data tokens and hollow arrows deliver control tokens.

A data flow graph is a directed graph, whose nodes correspond to actors and arcs correspond to links. The arcs in a data flow graph are pointers for forwarding tokens. Thus the order of instruction execution is controlled by the flow of tokens, not by the instruction counter.

There are two types of data flow processors: one is static and the other one is dynamic. In terms of tokens flowing in a data flow graph, static data flow processors allow at most one token on each arc at any time; dynamic data flow processors have no restriction on the number of tokens flowing on each arc. Apparently, according to the definition, dynamic data flow processors show more parallelism than static data flow processors; however, in this thesis, only static processors are considered thereafter. The reason for this decision is because the concept of static data flow processors

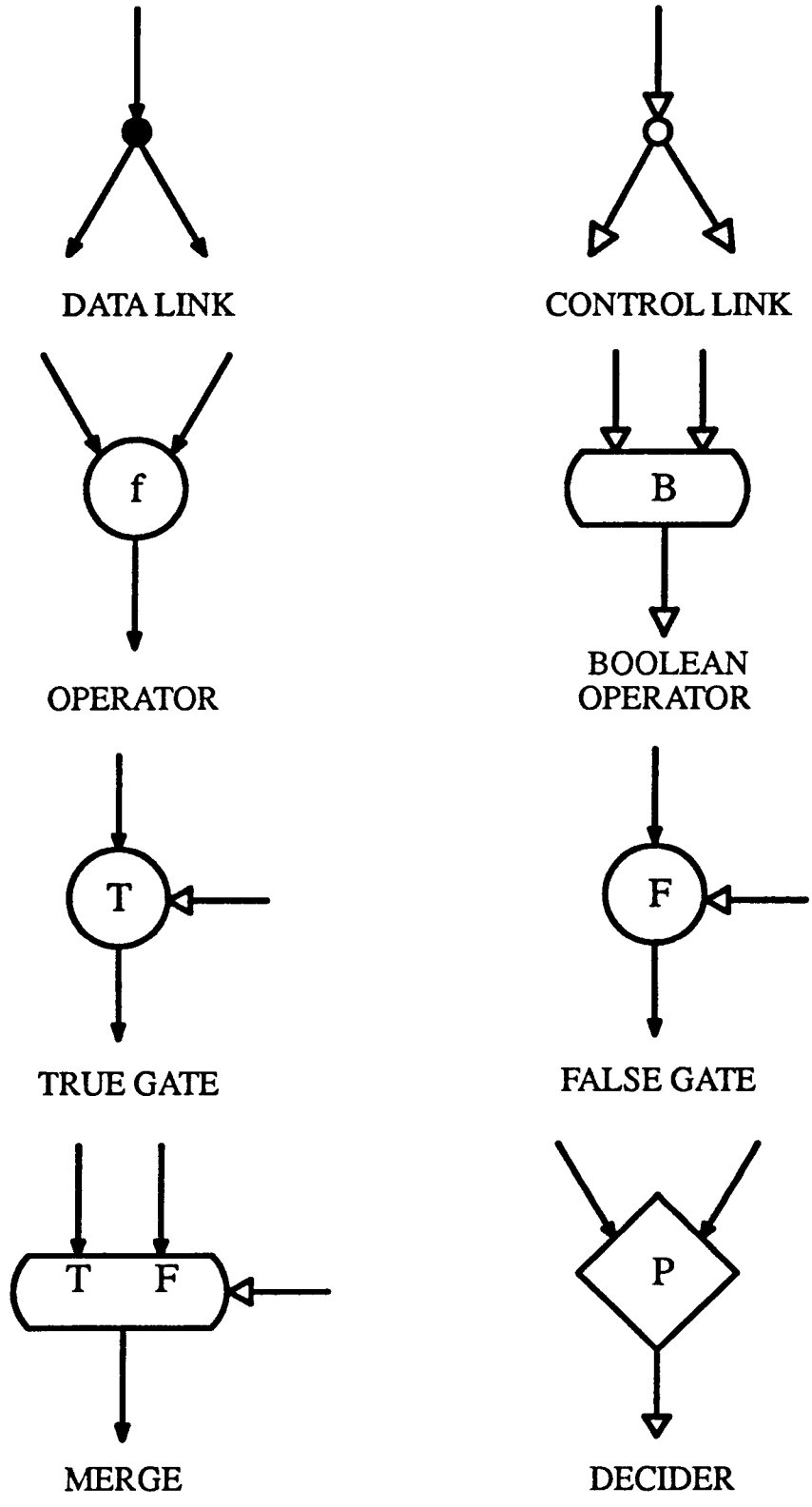


Figure 1: Primitives of a data flow graph

resemble the characteristics of micropipelines(see chapter 3) used as the implementation medium. We feel as a start, implementing a static processor help to gain critical experiences necessary for more ambitious design later.

A firing rule for a static data flow processor is described as follows. ***Firing Rule***: An instruction in a data flow graph is considered to be fired(enabled) when token exists on each of its input arcs and no token is present on its output arc. In general, when an instruction is fired, the tokens on its input arc are consumed and an output token is generated on its output arc. There are exceptions — merge and true/false actors. Merge is fired when no token on output arc and when (1) a true control token arrives and a data token is present at the true arc or (2) a false control token arrives and a data token is present at the false arc. The data tokens of counterparts in case(1)and (2)(if any) will remain intact. For the true(false) gate, if the control token is false(true) the input tokens are consumed and no output token is generated.

2.2 Parallelism In Data Flow Graphs

In terms of topology, there are two types of data flow graphs: one is acyclic and the other one is cyclic. Acyclic data flow graph is a graph containing no feedback loop. On the contrary, a cyclic data flow graph is a graph with at least one feedback loop. In an acyclic data flow graph, parallelism can be explored in two fashions: spatial parallelism and pipelining. It is best to illustrate these parallelism with an example, as shown in Figure 2(a). This Figure performs the following expressions:

Let U = a + b

Let V = c – d

Let W = e * f

Let X = (U * V)/(V – W)

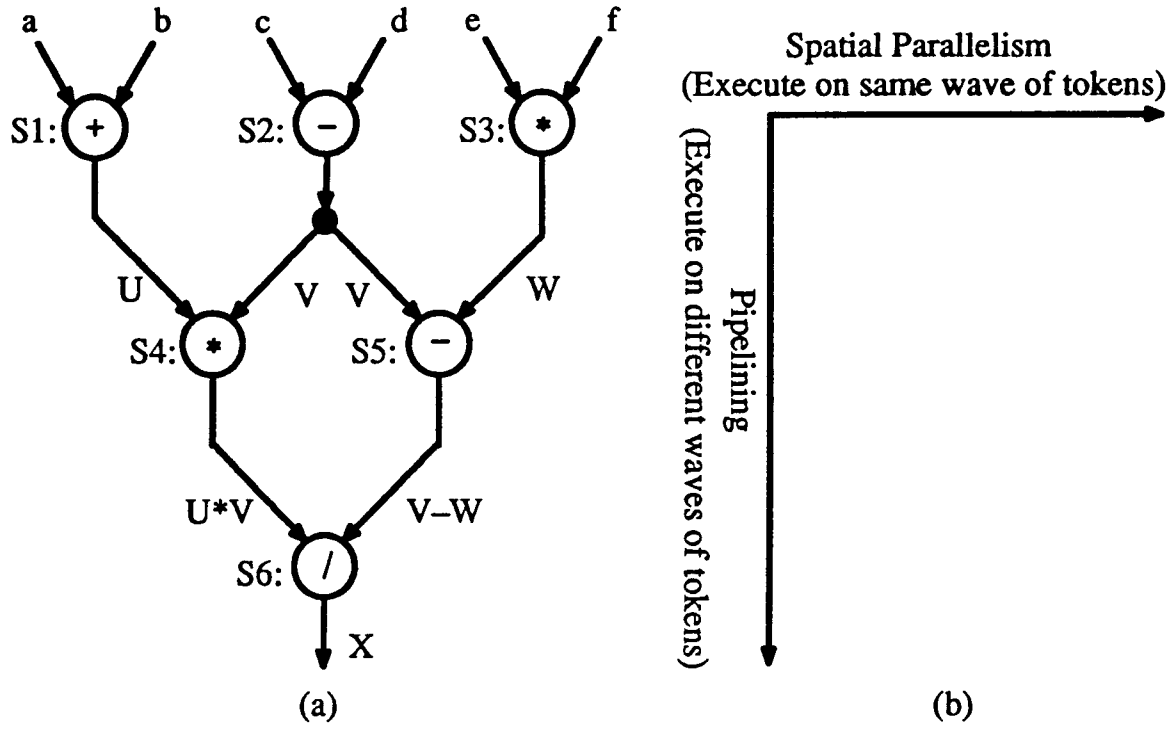


Figure 2: An acyclic data flow graph (a) graph itself
(b) inherent parallelism within graph

Spatial parallelism means the parallelism among nodes(instructions) which may be potentially executed in parallel unless there is an explicit data dependence among them. According to this definition, the nodes within the same horizontal level can be concurrently fired so parallel execution of instructions S1, S2 and S3 denotes the spatial parallelism.

Pipelining can be viewed as the parallelism in vertical direction in data flow graphs. It can be explained in two ways depending on how we define the delays of nodes. If we assume no latch for each input arc and no computational delays for the nodes in a data flow graph, then only the alternate instructions can execute concurrently(eg. S1 and S6). This is because at most one token is allowed on each arc according to the static firing rule. Token generated on an output arc(eg. U) must be consumed(eg. by S4) before the next one can be generated. In reality, delays

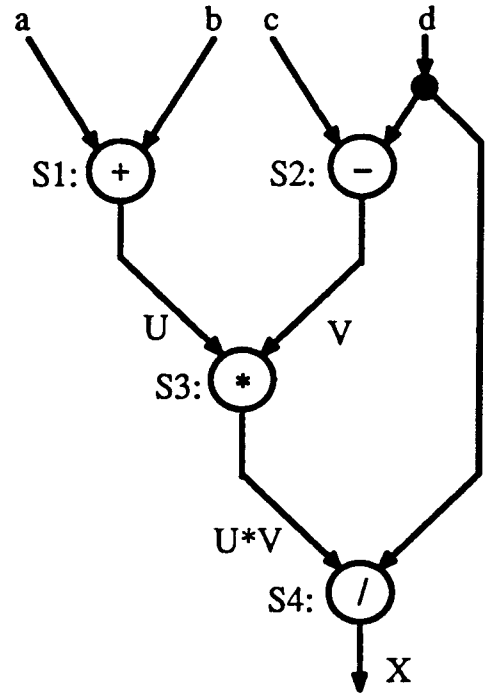
occur for all the instructions(implying each input arc with latch) so all the instructions in vertical direction(eg. S1, S4 and S6) can be executed simultaneously. Note that S1, S4 and S6 execute on different waves of tokens(this is why we called the vertical direction "pipelining"). This is different from the spatial parallelism which executes on the same wave of tokens, as shown in Figure 2(b).

Although the parallelism among a data flow program is easily explored, some frequent–occurrence graphs will greatly reduce the parallelism. The unbalanced graph is one example and the cyclic graph is the other example. In Figure 3(a), we depict a graph with left path and right path(an arc only) unbalanced so pipelining parallelism will be lost if only one token is allowed on each arc. If we want to preserve pipelining, tokens must be accumulated on right path, which violates the firing rule of static data flow processors. This is more like sequential execution in vertical direction. Spatial parallelism, however, is still preserved. In order to regain pipelining, two identity(I) nodes can be artificially added to balance the left and right paths with the price paid by more instruction numbers in total, as shown in Figure 3(b). Identity node is a special case of operator primitive.

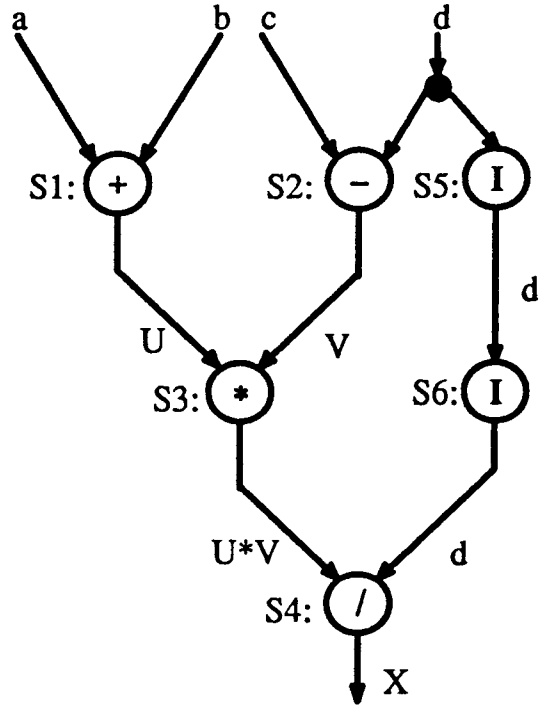
Figure 4 shows a cyclic data flow graph which represents the following expressions:

```
Let X = 0
Do X = 5 * X + X / 2
     Y = X * 5
Until Y > 200
Z = Y
```

Pipelining feature does not exist within this loop since merge is not executable until control token arrives. Each input token wave of X produces only one corresponding control token at decider(>) so the next token wave can not enter the loop until current one is finished, causing one token wave in a loop at any time instant which implies



(a)



(b)

Figure 3: A data flow graph (a) original form (b) after balancing

no pipelining. The loss of this kind of pipelining can not be regained by simply inserting some identities. The study of unfolding or partially unfolding this kind of feedback loop to increase the parallelism is an interesting topic for future research.

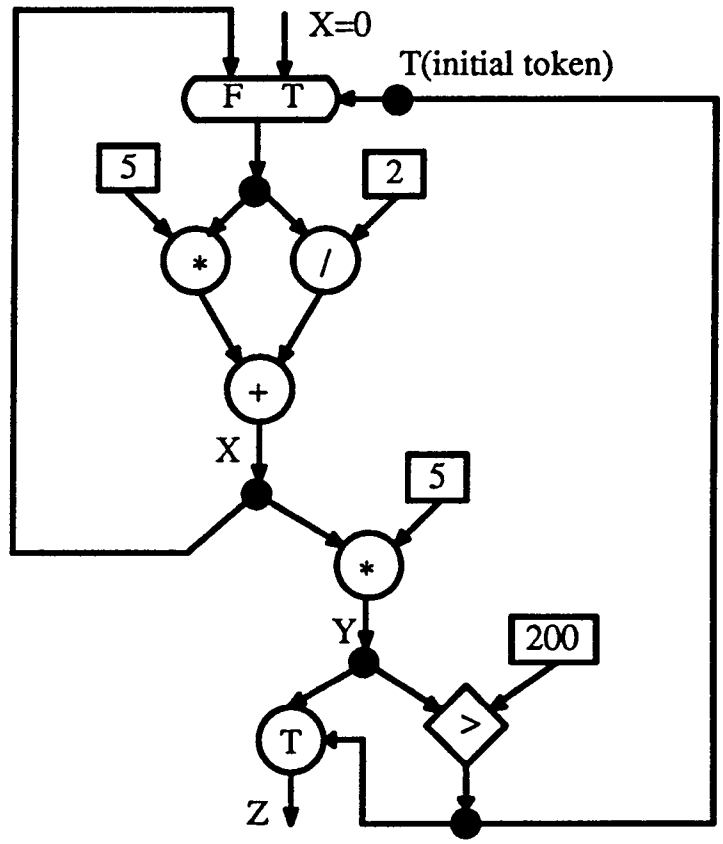


Figure 4: A cyclic data flow graph

2.3 A Preliminary Data Flow Processor

Jack B. Dennis proposed a preliminary architecture for a basic(general–purpose) data flow processor in 1975, as illustrated in Figure 5[15]. This processor can execute all the data flow programs(graphs) constructed by the primitives described in section 2.1.

A data flow program is stored in instruction memory which is organized into instruction cells. Basically, each cell holds a node of a data flow graph. The arbitration network is used to match the number of instruction cells with processing units(decision unit plus operation unit). As long as certain instruction(or instructions) satisfies the firing rule, it is enabled and signals the arbitration network that it is ready to transmit its contents as an operation packet to an operation unit or as a decision packet to a decision unit depending on instruction itself. Operation unit performs arithmetic operation and decision unit performs logic operation. The distribution network distributes data to the correct register according to the address specified in the data packet. Very likely situation can be applied to the control packet and the control network. This procedure will continue as long as there are instructions ready to be fired.

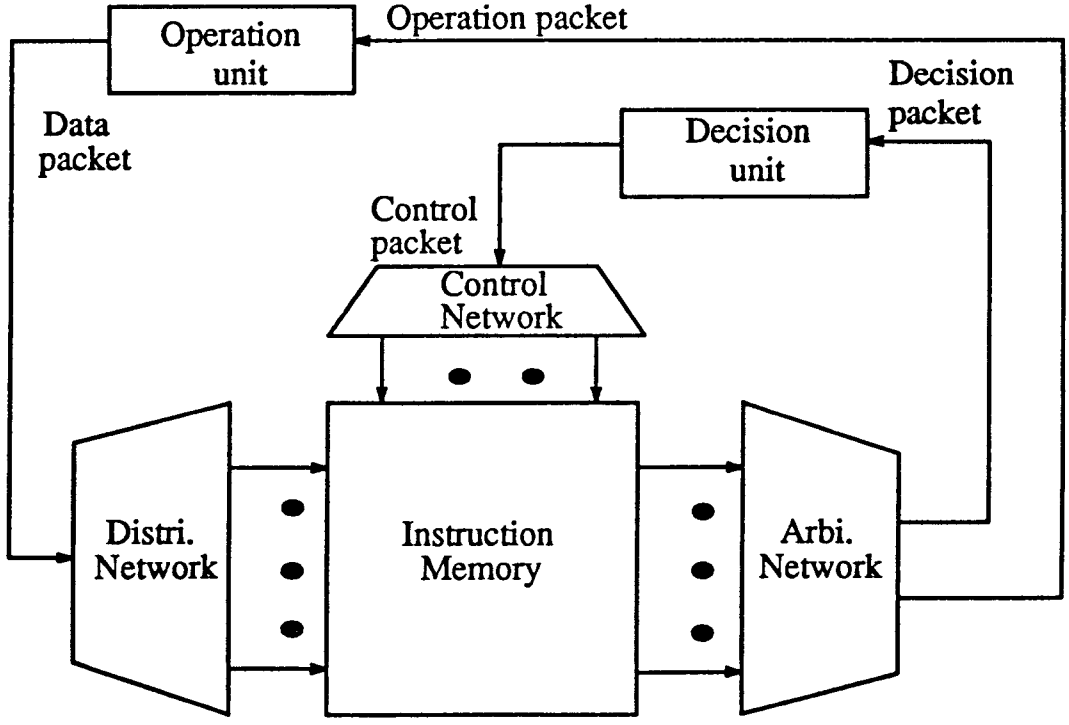


Figure 5: Dennis data flow architecture

2.3.1 Instruction Memory

In order to fully describe the token–flow behavior of a data flow graph, an instruction template for each instruction cell should include the following parameters such as: instruction codes, gating codes, gating flags, value flags, values, destination addresses and result tags, as shown in Figure 6.

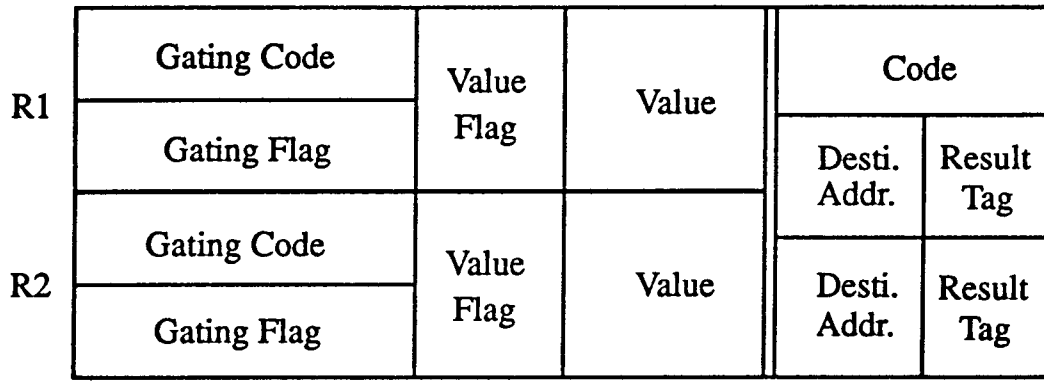


Figure 6: Instruction template

There are two registers, R1 and R2, for each instruction cell. Each arc in a data flow graph directs to one of these registers according to the destination address. Gating code indicates the properties of the previous instruction that arc connects with and the result tag indicates the types of control packet — value type or gate type. Value type control packet is considered as a data token for boolean operator and control type control packet is used as a control token for true/false gate and merge.

Among the whole primitives, only operator, boolean operator, decider and merge are implemented explicitly. Operator, boolean operator and decider are called active nodes(primitives) in a data flow graph. Merge has special operation rule so it should be considered separately in implementation. There are two fields of destination address in instruction template which are for the implementation of data link and

control link. True gate and false gate are implicitly implemented by integrating with active nodes. This is done by examining the gating code in the instruction template. Table 1 shows the gating code and its corresponding meaning.

Table 1: Gating code and its corresponding meaning

Gating Code	Meaning
Cons	The associated operand is a constant value
No	The associated operand is neither a true gate nor a false gate
True	The associated operand is a true gate
False	The associated operand is a false gate
True_m	True arc for merge
False_m	False arc for merge

If the gating code is ***Cons***, the corresponding register in instruction template is always enabled since data is always valid. The ***No*** gating code implies that the previous instruction connected with it is an operator, a boolean operator, a merge or a decider, not true nor false gate. Therefore, once data arrives, the corresponding register can be fired. For the ***true(false)*** gating code, the register becomes executable when both data token and true(false) control token arrive; otherwise, it can not be fired.

Gating flag in the instruction template represents the nature of a control token — ***Off***, ***True*** or ***False***. Gating flag equal to Off means that corresponding instruction does not require control token(signal) to be enabled, such as operator and decider. The True gating flag implies that the incoming control token has true value and the False flag denotes false value. Once the value flag is set(data arrived) and the gating

code and gating flag "***match***", the corresponding register is enabled. Here "match" means as follows:

(1) Cons gating code matches Off gating flag;

(2) No gating code matches Off gating flag;

(3) True/True_m gating code matches True gating flag;

(4) False/False_m gating code matches False gating flag.

Cons and No gating codes match Off gating flag since they do not need to wait control token in order to be fired(meaning no True/False actor connected with). It is obvious that True gating code matches True gating flag. This indicates a True control token arrives a True actor. True_m gating code also matches True gating flag since it implies the True control token arrives the True terminal of merge. The similar reasons can apply to the False/False_m gating code and False gating flag. When both registers are enabled, the corresponding instruction can be fired. For the merge, however, according to how it is implemented only one of these registers needs to be enabled before it can be executed(section 2.4 gives a more detail description of the merge).

2.3.2 Networks

There are two basic network elements(modules) from which the networks in this data flow processor are built. One is ***arbiter*** and the other is ***switch***. An arbiter passes one of its inputs to output based on the first come first serve principle. Therefore, if all the arbiters in Figure 7 are identical and if ***C*** arrives first, this 8x1 ***arbiter network*** will propagate ***C*** to output end ***O*** first. In order to avoid the simultaneous arrival situation, a round–robin discipline can be used to avoid this ambiguity.

A switch element passes input information to one of its output ends according to the boolean value at its vertical end. By cascading switch elements, one can steer

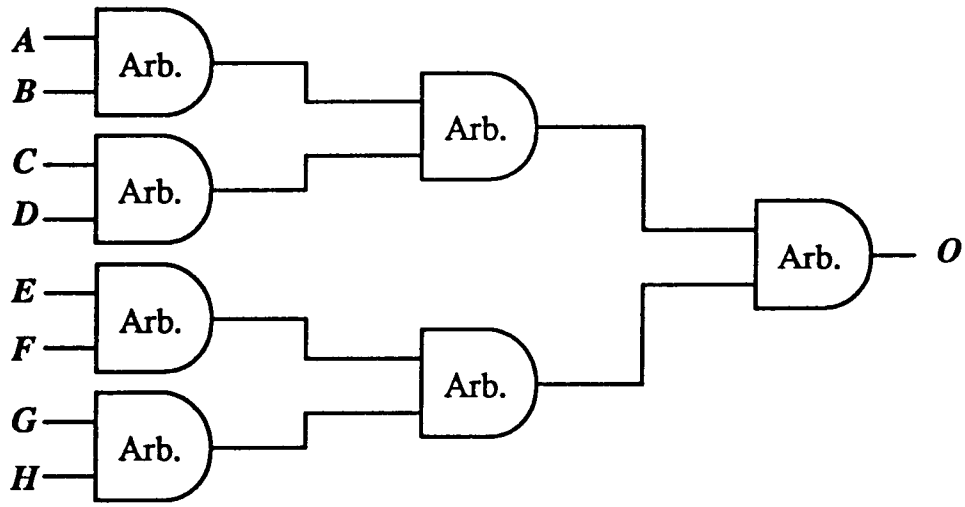


Figure 7: An arbiter network serves the arbitration function

an input data to any output end with appropriate application of boolean control values. In Figure 8, for example, if boolean control signals ***wxy*** equal 100, the input data will be routed to output end ***E*** for this 1x8 ***switch network***. Thus a switch network works just like a decoder. Actually, the control signals are not necessary from a separate controller. They could come from the incoming data themselves. This is true especially for self–timed circuits. If input data is sent to the output ends alternately and no boolean control input is required, this kind of switch is called ***Toggle***.

The ***arbitration network*** in this data flow processor serves two purposes. One is to match the number of instructions(the size of instruction memory) to the number of processing units. Usually, the size of instruction memory is much larger than the number of processing units. The second is to route the executable instructions in instruction memory to appropriate processing units(operation unit or decision unit). Therefore, the arbitration network must decide which instructions can use the processing units first and have the ability to choose all the available and appropriate processing units.

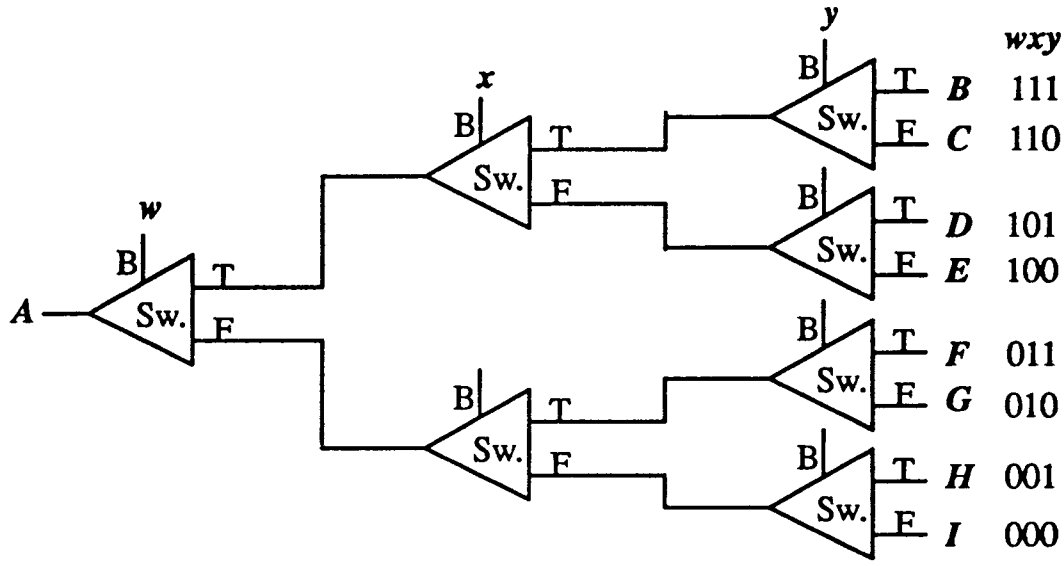


Figure 8: A switch network for steering input data to output end by applying control values *wxy*

Arbitration network is made of two parts: arbiter network and shuffle network. They are cascaded in series with arbiter network in front and then shuffle network. An arbiter network is composed of many arbiter modules. As discussed earlier, an arbiter network is used to determine which instructions can enter processing units first. Therefore, the first issue is solved. The second issue, however, can be resolved by shuffle network, as illustrated in Figure 9.

Shuffle network is a blocking network with the same numbers of input terminals and output terminals[16]. Any input terminal in this network has a unique path to any output terminal. This path is determined by the appropriate control signals. The advantage of this shuffle network is that only control signals decide the data output positions, irrespective of the input terminals it starts. For example, no matter which input will go to output position ***B***, the corresponding control signals are always the same(*xy*=10). This characteristic simplifies the control signal design.

In some cases, two inputs can not route to two arbitrary outputs at the same time without blocking each other. For example, if we are to route input ***I*** to output

A and *3* to *B*, the corresponding control signals are 11 and 10, respectively. The first conflict happens at the top arbiter in stage 2. Since if these two switches are identical, both data will arrive at this arbiter at the same time but only one of them can enter first. The second conflict occurs at the top switch in stage 3. The *1* to *A* mapping requires control signal $y = 1$; the *3* to *B*, however, needs $y = 0$. In order to have correct result, they must pass sequentially. The blocking problem will decrease the network performance[17]. The boolean control signals for shuffle in arbitration network are the opcode.

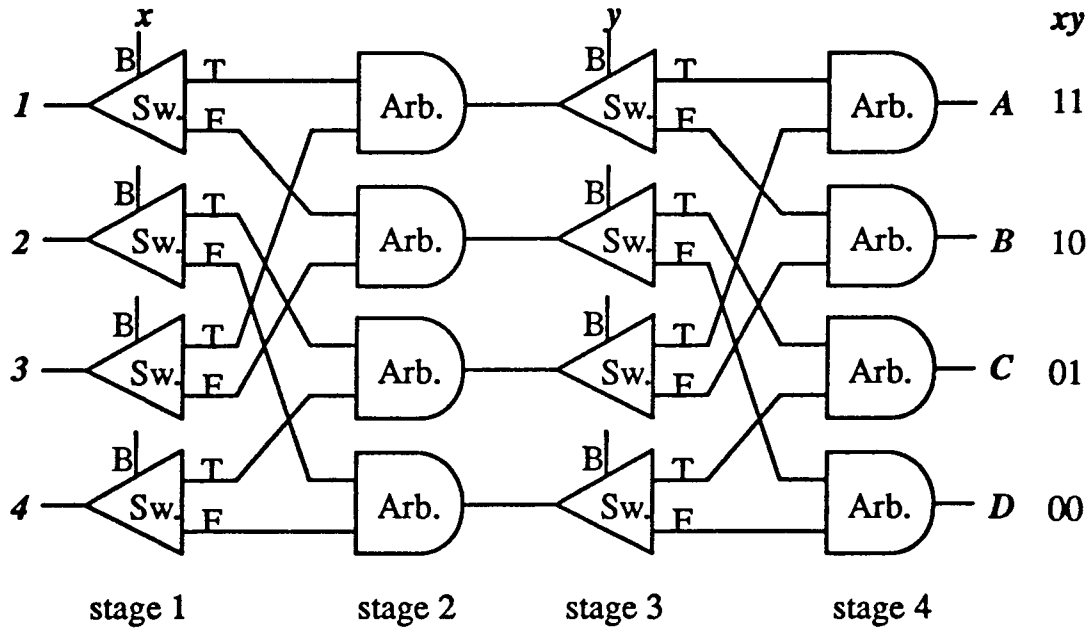


Figure 9: A shuffle network for input data permutation

The result from operation unit is sent to distribution unit and routed to appropriate instruction template according to the destination address. If only one result is generated by operation unit, a switch network can fulfill this routing work. If more than one result, however, are generated, shuffle network is required to route the results to any instruction template. That is, a ***distribution*** network is a shuffle network cascaded by a switch network. The boolean control values for distribution network are the

destination address of this result. *Control* network has the similar construction to the distribution network.

2.3.3 Processing Units

Processing units include operation unit and decision unit. Operation unit accepts operation packet, does arithmetic operation and generates data packet. Decision unit consumes decision packet, performs logic operation and produces control packet. Result tags must be included in decision packet and control packet to distinguish if the result is a value type or a control type. Table 2 shows the forms of packets.

Table 2: The packet forms for processing units

Packets	Forms
Operation	{ Op. 1, Op. 2, Opcode, Dest. Addr. 1, Dest. Addr. 2 }
Data	{ Result, Dest. Addr. }
Decision	{ Op. 1, Op. 2, Opcode, RT 1, RT 2, Dest. Addr. 1, Dest. Addr. 2 }
Control	{ Result, Result Tag, Dest. Addr. }

RT: Result Tag OP.: Operand Dest.: Destination

There is a sub–unit called split element in both operation unit and decision unit. This element deals with data/control link primitive. If there is no link corresponding to an instruction template, the second destination address is set to FF to indicate this situation. If the second destination address is not FF, split module will generate two data(or control) packets corresponding to two destination addresses. Otherwise, only one data(or control) packet is produced.

2.4 Merge and Identity Implementation

The reason why merge is mentioned separately in this section is that its firing rule and implementation differ greatly from other primitives. The first difference is that it has three input arcs(operands), two for data tokens and one for control token, one more than most other primitives. The other difference is that it only routes one of the input data to output without modifying the routed data. Although true/false gate also routes data only, they are implemented implicitly rather than explicitly like merge. In terms of implementation, merge can be dealt with just like true/false gate. However, it will complicate the control logic design for instruction template. For example, the results of S1 and S2 will both point to the address of R1 of S4. A complicated control design is expected to resolve this problem. This is why merge is implemented explicitly.

The gating code for merge, as shown in Table 1, has different meaning. It does not mean a node with which an instruction connects previously. It only shows which arc(true or false) is located in which register. This is different from what true/false gate are implemented. As stated above, merge only passes through data from input to output so it is implemented by add operation with the other operand set to zero by hardware automatically. The control signal for merge needs to go to two registers in its instruction template to form the control flags. In software, however, the destination address for the control signal is for true register only. The false register control signal is duplicated by hardware(refer to section 4.3.2). With this implementation the merge instruction is enabled as long as one of the registers in instruction template is enabled. An example as shown in Figure 10 and 11 will explain merge implementation.

It should be noted that the gating codes of R1 and R2 for merge(S3) are not Nos. They are True_m and False_m as was explained above. In addition, the

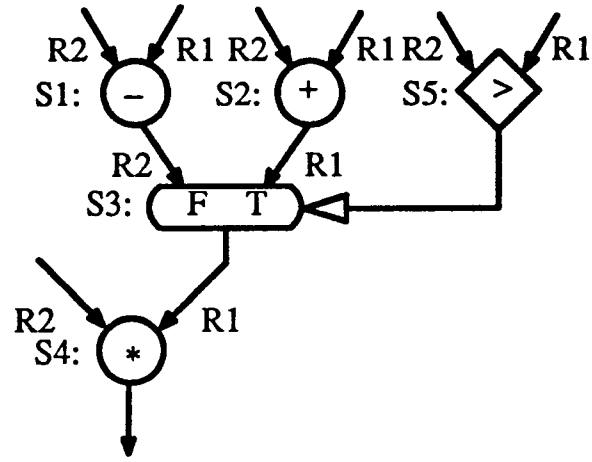
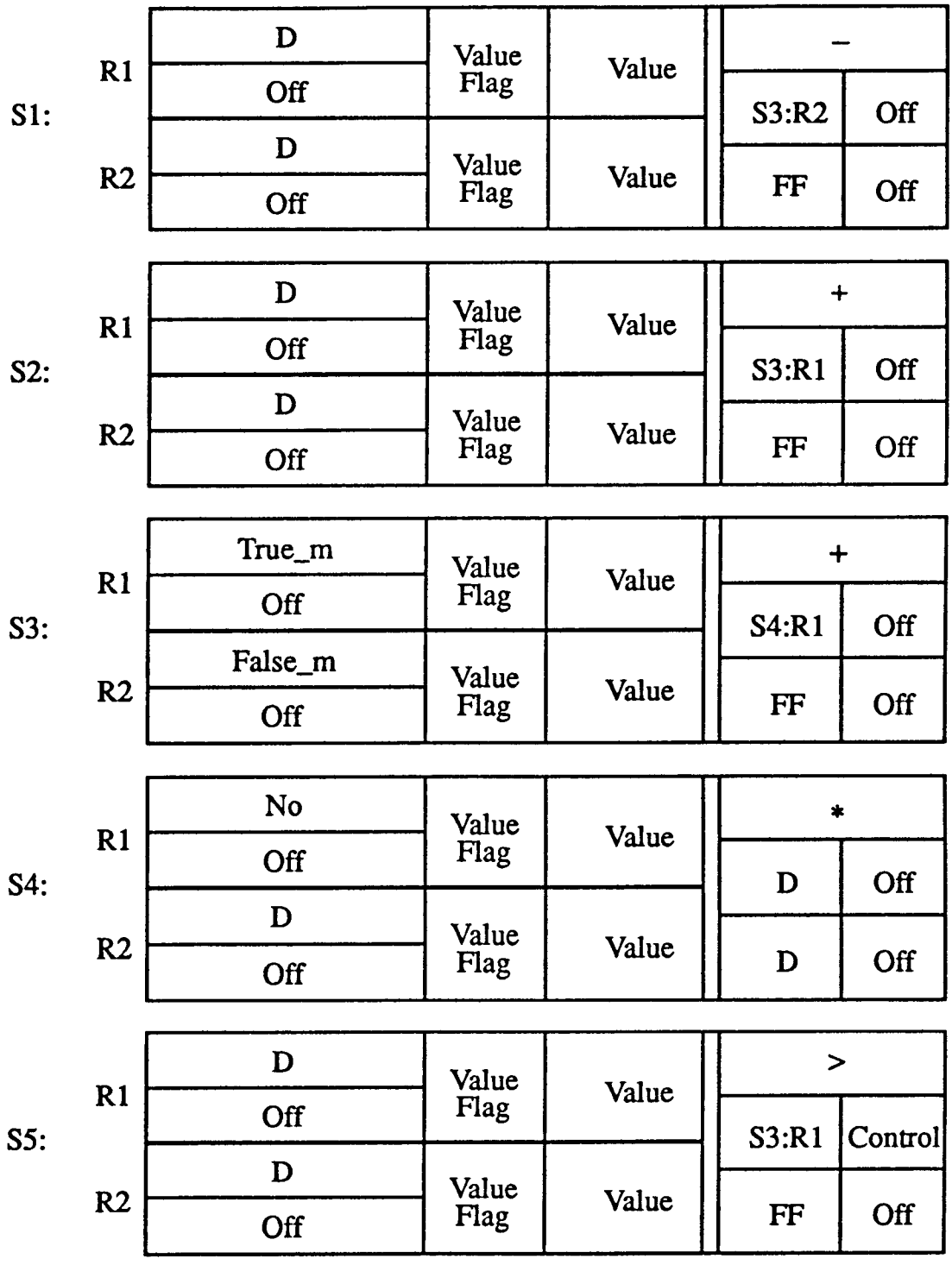


Figure 10: A data flow graph example for Merge implementation

destination address for ">" decider is pointed to first register of merge(S3:R1) only, not both. The control token generated by decider is control type rather than data type since it is for control purpose. The gating code for first register of node S4 must be No, not True_m or False_m. Figure 11 shows the instruction templates for the data flow graph in Figure 10.

How to represent the situation of either true or false gate is connected to one of the input arc of merge ? In our design, this can be handled by inserting an identity primitive between them. In terms of implementation, an identity instruction for data token/control token is a special case of add/OR operation with one operand set to zero. Note that this zero operand is considered as a constant so the gating code for this zero operand is Cons.



D: Depending on graph connection

FF: No data or control link

Figure 11: Instruction templates for the data flow graph in Figure 10

2.5 Summary

The data flow concept and a static data flow processor are described in this chapter. This static data flow processor is a preliminary version of general purpose processor. It can execute all the data flow graphs constructed by eight primitives — data link, control link, operator, boolean operator, decider, merge, true gate and false gate. However, only operator, boolean operator, decider and merge are implemented explicitly in instruction template. Except merge, an instruction can be fired only if both of registers are enabled. For merge, it becomes executable as long as one of the registers is enabled. After understanding of architecture for this static data flow processor, it is later implemented using micropipelines, which will be introduced in next chapter.

CHAPTER 3
MICROPIPELINES

Self–timed machines have many desirable features and advantages as mentioned in the first chapter. Many works have been reported on the implementation of self–timed processors[18][19][20][21]. They all use the 4–phase control scheme. In this thesis, however, we present the design of a self–timed static data flow processor using micro-pipelines, which employs the 2–phase control scheme. Micropipelines are self–timed elastic pipelines, hence, having pipelining inherent advantages. This chapter describes this different approach.

3.1 Transition Signalling

In a conventional design philosophy, the coordination between clock and control signals represented in absolute logic state(0 for Low and 1 for High) is the key to design a digital system. Clock–triggered concept helps eliminate the noise–handling problem since the signals will not be clocked into a module before they become stable or before noises die down. In order to distinguish the modes of control signals, absolute logic states are assigned to them. The active mode for different control signals could have different logic state, namely active–high or active–low. This kind of design framework is called clocked–logic conceptual framework, which is different from the transition–signalling conceptual framework proposed by I.E. Sutherland in 1989[22].

With the design using clocked–logic conceptual framework, the state of clock must return–to–zero(RZ) between two consecutive modes(signals). The RZ does nothing but for the preparation of next trigger so it is wasted in terms of time and energy cost. Therefore, in transition–signalling conceptual framework, no absolute highs or

lows are assigned to control signals. Instead, any signal transition(called an ***event***), either rising or falling, has the same meaning in terms of signal triggering. That is both rising and falling edges can trigger module operations. Thus, this framework may offer twice the speed potential of conventional clocking. This signal transition is called non–return–to–zero(NRZ) signal scheme in micropipelines. Since active modes are defined as signal transitions(low to high and high to low), the inactive modes in transition–signalling framework are signals in the absolute high and low levels. Figure 12 illustrates this idea. In a digital system made of CMOS technology, most of energy is consumed at signal transition and least of energy is used when signal does not change. Therefore, micropipeline–based circuits efficiently employ energy for useful operations and only very small amount of energy is wasted in inactive mode.

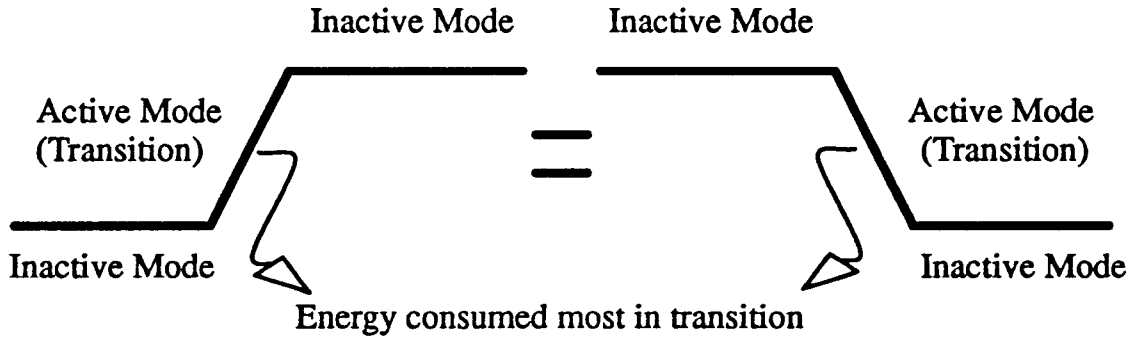


Figure 12: Two equivalent transitions(events) in micropipelines

3.2 The Two–Phase Bundled Data Convention

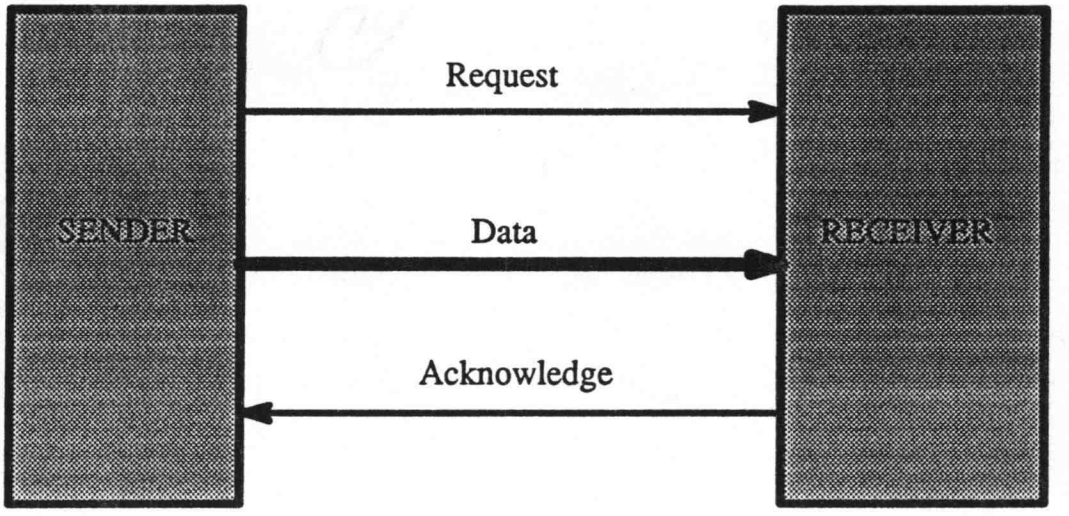
Clock and central controller are the two key modules that synchronize all the operations within a digital system based on the clocked–logic framework. How

is the synchronization done in a digital system based the transition–signalling framework(implying clock and central controller have been abandoned)? It is the **two–phase bundled data convention** playing the role of synchronization in micropipelines. Before we explain further, let us examine the physical connection needed for this synchronization method. In a sender and receiver environment, if they communicate asynchronously, handshaking protocol consisting of two control wires and many data wires is used mostly. One of the control wires is called request and the other one acknowledge. Request and acknowledge can be designed using either absolute logic scheme or transition signalling scheme; in micropipelines, however, they follow the transition signalling rule. For the data wires, absolute logic meanings are still preserved such that they are compatible with conventional combinational logic design. The relationship between sender and receiver is demonstrated in Figure 13(a).

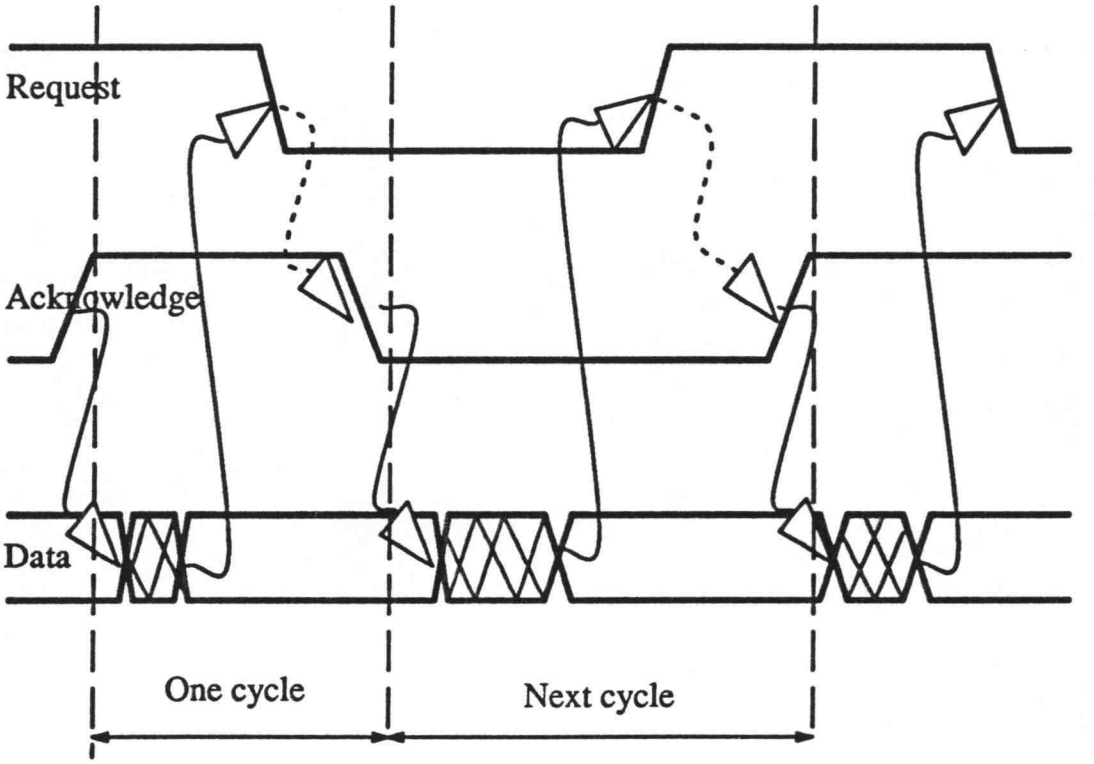
So what is "two–phase bundled data" convention? "Two phase" means that there are sender's active phase and receiver's active phase. "Bundled data" indicates that data and request wires must be treated as a bundle: data must arrive at receiver earlier than request signal. With respective to handshaking between sender and receiver, sender generates valid data on the data wires and then issues an request event to signal receiver the data availability. The receiver takes the data whenever it is ready and then produces an acknowledge event to tell sender the data consumption. At this point, sender can put new data on data wires and the whole procedure repeats. Note that data can not be changed before acknowledge event is issued. The relative signal timing is illustrated in Figure 13(b).

3.3 Event Logic Module And Event–Controlled Storage Element

As mentioned in section 1.3, self–timed circuits feature modularity and composibility so some basic but useful modules will be described. They include event OR,



(a)



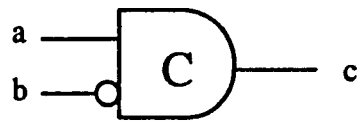
(b)

Figure 13: Two–phase bundled data convention (a) Module interface in micropipelines (b) Timing diagram

event AND, Toggle, Select, Call and Arbiter, as shown in Figure 14. These modules can be used to form various event logic circuits.

(1) Event OR: If any input of this module changes its state, output also changes its state. In terms of event, the arrival of any input event generate an output event. This works exactly the same as conventional XOR gate so its symbol is adopted as event OR module.

(2) Event AND: If all inputs of this module are in the same logical state, the output of this module will copy the input logical state to the output; otherwise, the output remains unchanged. This module's behavior plus the event definition imply that the output event will be generated only when all input events arrive. The event AND module is also called the Muller C–element[23]. For the control purpose(see next section 3.4), there is a useful variation of the Muller C–element as shown below, which is called the Muller C–element with bubble or the bubbled Muller C–element.



IF a and b are in different state
THEN copy a state to c state
ELSE hold previous c state

(3) Toggle: Toggle generates an output event alternately starting with the dotted terminal. It responses to events at its input after the initial master clear signal.

(4) *Select*: Select steers input event to the corresponding output according to the boolean value at the other input terminal. Note that the Boolean value must arrive before the input event.

(5) *Call*: This hardware Call module serves as the role of memorizing subroutine return address. It remembers which input events, R1 or R2, has called the procedure(event at R is generated). After the completion of procedure execution(event at D is generated), it returns a matching done event on D1 or D2. The Call works properly only when the current call has completed before the next call occurs.

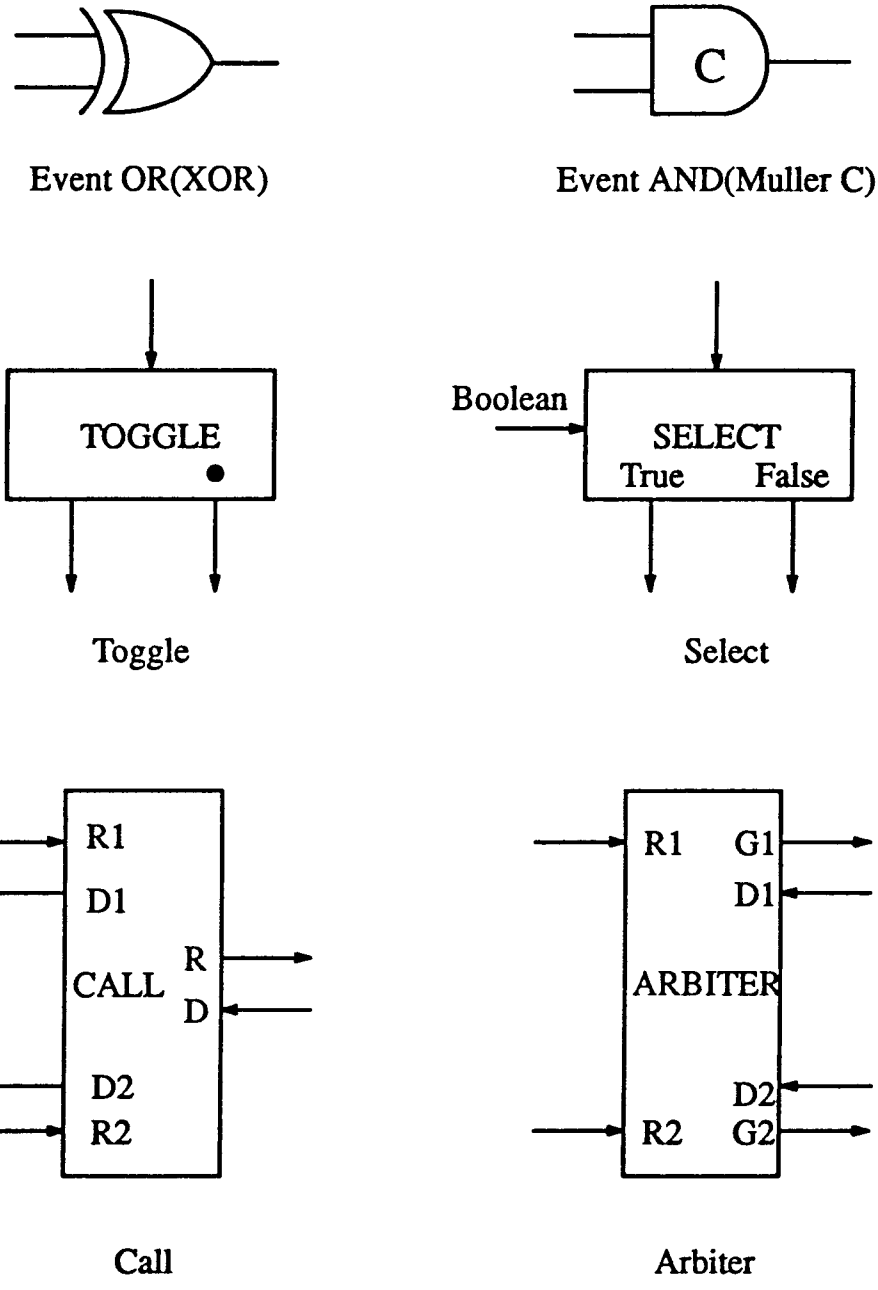


Figure 14: Event logic modules

(6) ***Arbiter***: An arbiter is used to guarantee mutual exclusive access to shared or protected resources. It grants requests(transition of R1 or R2 signal) by a transition on the G1 or G2 terminal. The service is performed using first come first serve rule. When one of the input event grants the service, the other event(if any) will be automatically delayed by this module until the current one is done.

In Sutherland's paper, event–controlled storage elements have two control wires, called ***capture*** and ***pass***. The ***capture*** signal latches present data and the ***pass*** signal allows new data to enter the storage element. There are two feedback loops within a storage element, which prevent data loss between the time when capture signal is issued while pass signal is not yet generated. When its two control signals are in the same state, this storage element become transparent, which is good for hardware debugging as will be seen in the next section.

There are, however, three reasons to replace this kind of storage elements with Double–Edge–Triggered Flip–Flops(DETFF)[24]. ***First***, the design methodology for DETFF is using two cross–coupled latches with input gating devices and some simple pass transistor logic instead of using conventional CMOS design with feedback. In addition, only one control signal is required to latch the data for DETFF so DETFF has potential speedup than 2–control–wire storage element. It has been pointed that this CMOS DETFF is capable of operating at more than 50MHz which is equivalent to 100MHz one–edge–triggered system frequency. ***Second***, DETFF uses less transistors(26) than the faster version of Sutherland's work(28). ***Third***, DETFF is readily available. Although the DETFF does not have transparence property, this two kinds of storage elements have identical functionality in terms of high–level logic design.

3.4 Standard Control Circuit

Due to the modularity and composibility feature, the standard control circuit for micropipelines can be easily and readily constructed by cascading each stage together directly. From the higher–level sender/receiver point of view, each stage plays both roles(sender and receiver) in micropipelines, as shown in Figure 15(a). For example, stage 2 performs receiver role with respect to stage 1 when stage 1 tries to send data to stage 2; after stage 2 obtains data from stage 1, it will pass data(may be modified) to stage 3 as a sender. All the data and control signal transfer between consecutive stages must follow the two–phase bundled data convention.

The mechanism to fulfill the two–phase bundled data convention requirement is very straightforward. Only string of bubbled Muller C–elements with appropriate interconnection, as illustrated in Figure 15(b), is required to form the standard control circuit for micropipelines. Observe that only odd number of bubbles(one in the standard circuit) are allowed in every loop around which events flow. The purpose of the bubble in each loop is to form oscillation such that a data fed into at input end can be bubbled through to the output end automatically without other control circuits.

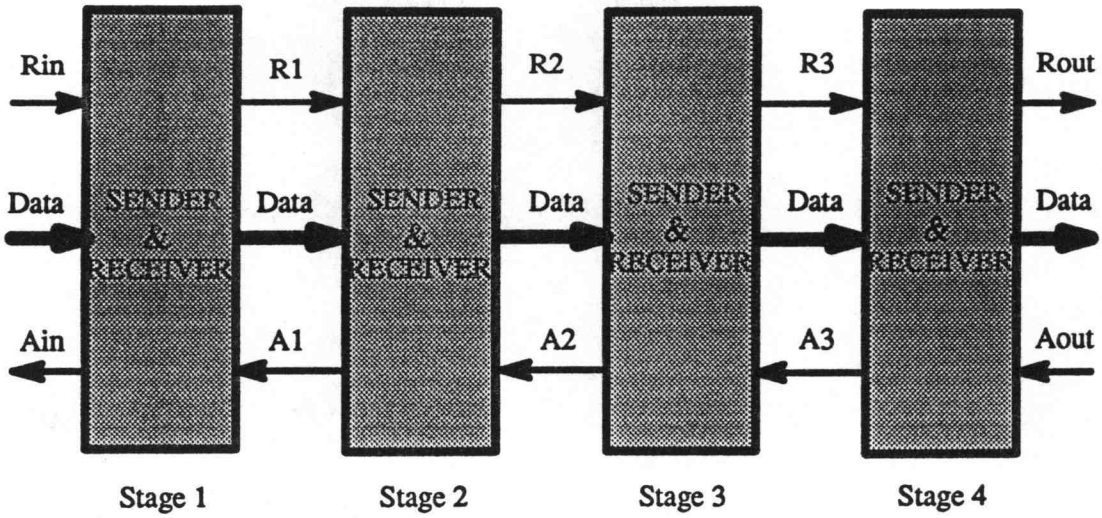
Although the absolute state of a control signal does not matter, its state relative to the other related signals(eg. two input terminals for a bubbled Muller C–element) does matter. With this understanding and the operation of bubbled Muller C–element, how this circuit works can be described in terms of the states of predecessor and successor Muller C–elements(with bubbles):

IF the states of predecessor and successor are different

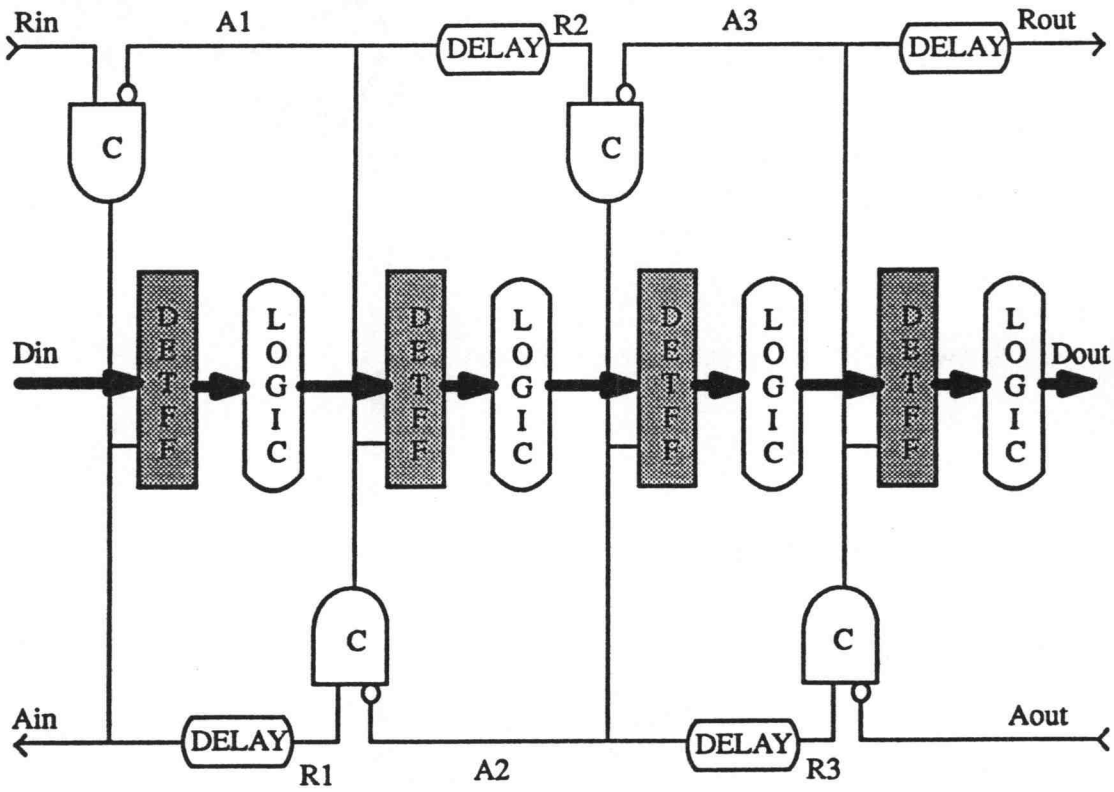
THEN copy predecessor's state

ELSE remain the present state

The control circuit is stable only when one of the following three conditions is satisfied:



(a)



(b)

Figure 15: Micropipelines in cascaded form (a) abstract level
(b) circuit level

(1) All the bubbled Muller C–elements are in the same states. This corresponds to an empty pipeline. New data can be fed into this circuit.

(2) Alternate stages are in opposite states. This corresponds to a filled pipeline. No more data is allowed to enter this pipeline until some data have been consumed(removed).

(3) Stages near the input end have the same state and stages near the output end have alternate state. This corresponds to a partly filled pipeline. New data can only fill empty stages.

The stage states that are not in one of these state conditions cause the circuit to be unstable. The circuit will not stop(idle) in an unstable state; it will continue to propagate through automatically until one of the above conditions is reached. Once a stable condition is reached, the circuit become idle(no power consumption theoretically) until a new datum enters or a datum moves out to destroy the stability.

The delay elements in Figure 15(b) ensure that the request signals and data are bundled in a way that valid data will always arrive at the next stage before the corresponding request signal's arrival. Different stages might have different delays depending on the complexity of combinational logic circuits within that stage. In general, that more complex the logic is implies a longer delay for the stage. Because each stage has different delay, it will operate at its own pace leading to the circuit maximum operation speed limited by the slowest one.

The delays can be eliminated if no significant processing logic is placed in the data path of a stage. This is usually assured by (1) the Muller C–element(with bubble) used in control signal path is more complex than the event–controlled storage element(or DETFF) used in data path so the control signal will be slower than data inherently; (2) a control signal must be amplified before it can drive many storage elements(depending on the data width) therefore the amplification increases the delays; (3) the layout of control signal path(zigzag path) is longer than the data path which

implies longer delays for control signals. Since FIFO is to buffer the data without modification, no delay elements are required.

In Figure 15(b), if two–control–wire registers are used instead of DETFFs, the registers become transparent when the pipeline is empty. This means that the whole circuit looks like a huge combinational logic circuit when we see from output to input. This is good for hardware troubleshooting in some ways. If all the internal processing logics are removed, the whole circuit in Figure 15(b) works like a First Input First Output buffer(FIFO). Figure 15(b) is just a simple standard control circuit. In a real circuit design, the control signal routing path could be very complicated. Some design examples will be presented in the next chapter.

3.5 Summary

Micropipelines feature NRZ signal scheme. All the transitions(events) are the same with respect to control signals. In order to have correct result, micropipelines must follow the Two–phase bundled data convention. Since micropipelines deal with events in designing the control path, some basic but important event logic modules are provided for convenience. Once each module is designed appropriately(following convention), direct interconnection is allowed between modules to form a larger and complex module or system. This feature releases the designers from the burden of timing calculation, which is essential in synchronous design. With this implementation tool ready, we are in a position to realize the static data flow processor stated in chapter 2 using micropipelines. This is the topic of chapter 4.

CHAPTER 4

IMPLEMENTATION OF A MICRO DATA FLOW PROCESSOR

In chapter 2, the high level data flow concept was described and also a preliminary data flow architecture based on this concept was presented. In addition to data flow architecture, an implementation methodology, called micropipelines, for self–timed machines was introduced in chapter 3. With this data flow architecture and implementation method in place, this chapter, therefore, is devoted to the realization of this preliminary data flow processor. This data flow processor implemented by micropipelines is called ***micro data flow processor***[25]. Some circuits and problems in implementation will be addressed.

4.1 Data Flow Graphs vs. Micropipelines

The relationship between data flow graph "token" and micropipeline "event" can be described as follow. When we say that token is generated, it means a request event occurs; when we say that token is consumed, it means the corresponding acknowledge event occurs. With this understanding, the data flow graph primitives can be further transformed to a lower level representation, as shown in Figure 16, which is closer to the micropipeline format[26]. Any arc in a data flow graph can be transformed into two arcs, one is request and the other one is acknowledge. *R* in this Figure mean Request and *A* represent Acknowledge. Both Requests and Acknowledges in a transformed data flow graph(primitives) correspond to the Requests and Acknowledges in micropipelines. A lighter color is assigned to acknowledge arc to increase the readability.

For example, in Figure 17(a), when only T_A (token at arc A) arrives, adder is not set to fire. After certain delay if T_B (token at arc B) also arrives and no

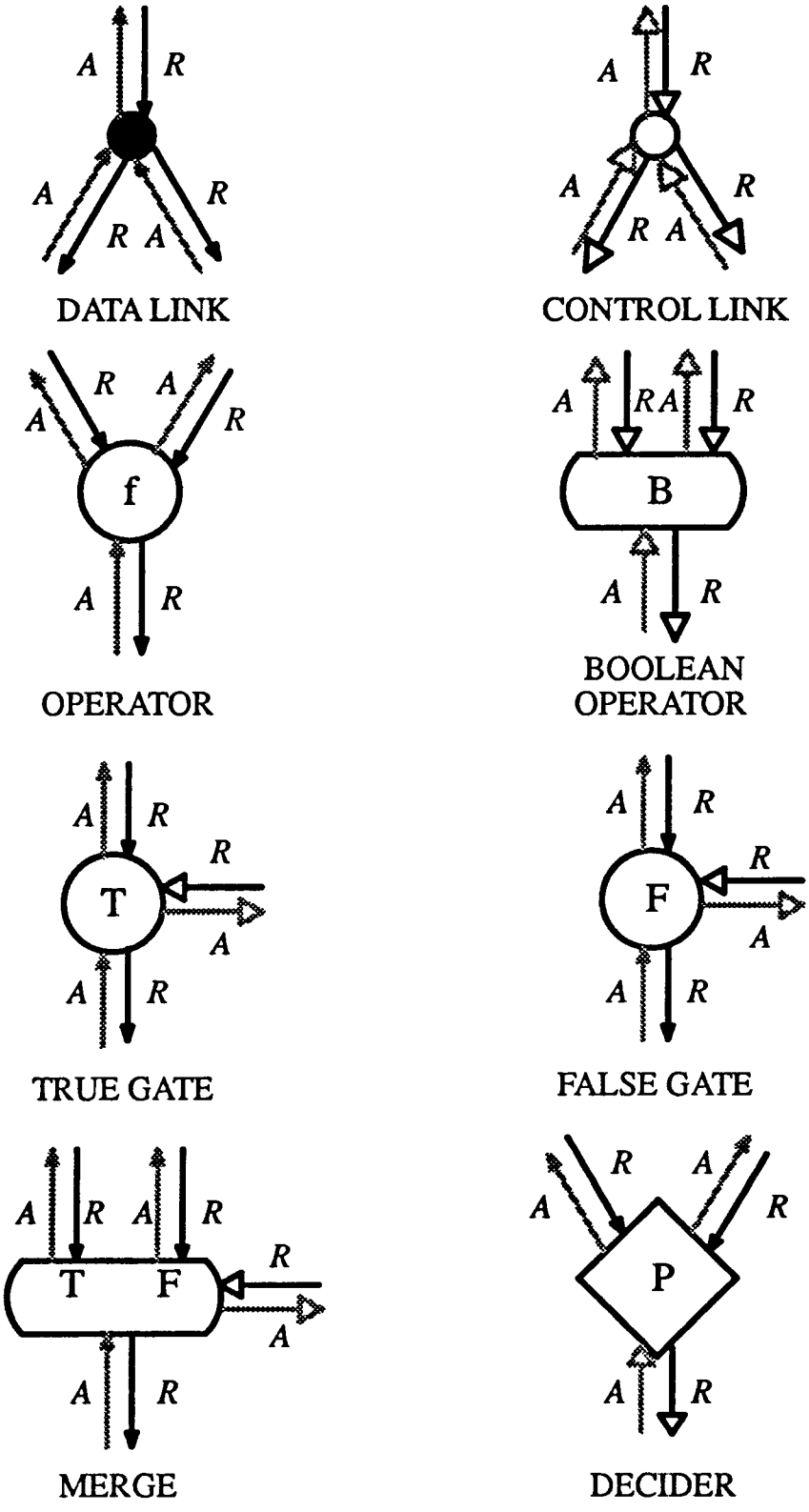
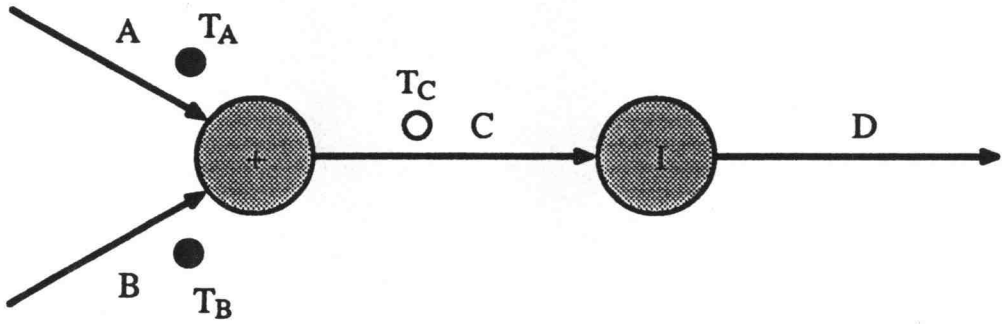


Figure 16: Transformed primitives of a data flow graph

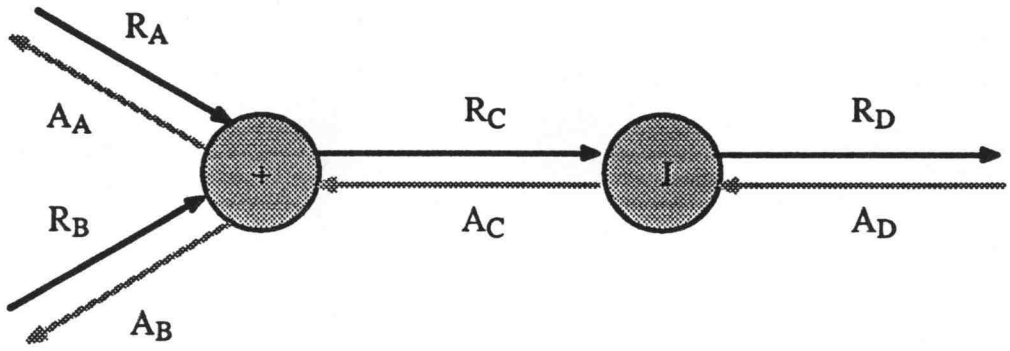
output token is present, then adder can fire. This leads to the consumption of T_A and T_B and the generation of a token T_C at arc C. This simple data flow graph can be implemented using micropipelines as shown in Figure 17(c). Muller–C without bubble(event AND) ensures that only after the arrival of both request event R_A(indicating token T_A generation) and R_B(indicating token T_B generation) will generate an output event. This output event will be fed into the Muller–C with bubble and then cause it to issue acknowledge events A_A and A_B(indicating tokens T_A and T_B consumption). At the same time(or after certain delay) an request event R_C(corresponding to T_C) is generated. A hollow token for T_C represents that this token is generated only when T_A and T_B are consumed. Figure 17(b) illustrates the intermediate state. The general mapping procedure is that convert a data flow graph(eg. Figure 17(a)) into a intermediate graph(eg. Figure 17(b)) by replacing each arc in a data flow graph with two arcs(request and acknowledge). Then implement each node in intermediate state using micropipelines. This implementation may not as easy as the example given in Figure 17(c). For example, according to the firing rule of Merge, its micropipeline realization is much complicated[26]. Direct mapping is especially useful when a special purpose processor is considered[27]. Although this mapping can not apply to a general purpose processor directly, such as the one we are describing, it still helps demonstrate the basic idea of micropipeline implementation.

4.2 Implementation Consideration

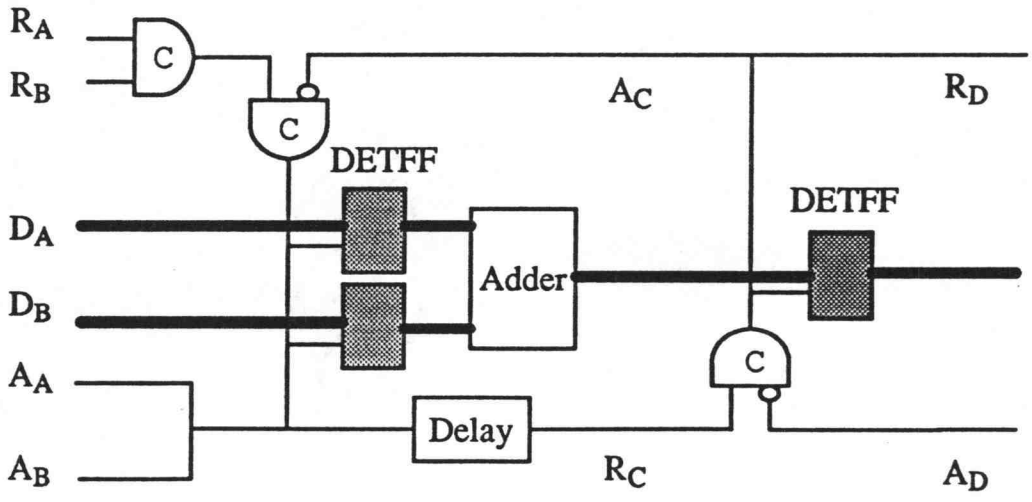
The standard control circuit for micropipelines has been discussed in chapter 3. In more complex design, the circuit itself is beyond the standard control circuit format. In reality, the circuit could be constructed in any form whenever possible. As a result, some problems occur and special attention is required. In this section,



(a)



(b)



(c)

Figure 17: A data flow graph and its corresponding micropipelines
(a) original (b) after transformed (c) micropipelines

the problem with matching token(phase) and the interaction between data and control signals are addressed.

4.2.1 Color token and phase problem

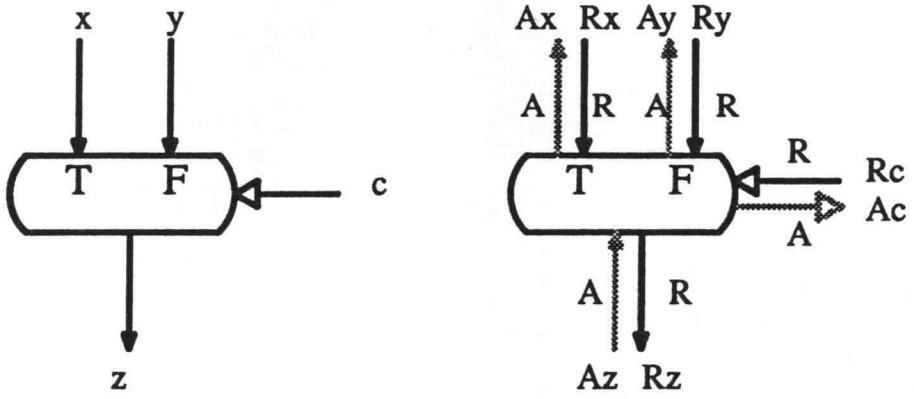
In the pure data flow concept, tokens flowing in a data flow graph can be distinguished by their functions as either data tokens or control tokens. No further distinction, however, is made for each kind of tokens. This is insufficient to precisely describe the token behavior if the implementation technology is taken into account. In using micropipelines to implement, two–color token, dark and light, is required to be made to distinguish if the oncoming event is either rising–edge trigger or falling–edge trigger. We can arbitrary assign the rising–edge triggered event as the dark token and the falling–edge triggered event as light token.

Some observations can be made for the merge actor implemented using micro-pipelines, as illustrated in Figure 18. The rows marked nB(where n = 1, 2, 3, 4, 5) indicate the token(event) conditions for each arc ***before*** the merge actor is fired. The shaded rows with mark nA, however, are token(event) conditions ***after*** the merge actor is fired. Here n represents the token wave input sequence. The control signal value(true or false) for token on arc c is indicated by the "Value" next to column c. Assume the acknowledge signal on output arc Az is generated immediately after the request signal Rz is produced. This is why the Rz and Az are in phase. Note that all signals are reset to the low state initially.

Observations we made are:

(1) When we say that token is generated, it means a request event has occurred; when we say that token is consumed, it means the corresponding acknowledge event occurs.

(2) Request and corresponding acknowledge signal for each arc are always



	x	y	c	Value	z	Rx	Ax	Ry	Ay	Rc	Ac	Rz	Az
Ini.						—	—	—	—	—	—	—	—
1B	●	●	●	T		⌐	—	⌐	—	⌐	—	—	—
1A		●			●	—	⌐	—	—	—	⌐	⌐	—
2B	◎	●	◎	F		⌙	—	—	—	⌙	—	—	⌐
2A	○				○	—	—	—	⌐	—	⌙	⌙	—
3B	◎	◎	●	F		—	—	⌙	—	⌐	—	—	⌙
3A	○				●	—	—	—	⌙	—	⌐	⌐	—
4B	◎	●	◎	T		—	—	⌐	—	⌙	—	—	⌐
4A		●			○	—	⌙	—	—	—	⌙	⌙	—
5B	●	●	●	F		⌐	—	—	—	⌐	—	—	⌙
5A	●				●	—	—	—	⌐	—	⌐	⌐	—

● : Dark token
◎ : Light token
— : State low
⌐ : Rising–edge event
⌙ : Falling–edge event
— : State high

Figure 18: Two–colour token for merge using micropipeline implementation

in phase. That is if the request signal is rising–edge trigger then the corresponding acknowledge signal occurring after must be also rising edge. Same thing can be applied to falling–edge signal.

(3) Token color for c and z always alternates each time the merge actor is fired. This also means events on Rc and Rz are always alternate(rising — falling — rising ...). This is because token on c is always consumed and on z is always produced no matter what's the control signal value it is.

(4) Tokens on arc x and y could be different color when they are processed. Same thing can apply to x and c, and y and c. The corresponding part for event illustration Rx and Ry, Rx and Rc, and Ry and Rc could be out of phase. This phenomenon leads from the merge firing rule.

In terms of hardware realization, of course, out of phase situation will increase the difficulty in hardware implementation. Actually, in addition to merge, true and false gate also have the similar out–of–phase problem. The input data token and control token must always be in phase but the input and output tokens could be out of phase. Figure 19(a) is a phase transformer symbol and its corresponding circuit. This phase transformer helps deal with the in–phase/out–of–phase problem. One condition is that S must arrive earlier than event input Ein in order to have a correct operation. Multiplexer in this figure provides a feedback loop from output end to input end. This loop determines if output event Eout should be generated or not according to the S value. Figure 19(b) is an example for input–output event waveform.

From the implementation point of view, the circuit in Figure 19(a) provides an interface between sender and receiver modules in considering the complex design of control signal logic path for micropipeline implementation. With this circuit as an interface, the sender can issue an event and the receiver can accept no event, in–phase event or out–of–phase event according to the value at S. This circuit appears to have ***memory*** to memorize the type of previous event(falling or rising) that sender

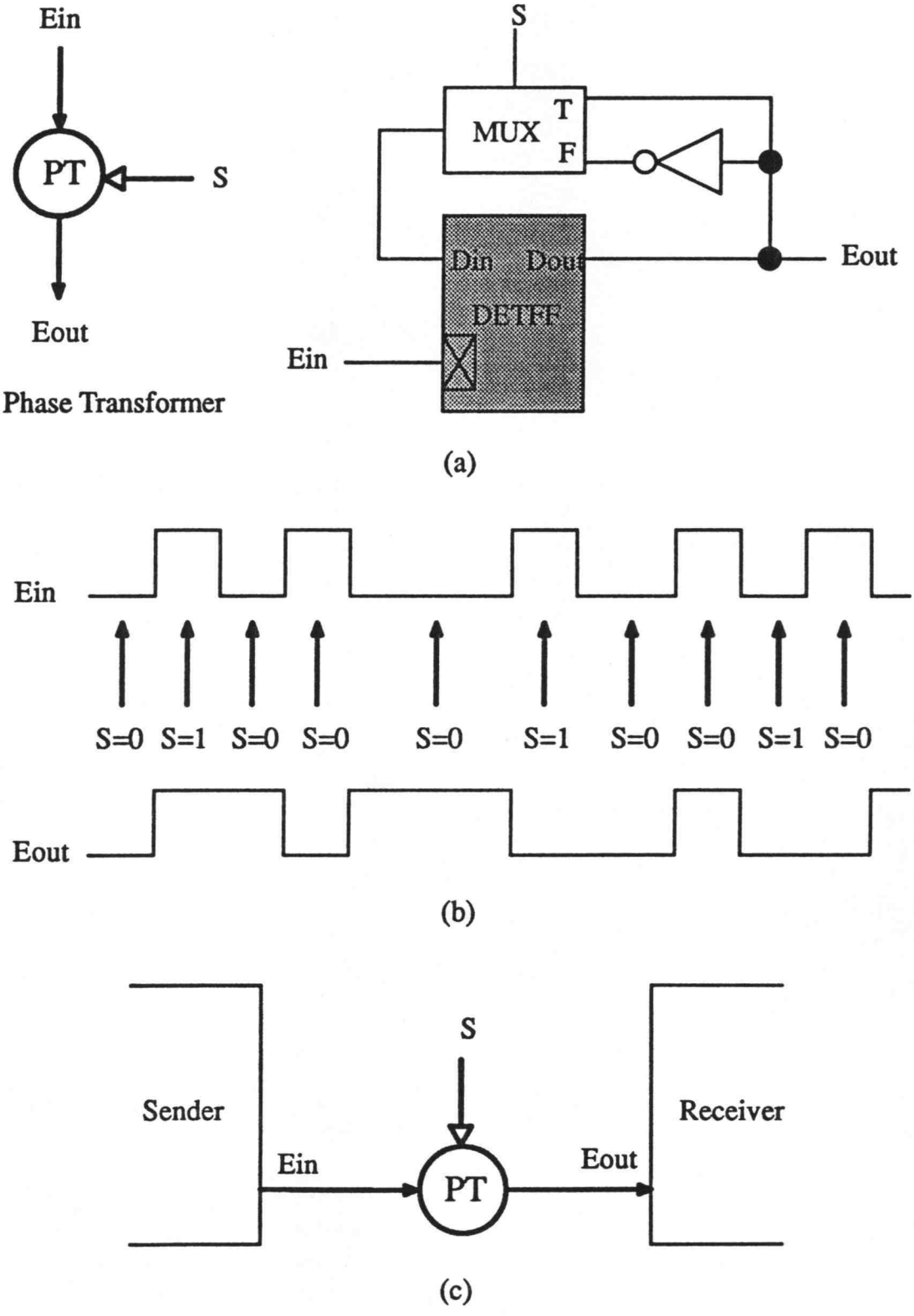


Figure 19: Phase transformer (a) implementation (b) input–output event waveform (c) application in micropipeline implementation

sent and to adjust current input event to generate(if any) an appropriate output event(in–phase or out–of–phase) to receiver. Figure 19(c) shows this application. This phase transformer has been widely used in this data flow processor implementation.

4.2.2 Data Path and Control Path

The data path in micropipelines is the path that data pass through only. Basically, it is the path through registers and combinational logic circuits in a standard control circuit. The upper and lower path through bubbled Muller C–elements is the control path. In terms of topology in a standard control circuit, data path and control path are totally isolated except that the registers are triggered by control signals. In terms of operation, however, they are intimately bundled by delay element in each stage.

As mentioned in section 3.4, this delay element can be eliminated if the combinational logic is not so complicated. If this is the case, the design work becomes very easy to handle since the data path design and control path design can be totally separated. We are more familiar with the data path design so let us just focus on the control path design.

In practical design, data path and control path could interact. This is especially true when control path is complicated. Figure 20 illustrates this idea. Data x is fed to lower control circuit for control purpose. Since data x is not stable until certain delay of time, a delay element must be incorporated into control path such that the stable data x arrives earlier than the event control signal E. Strictly speaking, this is not a bundled data convention but it can be treated as the same way. Actually, data should always be valid before the control signal arrives when asynchronous circuit design is considered.

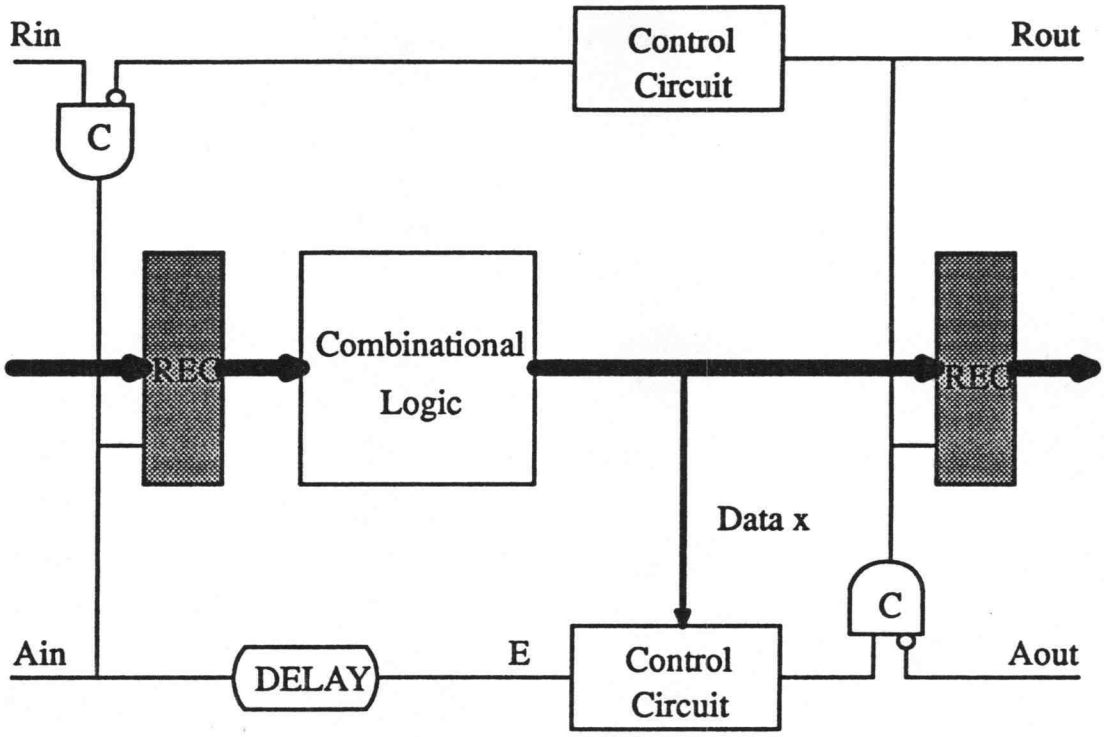


Figure 20: Interaction between data path and control path

Although data can be used to control the control circuit, this does not mean they can be connected in an arbitrary way. On the contrary, data path(combinational logic outputs), in general, are ***not*** allowed to connect to the inputs of event–controlled logic modules. This is because the data path outputs are varying(oscillating) before they become stable. If they are connected to the inputs of event logic modules, these modules will treat these oscillating signals as event triggers leading to incorrect results(multiple events could be generated). In conventional clocked circuit design, mix data and control signals are common since the noise can be filtered out by clock synchronization. Note that the control signals in a conventional logic design and in micropipeline have different functionalities. In a conventional logic design, control signals are used for control purpose only. Synchronization of the system is done by the global clock. In micropipelines, however, control signals are used both for the control and synchronization purposes. The data S from data path in Figure 19

is an example which a datum controls the event logic circuitry to generate(if any) an output event.

4.3 Data Flow Processor Implementation

This section introduces the implementation of a static data flow processor described in chapter 2. Two of the most important and interesting parts of this processor are the instruction memory and the network design. Therefore, we will first cover the design and implementation of the network and the instruction cell in this section. We will also discuss the deadlock(hardware itself) resulted from cyclic micropipelines.

4.3.1 Network implementation

As mentioned in chapter 2, all the networks in this processor are made of two basic elements — Switch and Arbiter. Therefore, we will show how to use the basic event logic modules to implement the Switch and Arbiter elements.

4.3.1.1 Switch circuit

The switch module in a distribution network of the data flow processor can be implemented by the Select module described in section 3.3. In our design, each Switch module is a micropipeline stage. That is, each stage must satisfy the two–phase bundled data convention. Therefore, a direct connection with the other modules is allowed(modularity and composibility). No additional timing concern is required if each module is correctly designed.

Figure 21 shows the Switch circuit. Initially all the wires reset to low states. When an event occurs on R(state becoming high), this event will go through the bubbled

Muller–C to produce an event on A(state becoming high) which will latch the incoming data into register. Thereafter, this event will go either to the false end or the true end of the event selector(Select module) according to the value at S. An XOR module(event OR) is used for acknowledges either from AF or AT, to indicate that data have been latched by the next stage and also to indicate the ability to latch new data from previous stage. Note that there is a delay module prior to the event selector. Without it, the bundled data convention will not hold and therefore the circuit fails to work. Since delay at register is 1ns and the delay between S and CLK is 1.22ns, the minimum delay required to maintain bundled data convention is 1+1.22=2.22ns. We use 2.5ns for the delay module which includes some tolerance.

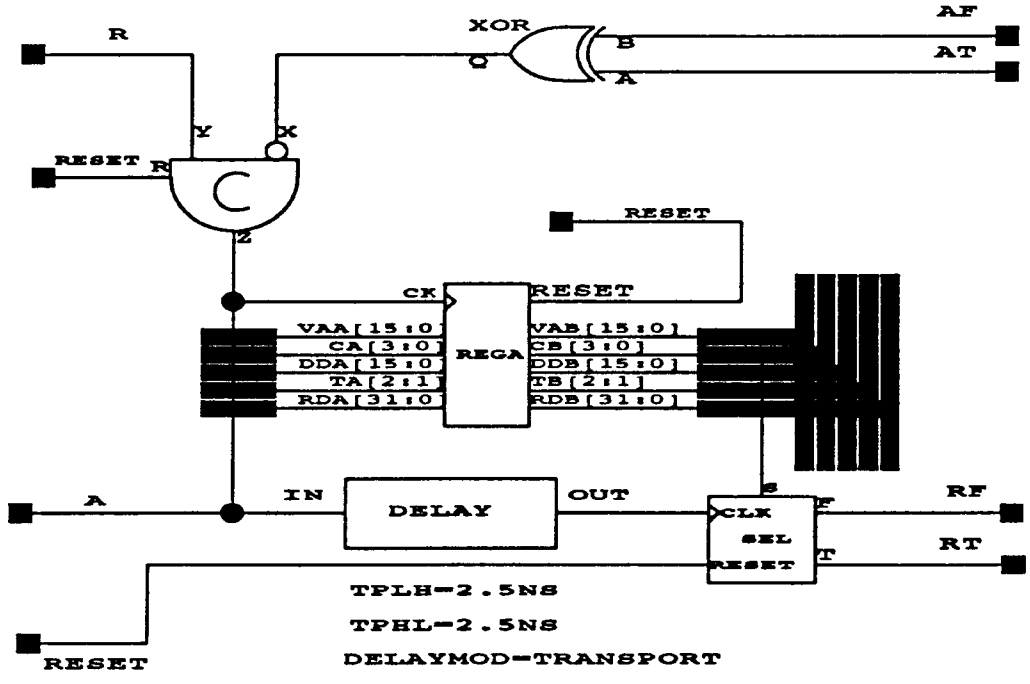


Figure 21: Switch circuit

4.3.1.2 Arbiter circuit

The arbiter module in the arbitration network can be readily designed using the basic event–controlled logic modules — Arbiter and Call. For the arbiter circuit in Figure 22, assuming event R1 arrives first, an event will be generated on G1 which sets S to 1 to select corresponding data. Arbiter will delay any incoming event on R2(if any) until a done signal on D1 is generated. In turn, an event passing through Call and bubbled Muller–C occurs at R (and D). The event at D will be routed back through Call and Arbiter to activate the event at R2(if any). At the same time, the S is reset to 0. The lower path is then established if event on R2 is present. The circuit between Arbiter and Call ensures S is reset to 0 when the R1 request has done.

4.3.2 Instruction cell implementation

In the data flow processor, the grouping of instruction cells forms the instruction memory. Each instruction cell consists of two identical instruction registers. This section shows the circuit of an instruction register and the block representation for an instruction cell.

The circuit for instruction register in Figure 23 determines if the register is enabled or not. Data packet out of operation unit will go to the upper bubbled Muller–C section and the control packet out of decision unit will go to the lower bubbled Muller–C section. Depending on the instruction itself and the data extracting from data packet and control packet, the values of E, R and S will serve as instruction register status flags.

When value flag is set(V=1 for the output of block 1) and gating code and gating flag match(M=1 for the output of block 3), E becomes high. When E is

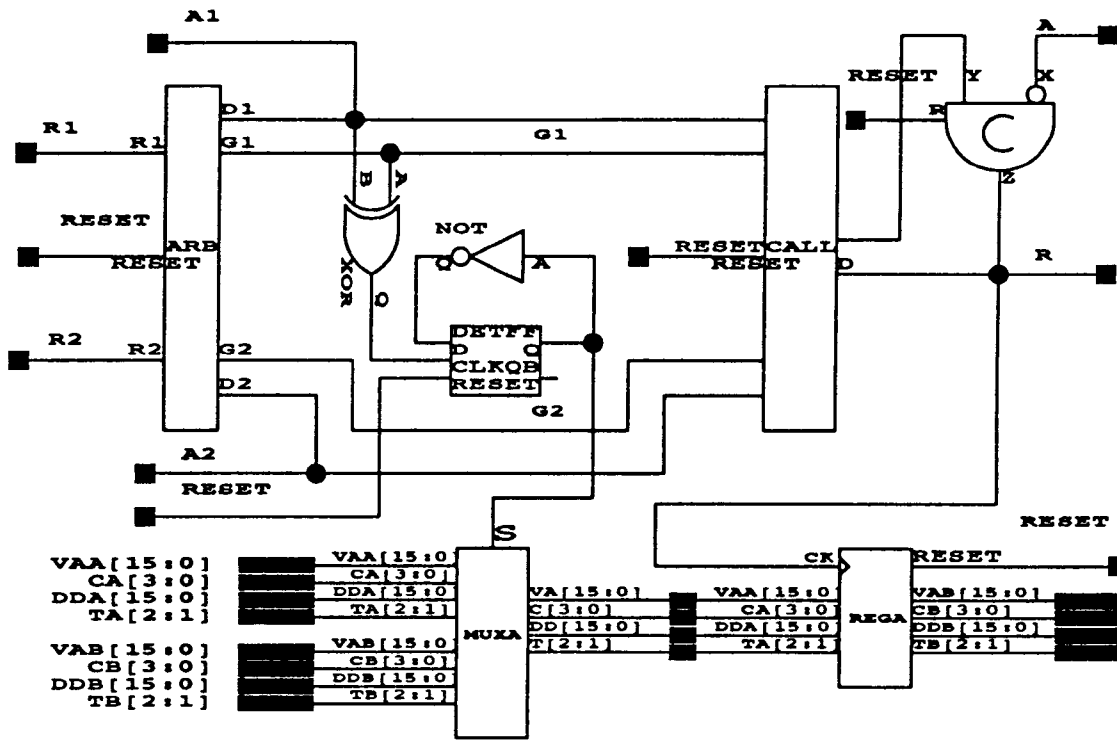


Figure 22: Arbiter circuit

set, it means the register is enabled(event occurring on RO). For operator, boolean operator and decider, enabled register means the data token has arrived on the corresponding arc. For true/false gate and merge, it means both the data token and control token have arrived and the operation is ready to fire.

If R is set, this implies a false control token has arrived for a true gate or a true control token has arrived for a false gate. Both events out of bubbled Muller–Cs will then loop back to themselves through block 6, block 7 and XORs(event OR) instead of going out to RO.

If S is set, only the event out of lower bubbled Muller–C will loop back block 7. The event comes from upper Muller–C will be left intact. In fact, this follows the firing rule of a merge actor.

Figure 24 illustrates the block representation of an instruction cell. The register block diagram in Figure 24 is an abstract representation of the circuit depicted in

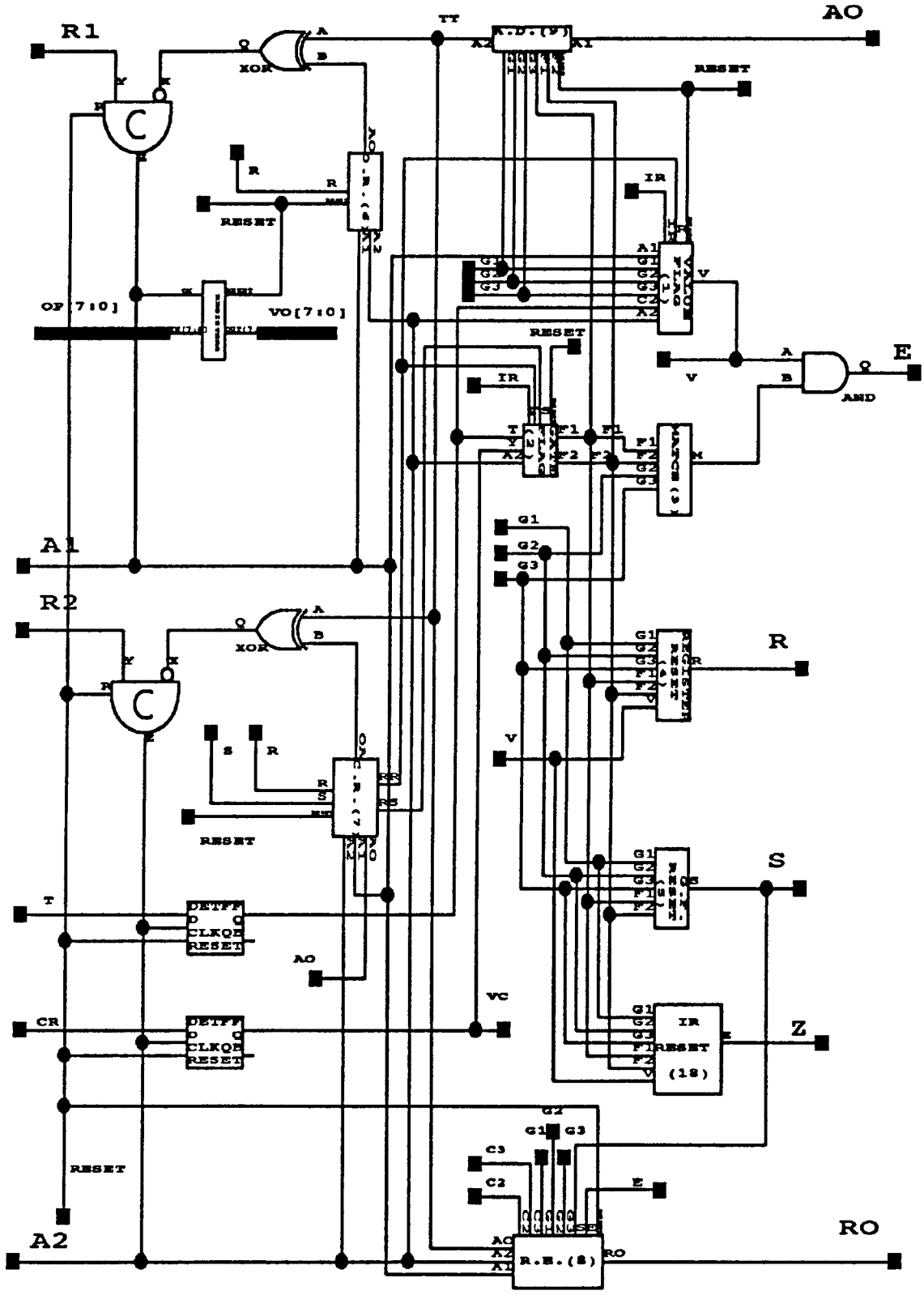


Figure 23: Circuit of an instruction register

Figure 23. Operation control part in register block corresponds to the upper bubbled Muller C–element portion and decision control part to lower bubbled Muller–C element portion. The data streams are output packets from distribution network. These streams are the data packets excluding destination addresses. This is obvious since the destination address has been used up in distribution network for data routing. Similarly, control streams are the outputs of control network and are equal to the control packet except the destination address. That data stream concatenates control stream(excluding result tag) leads to value stream.

For the active primitives(operator, boolean operator and decider), when all the data streams and/or control streams have arrived and both register 1 and register 2 are enabled(event occurs on both R1 and R2), the instruction enable unit will generate an event on R, which means the corresponding instruction cell is enabled. Or in terms of data flow concept, the corresponding node(instruction) is fired. The data manipulation unit under active primitive processing has two functions. First, it is used to merge boolean result into Least Significant Bit (LSB) of operation result to form an operand if boolean operation is processing. Second, it concatenates two operands and incorporates two destination addresses, opcode and result tags to form a whole stream. This whole stream is then fed into the arbitration network. The control switch unit works as a transparent element under the processing situation of active primitives.

In considering the merge processing, either R1 or R2(not both) has an event on, the instruction enable unit will generate output event on R to indicate merge becomes executable. Actually, this is the firing rule of a merge actor. It is natural to the merge actor that only control stream 1 is valid(active). Control stream 2 is not used. This is because only one control token is delivered to the merge actor. The stream to the decision control part of register 2 is duplicated from the control stream 1 by control switch unit. As mentioned in section 2.4, the merge actor is realized by

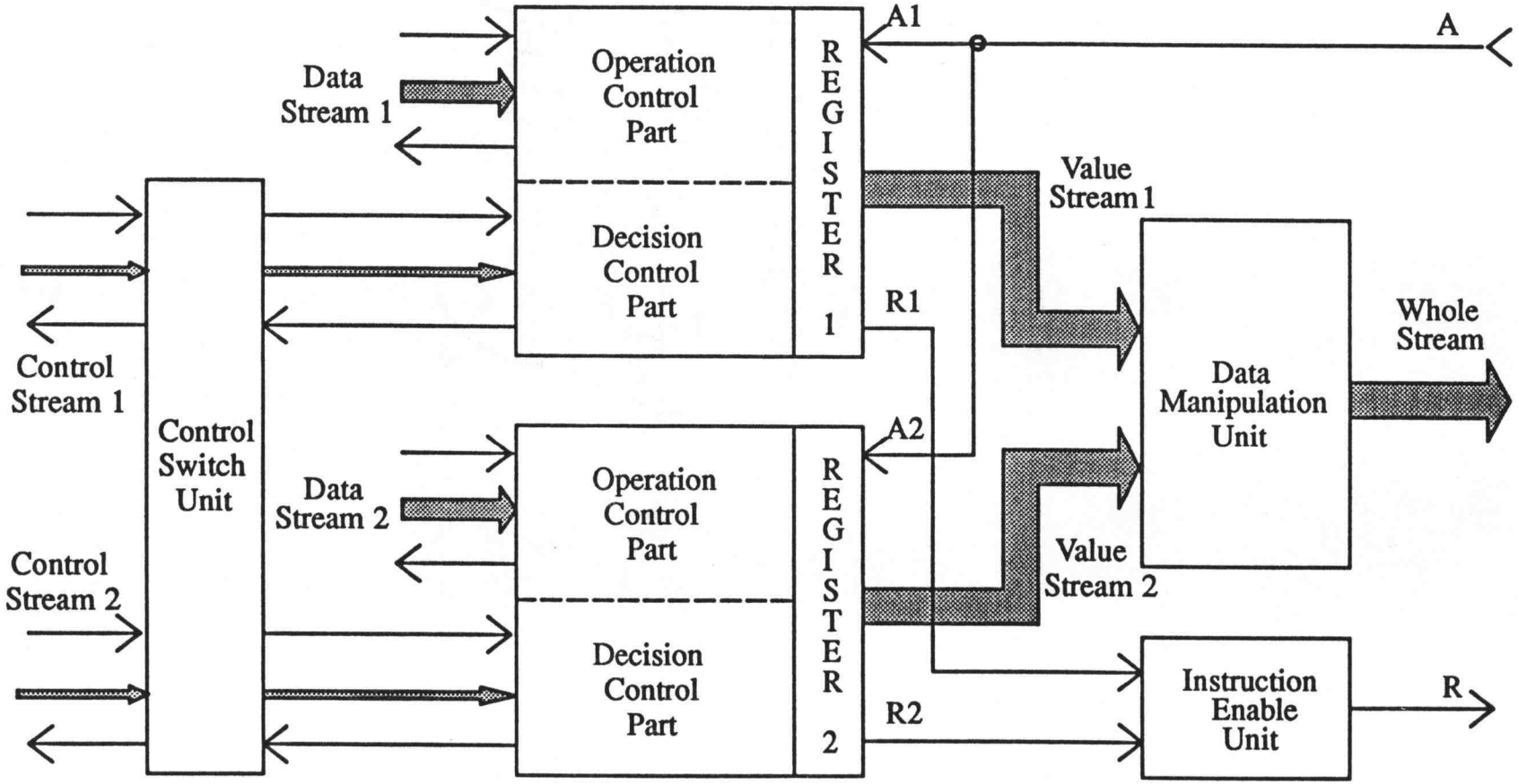


Figure 24: Block representation of an instruction cell

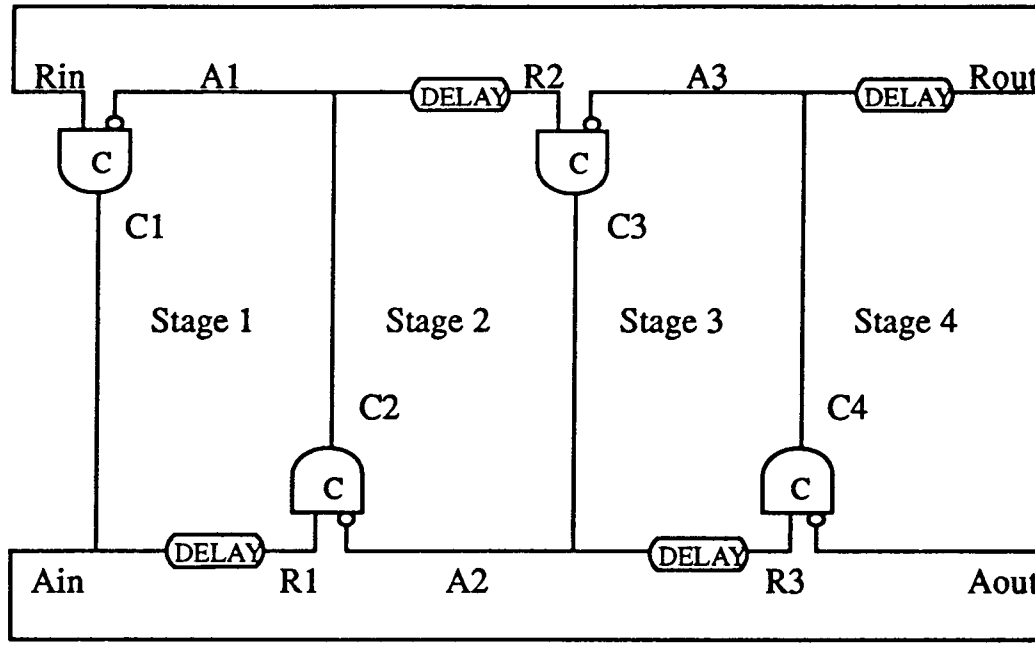
addition. One operand is either data stream 1 or 2 and the other operand should always be zero. This zero is automatically produced by the data manipulation unit once merge is detected.

4.3.3 Cyclic Micropipelines

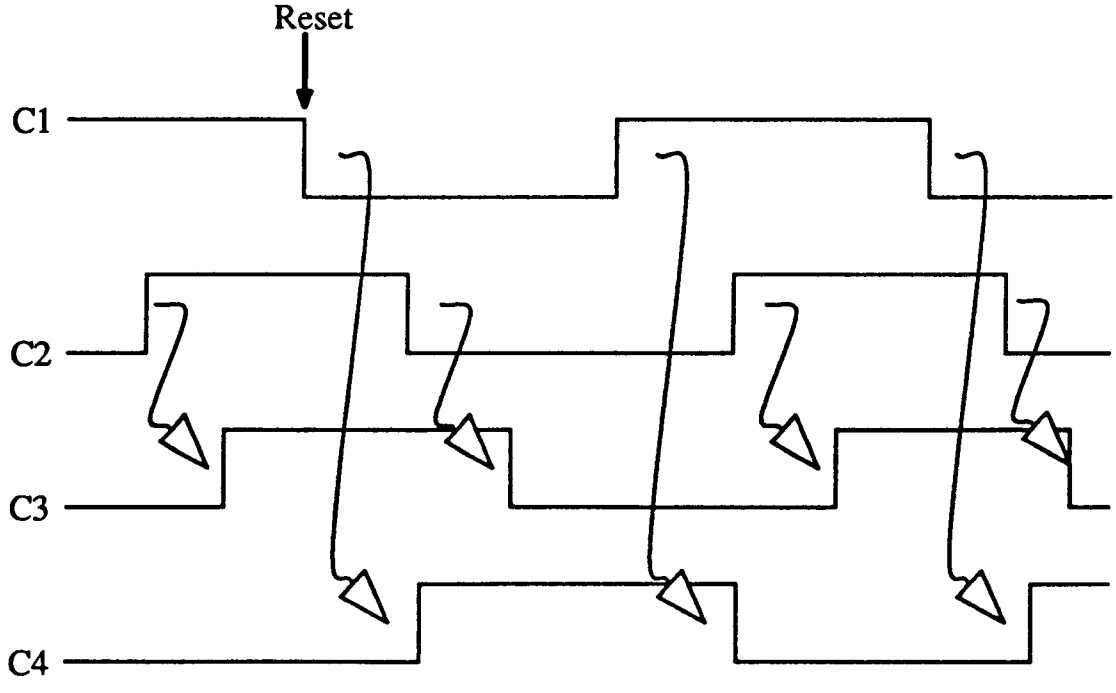
The standard control circuit for micropipelines is straightforward. Some problems, however, arise if the micropipeline is connected in circular manner, as shown in Figure 25. The first problem is how to initialize the system? This is equivalent to asking how to generate the initial tokens? The second problem is will this circuit execute in circular fashion? That is, in Figure 25, will the execution follow the order of stage 1 —> stage 2 —> stage 3 —> stage 4 and then back to stage 1 again? The third one is how to interface with its environment for primary data input?

If the circuit is initiated by setting C1 = 1, data will only propagate to stage 3 and stay there forever. This is because Aout is high initially and R3 is also high when data arrive stage 3. No event will be generated for bubbled Muller C–element with two input terminals at the same state. A way to allow the circuit to continue is to reset the first bubbled Muller C–element. By resetting it, C1 becomes low later. The timing diagram of resetting C1 and what follows is shown in Figure 25(b). Unfortunately, this way does not solve the second problem since from the timing diagram the output event for each stage is not generated in the order as described above. Neither of this method provides an acceptable solution for the third problem.

In the data flow processor proposed, there exists cycles. That is, token(packet) will initiate from instruction memory through arbitration network, processing unit, distribution network or control network then finally back to instruction memory. In using micropipeline to implement the architecture, we face the problem of how to



(a)



(b)

Figure 25: Circular micropipelines (a) circuit (b) timing diagram

generate initial token and to interface with environment, and yet to execute in proper sequential order. Figure 26 shows a way to deal with these problems by assigning some appropriate values to S1, S2 and S3. If we let S1 equal to 1, this initial event passes through event OR module and generates an output event on Rin and therefore, initiate the whole circuit operation. If we also set S2 = 1 and S3 = 0, the whole circuit will operate in a circular order as shown in Figure 26(b). This circular operation is preserved due to the introduction of an "inverter function" at the bottom. The XOR gate with one input terminal(S2) equal to 1 functions as an inverter. This inverter flips Aout back to 0 before R3 becomes 1 when next event occurs. This is the way to generate the initial token. It is similar for the situation of feeding external data into cyclic micropipelines. This is done by assigning S3 = 1 to block any undesirable Acknowledge signals back to previous stage. This extra circuit controlled by S1, S2 and S3 is placed between the control network and instruction memory for the data flow processor we have implemented. Similar circuit is also located between distribution network and instruction memory. The following conditions are a summary of all general situations in micropipeline circuits.

(1) Normal situation(no initial token and external data input):

==> S1 = 0, S2 = 0, S3 = 0;

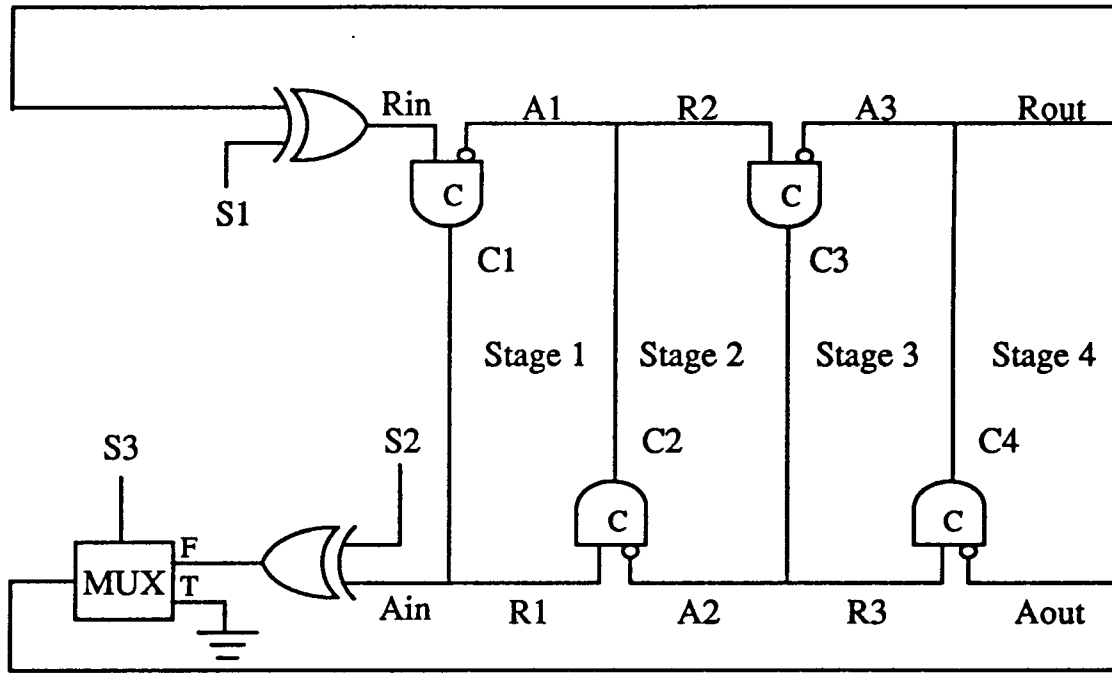
(2) Initial token required:

==> S1 = switching from 0 to 1 once, S2 = 1, S3 = 0;

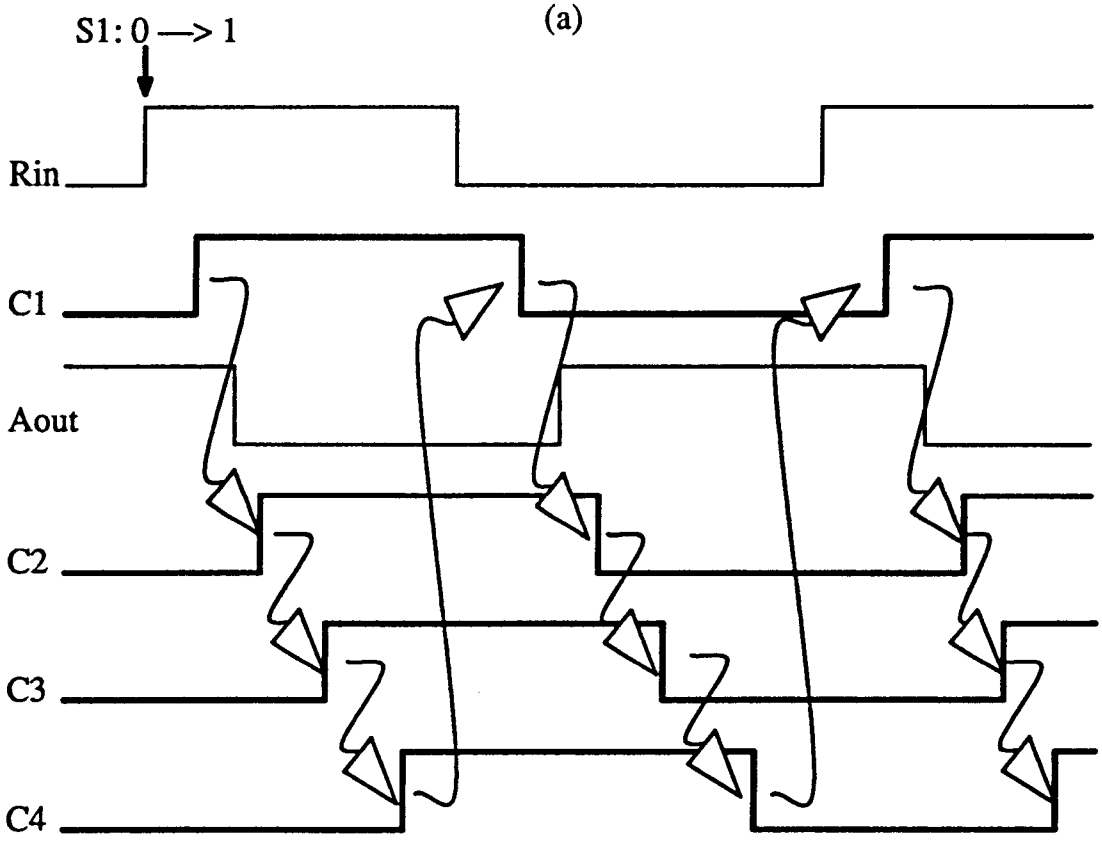
(3) External data input:

==> S1 = toggling between 0 and 1, S2 = 0 or 1, S3 = 1.

Appendix A shows the corresponding names for S1, S2 and S3 in each instruction register of the micro data flow processor.



(a)



(b)

Figure 26: Modified circular micropipelines (a) circuit (b) timing diagram

4.4 Summary

The static data flow processor described in chapter 2 has been implemented using micropipelines in this chapter. Some useful design tips for stepping into more complex design using micropipelines are described. These include the ways to handle phase problem, and data path and control path interaction. With the help of these tips, the Switch, Arbiter and Instruction cell can be readily designed. The initial token generation and environmental interface are essential issues in designing the data flow processor described in chapter 2. These are resolved by introducing some extra but simple circuit with three control signals S1, S2 and S3. Since all these issues have been explored and solved, we are in the position to simulate this processor and investigate its performance. This is the topic of next chapter.

CHAPTER 5

SIMULATION AND PERFORMANCE EVALUATION

The architecture and implementation of micro data flow processor have been described in detail previously. Simulations are performed and results are presented in this chapter. We first describe the processor's specifications in detail. Then, some example programs, like general difference recursive equation, are given to test the correctness and to evaluate the performance of this processor. The potential of deadlock in this static data flow processor is pointed out. Several solutions are presented also.

5.1 Specifications of The Micro Data Flow Processor

The micro data flow processor designed is an 8–bit microprocessor in terms of data and address width. Therefore, its address space is $2^8 = 256$. Since there are two registers per instruction, the instruction memory can accommodate up to 128 instructions. In our design since address FF is used to indicate the no–link situation, therefore, the total valid instruction spaces are 127. As for the ALU, two operation units and two decision units are included in this data flow machine. Operation unit performs integer(signed) addition/subtraction and multiplication. Each operation unit contains a Carry–look–ahead adder and an Array multiplier used to calculate addition/ subtraction and multiplication. Greater than, less than, equal to, NOT, AND and OR are the functions that a decision unit is capable of performing. In terms of the maximum parallelism for computation, only four instructions(two for operations and two for decisions) can be executed at the same instant of time. As a result, the instruction memory is divided into four groups. Each group is called an ***instruction block*** containing 32 instructions. Only one of these 32 instructions in a block can be fired at any time instant. This is achieved through the use of a 32x1 arbiter network. Thus,

there are total of four these 32x1 arbiter networks, forming a 128x4 arbiter network, in the arbitration network. Similarly, there are also 4x128 switch network circuits in both the distribution and control network.

With instruction template presented in section 2.3.1, this 8–bit machine's instruction has a total of 38 bits for a decision packet(token), including two 8–bit input values, two 8–bit destination addresses, one 4–bit opcode and two 1–bit result tags. Appendix B details the opcode, gating code and result tag assignments. As for the operation packet(token), since no result tags are required, only a total 36 bits are needed. There are 10 bits for control packet(token), consisting of a 8–bit destination address, a 1–bit boolean result and a 1–bit result tag. The bit number for data packet(token) increases to 16–bit wide since its result is 8 bits and no result tag is included. The above numbers do not take into considerations of the micropipelines' control signals(request and acknowledge). If they are included, two additional bits are needed for each of the above bit counts. Table 3 summarizes the above discussion.

Table 3: Bit counting for each packet(token)

Packet(Token)	Without control signals(bits)	With control signals(bits)
Operation	36	38
Data	16	18
Decision	38	40
Control	10	12

How does the processor determine to which processing unit an instruction should be assigned during the execution? A 4x4 shuffle network, as mentioned in section 2.3.2, can fulfill this purpose. The first Switch stage, controlled by opcode,

in shuffle network determines if the instruction should be routed to either operation units or decision units. The Switch modules in the third stage of shuffle network are replaced by Toggle modules for the arbitration network. These Toggle modules determine that two operation(decision) units are utilized alternately. Since each of the Switch and the Arbiter module is made of only one micropipeline stage, the 128x4 arbiter network requires 5 micropipeline stages and 4x4 shuffle network 4 stages. A total of 9 stages for arbitration network.

Analogous analysis can be made to determine the numbers of stages for distribution and control networks. Both the distribution network and control network require 10 stages. As for the ALU in our design, both adder/subtracter and multiplier are designed to have 6 micropipeline stages. In a decision unit, the comparator has 6 stages and the logic(NOT, AND and OR) circuit requires 4 stages. Finally an instruction cell itself is implemented with one micropipeline stage. All delays for gates and basic modules used in circuit implementation are from references [28] and [29] and are illustrated in Appendix C,. With these delays given, the latency for each part can be evaluated as shown in Table 4.

A practical question will be always asked: how many transistors are required for this micro data flow processor implementation? Will this processor be manufactured in one single chip or in a chip set? This question can be answered by counting the total number of basic elements(modules) NOT, AND, OR, XOR, Muller C–element, DETFF and Delay this processor has. Table 5 shows the total transistor count and the percentage each basic module contributes. With today's VLSI technology, this micro data flow processor can fit into a single chip with total transistor count less than one million.

As mentioned previously, 32 instructions are grouped in one instruction block and only one of these 32 instructions is capable of firing at any time instant. Our intuition may want to group the first 32 instruction cells into the first instruction

Table 4: Pipeline stages and latency for each part of micro data flow processor

		Pipe–stages	Latency(ns)
Operation Unit (2)	+/–	6	34.67
	X	6	43.17
Decision Unit (2)	Comparator	6	29.67
	Logic	4	27.67
Arbitration network		9	105.35
Distribution network		10	60.34
Control network		10	60.34
Instructions(128)		1	19.36

block, the next consecutive 32 instruction cells as the second block, and so forth. If we arbitrarily assign our instructions(graph nodes) to the instruction memory in sequential order(this is allowed in data flow machine since instruction is activated by the incoming data not Program Counter), some parallelism will be lost. For example, in Figure 27(a) the instructions(nodes) A and C will not be executed in parallel if all these instructions are in the same instruction block. In order to maintain execution parallelism, the nodes A, B and C should, for example, be assigned as instruction 0, 2 and 32, respectively, as shown in Figure 27(b). If we want to prevent a program from scattering around the whole instruction memory, the grouping could be organized in a shuffle manner, as shown in Figure 28. Once this grouping strategy is adopted,

Table 5: Total transistor count and the percentage for each basic element

Element	Transistor	# of element	Total transistors	Percentage
NOT	2	22490	44,980	5%
AND	6	29236	175,416	20%
OR	6	13562	81,372	9%
XOR	10	4032	40,320	4%
MULLER–C	14	2892	40,488	4%
DETFF	28	18204	509,712	57%
DELAY	5	2340	11,700	1%
			903,988	100%

the instructions in Figure 27(b) could be reassigned as shown in Figure 27(c) to closely locate the instructions. This will ease the debugging effort. Here we see that how a compiler assign instructions to memory locations will determine the parallelism achieved by the processor. Therefore, assigning the instructions appropriately will achieve higher spatial parallelism.

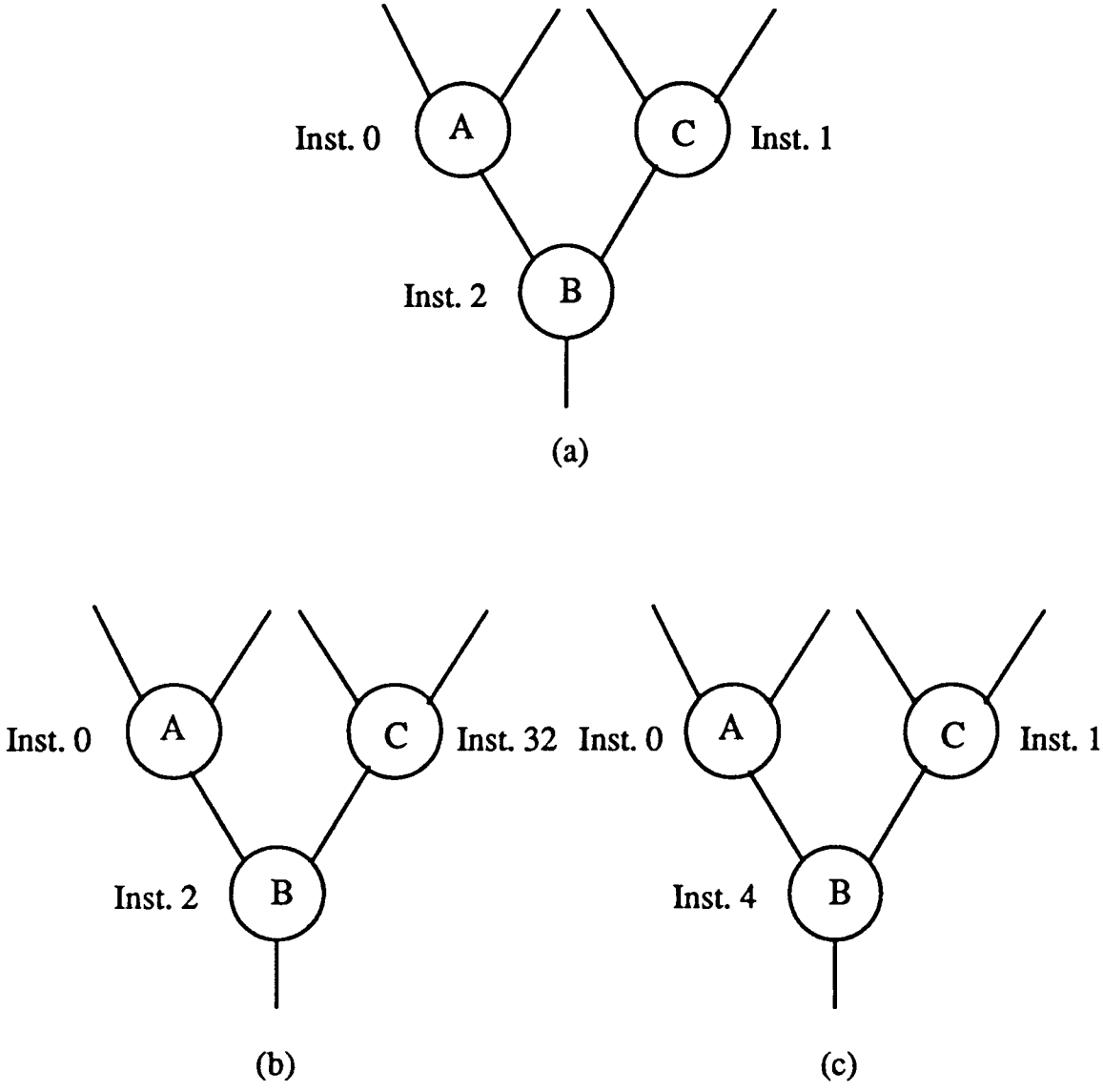
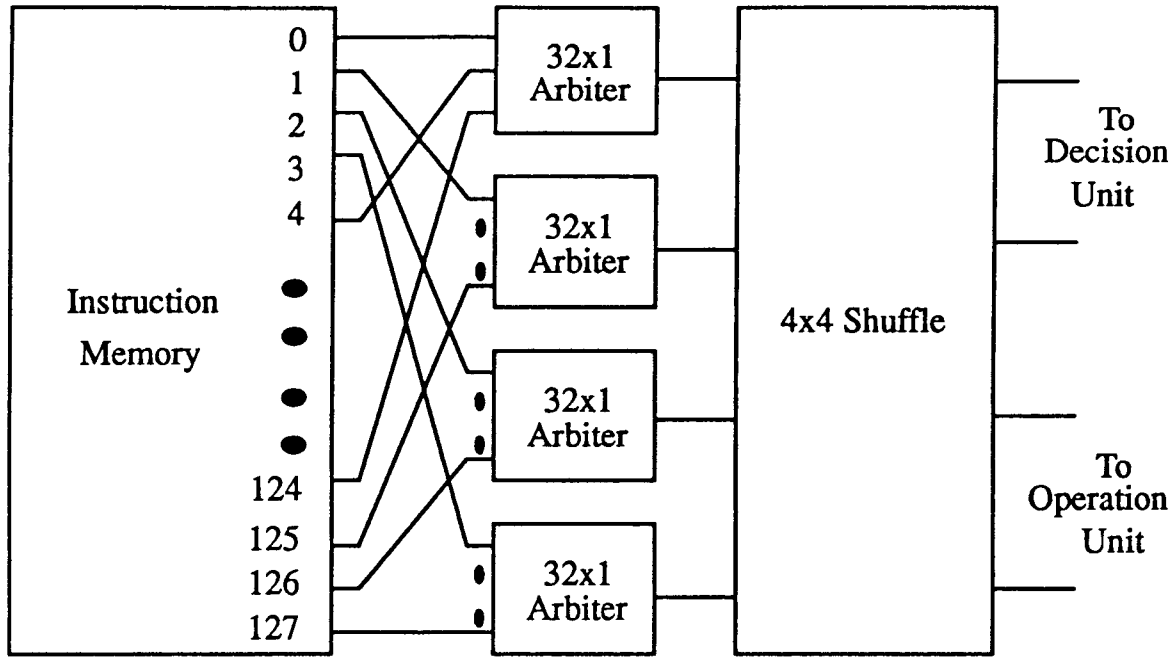


Figure 27: Instruction assignments (a) sequential assignment (b) scattering assignment (c) closer(gathering) assignment



. 0, 4, 8 .....124: first instruction block
. 1, 5, 9 ..... 125: second instruction block
. 2, 6, 10 .....126: third instruction block
. 3, 7, 11 ..... 127: fourth instruction block

Figure 28: Instruction grouping

5.2 Simulation and Performance Evaluation

Two examples are used to test the correctness and to investigate the performance of this processor. These examples are actual programs loaded in the data flow machine implemented. VIEW*logic*'s Viewsim[30] is used to perform the simulation. The first example is a simple calculation of adding and logic OR–ing two numbers. This example allows us to estimate the maximum performance(latency and throughput) this processor can attain. Note that due to the hardware limitation for simulation, only 32 instruction cells are actually included in the instruction memory for our processor. After this modification, arbitration, distribution and control network becomes 7, 8 and 8 stages,

respectively. To reduce the possibility of deadlock(details in next section), 5 stages of FIFO are placed between control(distribution) network and instruction memory.

For the example W = X + Y, ten identical data are fed into X and Y for addition. In Figure 29, Im0o(11) and Im1o(22) are the input data for X and Y, respectively. Im0o9 and Im0o0 are the request and acknowledge signals of micropipelines for input X; Im1o9 and Im1o0 are for Y. Result W is shown on Im63o, which is 33. Simulation starts at 200ns. The first result(the first event on Im63o0) is obtained at 459.79ns; hence, the latency for this addition is 259.79ns(459.79 – 200 = 259.79). Actually, the latency can also be estimated from Table 4 in previous section. For addition, token starts from instruction memory through arbitration network, operation unit(+), distribution network and then back to memory again. Thus the total latency is the sum of each unit's latency. That is, total latency = 19.36 + 105.35 + 34.67 + 60.34 = 219.72ns. Note that the latencies(worse case) shown in Table 4 do not include the bubbled Muller–C delay, which is 2.13ns. Moreover, the 5 stages FIFO for deadlock reduction need to be counted in also. Therefore, if we consider these delays(each stage with one Muller–C), the total latency becomes 219.72 + (1 + 7 + 6 + 8 + 5) * 2.13 = 277.23ns. This number is consistent with the simulation result in certain way since Table 4 is calculated using the worse case assumption. For example, for Instruction memory, the shortest latency and longest latency differs in 10ns. From Figure 29, throughputs(time difference between two events) are 34.81, 34.6, 34.8, 34.6, 34.8, 34.6, 34.8, 34.6 and 34.8ns. In average, it is 34.71ns. That is, every 34.71ns in an average a result will be generated. This is the maximum throughput that this processor can attain for the add operation. Similar analysis can be applied to C = A OR B. Simulation results are shown in Figure 30. Its simulated latency is 249.0ns compared with the 265.97ns latency calaulated from Table 4. Throughputs are the same as that of W = X + Y.

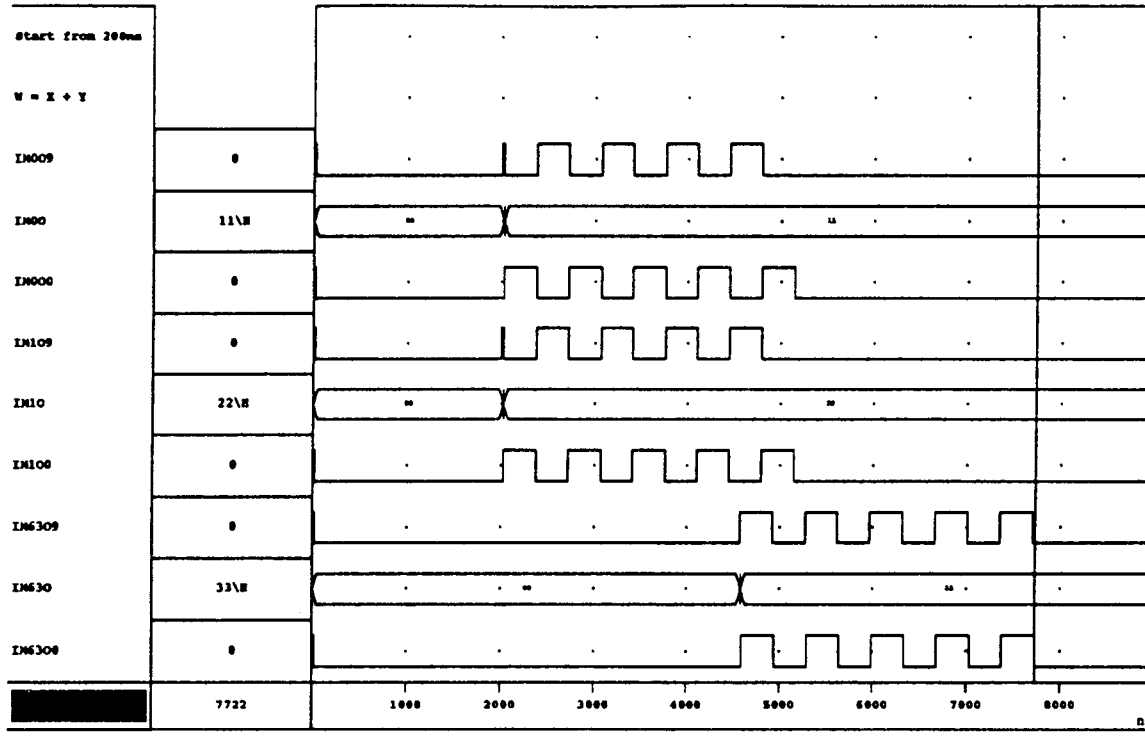


Figure 29: Simulation output waveforms for W = X + Y

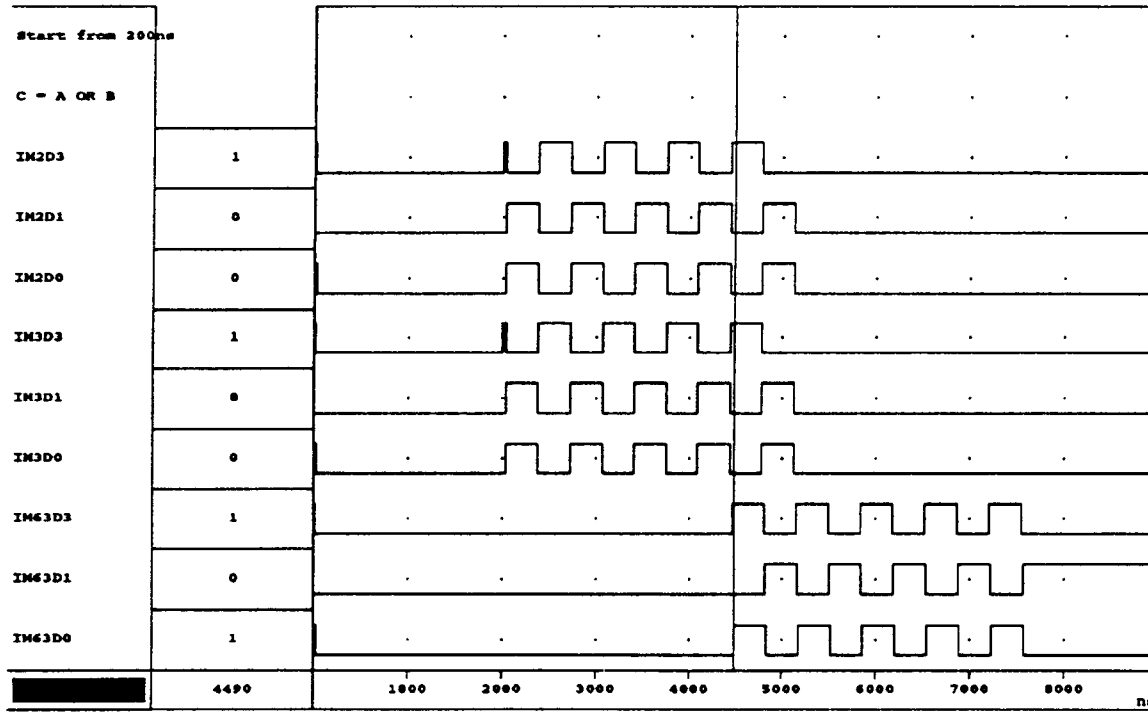


Figure 30: Simulation output waveforms for C = A OR B

A more practical example is a general difference recursive equation. This is widely used in DSP application. We would like to see the performance deviation from the processor's maximum performance for this particular example. A general difference recursive equation can be written as follow:[31]

$$Y(n) = 1/a_0 \left\{ \sum_{k=0}^{M} b_k * X(n-k) - \sum_{i=1}^{N} a_i * Y(n-i) \right\}$$

where X(n): input signal at time n;

X(n – k): input signal at time n – k, $0 \leq k \leq M$;

Y(n): output signal at time n;

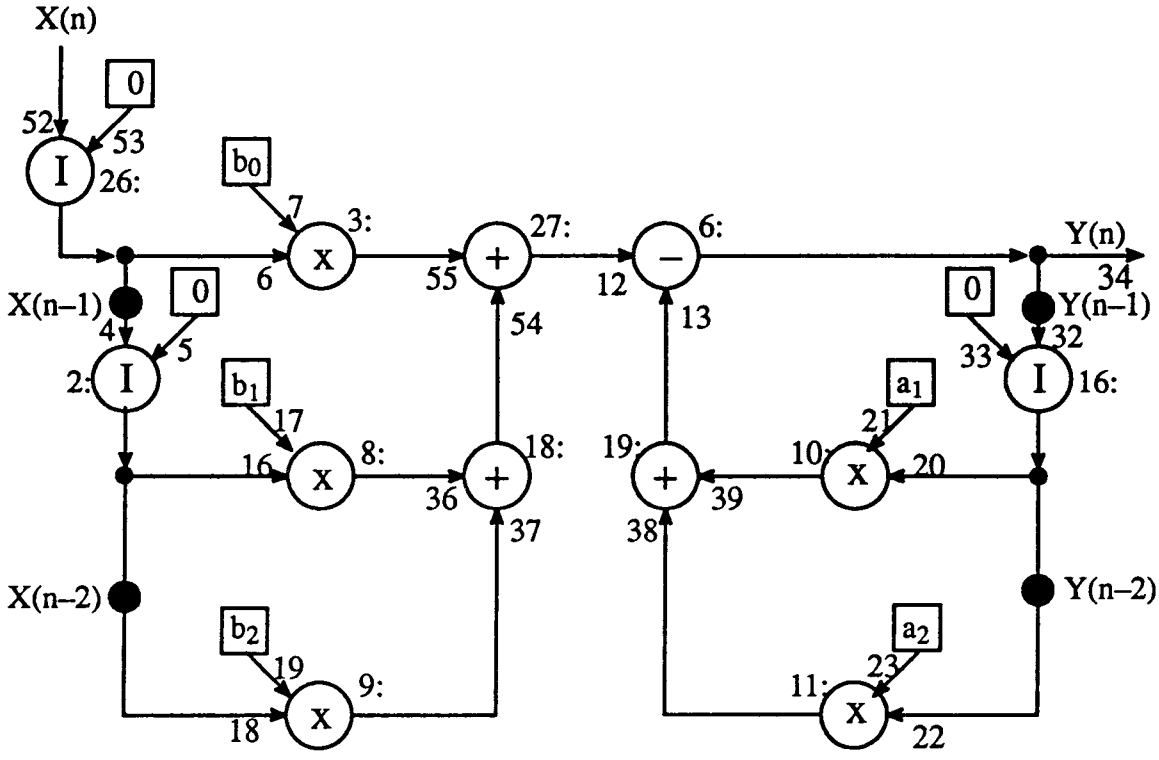
Y(n – i): output signal at time n – i, $1 \leq i \leq N$;

b_k and a_i(including a_0): the weighting coefficients.

This recursive equation expresses the output at time n in terms of previous values of the input and output. How far the current output signal related to previous input and output signals is determined by the values of M and N. For simplicity and convenience, let M = 2, N = 2 and $a_0 = 1$. The above equation, therefore, becomes:

$$Y(n) = b_0 * X(n) + b_1 * X(n-1) + b_2 * X(n-2) - (a_1 * Y(n-1) + a_2 * Y(n-2))$$

The corresponding data flow graph is shown in Figure 31. In this figure, the number followed by ":" represents the instruction number. The number sitting around the arc is the register address in decimal representation. For simulation verification, we further let all coefficients and initial conditions(X(–1), X(–2), Y(–1) and Y(–2)) equal to 1. With all these assignment, Y(n) is equal to 1 if X(n) is equal to 1 for all n. The detail machine codes for this example is shown in Appendix D. The simulation command file is listed in Appendix E.



$b_0 = b_1 = b_2 = a_1 = a_2 = 1$

$X(-1) = X(-2) = Y(-1) = Y(-2) = 1$

□ : constant

● : initial token

• : data link

Figure 31: A data flow graph for a general recursive equation with M = N = 2

Figure 32 illustrates the output waveforms for this example. Im52o shows the input X(n) = 1 and Im34o indicates output Y(n) = 1 also for $0 \leq n \leq 9$. This verifies that the processor is functioning correctly. The first result comes out at 1992.59ns when input starts at 200ns. Therefore, the latency is 1992.59 – 200 = 1792.59ns. Throughputs are obtained as 1007.41, 1045.4, 1043, 1045.4, 1043, 1045.4, 1043, 1045.4 and 1043. It is 1040.1ns in average compared with maximum 34.71ns throughput stated above. There are three major reasons for this performance degradation.

(1) Feedback loop: There is a feedback loop at the right hand side for the data flow graph in Figure 31. This feedback loop only allows one token flowing around this loop according to the firing rule. Therefore, parallelism due to pipelining is lost, as mentioned in chapter 2. Compared with right hand side graph in Figure 31 for instruction 10(IM10I39), instruction 3(IM3I39), which is not in feedback loop, takes the advantage of pipelining.

(2) Limit processing unit: This processor has only two operation units for addition and multiplication operations. However, the following five multiplication instructions(3, 8, 9, 10, 11) could potentially be executed in parallel. This leads to the loss of three possible parallelism. O2P37 shows one of the operation unit utilization waveform.

(3) Unbalanced match: As mentioned in chapter 2, one of the arbitration network purpose is to match the number of instruction cells to the number of processing units. Due to the limit processing unit, this arbitration network must arbitrate all the input inquiring. The waveform of IC39 shows this arbitration. Unfortunately, arbitration network has the longest latency as shown in Table 4, hence, becoming the bottleneck. One way to become a more balance match is to incorporate more operation units; this, however, also increase the size of arbitration network. Since Arbiter is very costly in terms of hardware, this increasing will cost hardware very

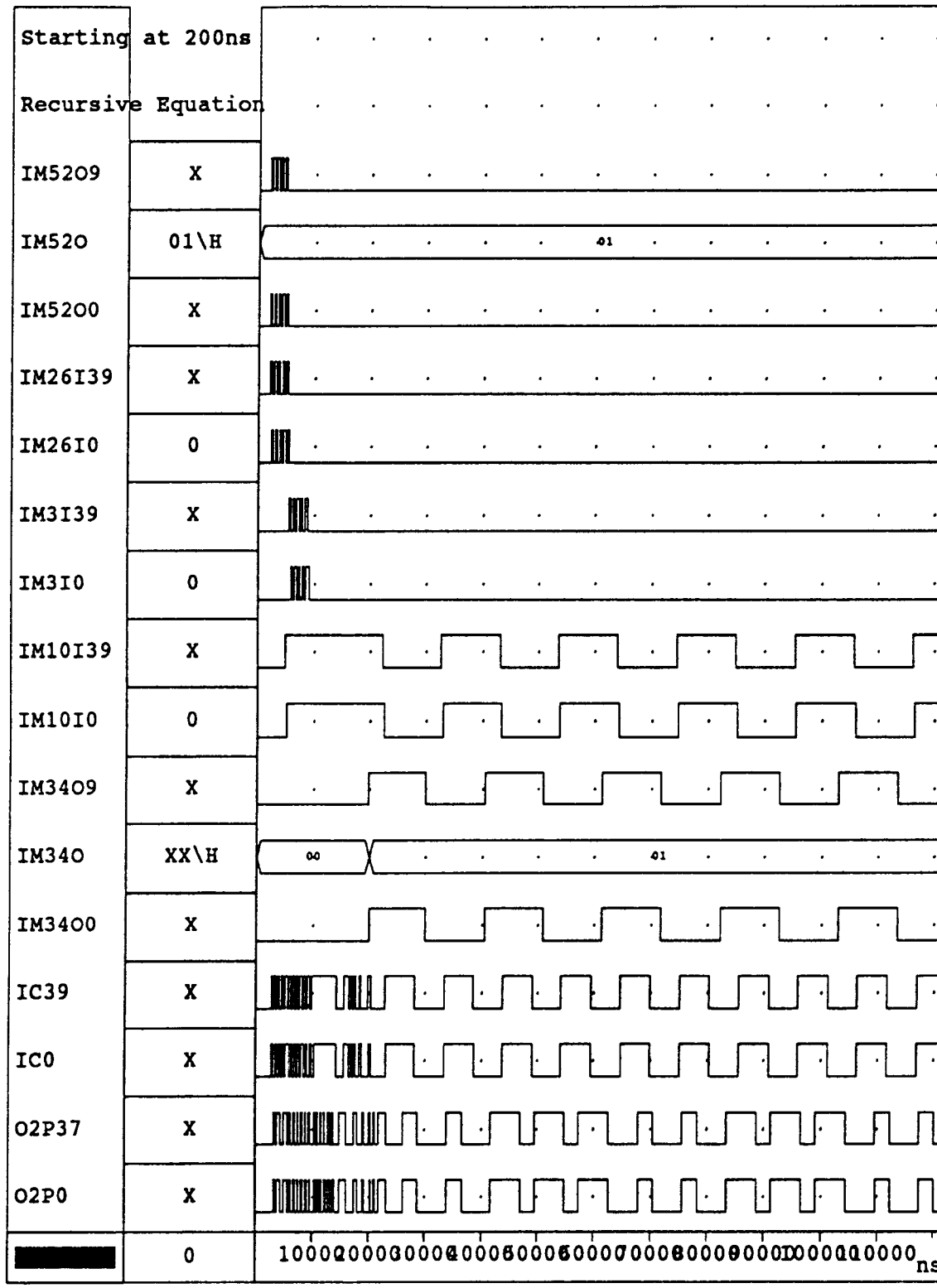


Figure 32: Output waveforms for the general recursive equation with M = N = 2

much. It is interesting to have further research in this topic for the variation of performance/cost ratio.

The other interesting observation is made on the pattern of the throughputs of these two examples. They are restated below.

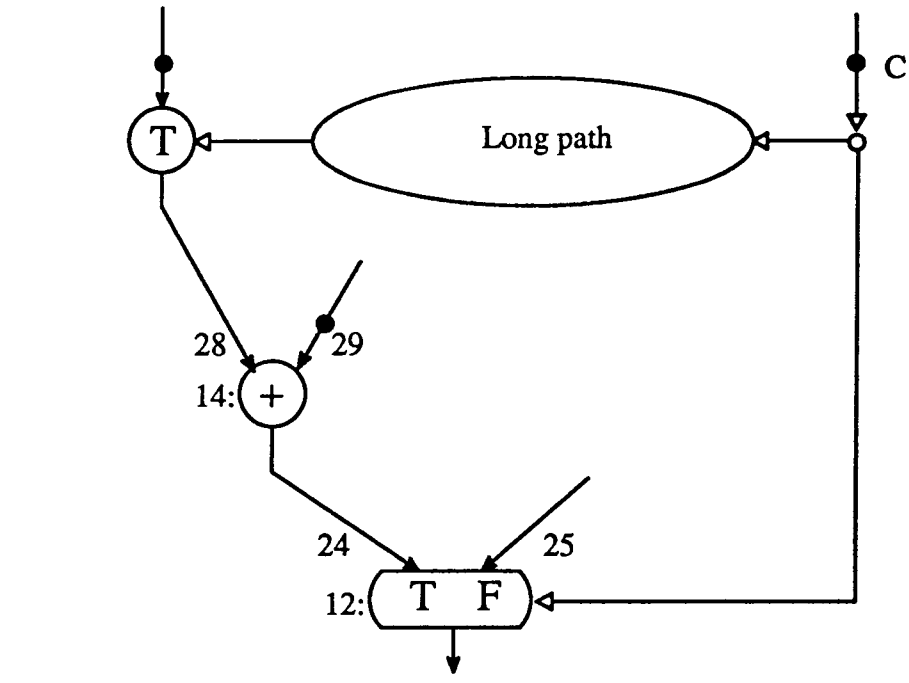
1st: 34.81, 34.6, 34.8, 34.6, 34.8, 34.6, 34.8, 34.6, 34.8

2nd: 1007.41, 1045.4, 1043, 1045.4, 1043, 1045.4, 1043, 1045.4, 1043

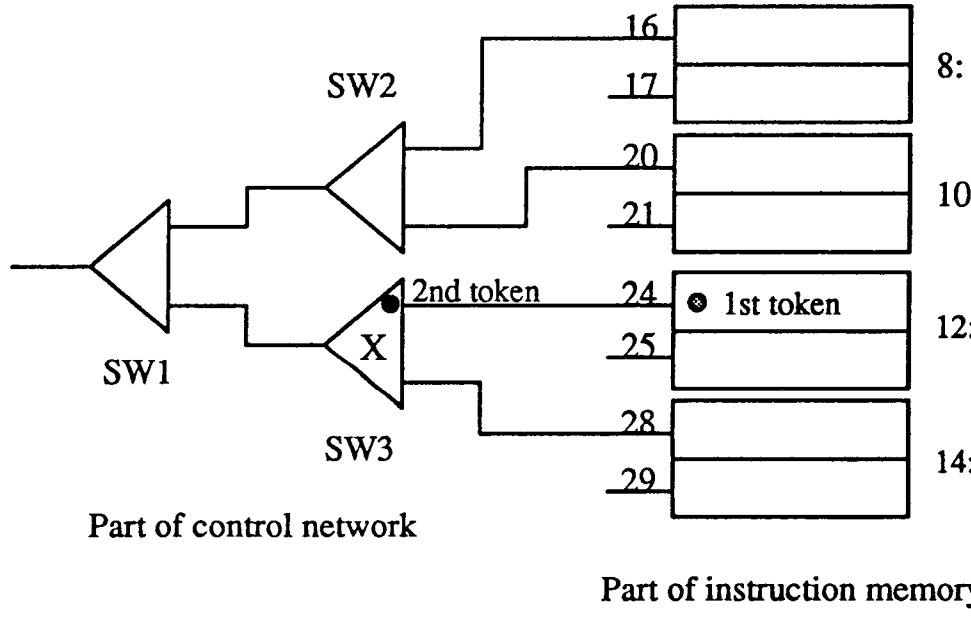
The throughput repeats alternately. There might be some reasons for this pattern. If the cause for this pattern has been found, it is possible to create a model to predict the processor performance or any other circuits implemented using micropipelines. This is left as a future work.

5.3 Deadlock

As mentioned in chapter 2, the control network in data flow processor consists of a Shuffle network and a Switch network. Under certain situations, this Switch network will cause ***deadlock***. The following example, as shown in Figure 33, demonstrates how a deadlock occurs. A data flow graph is shown in Figure 33(a). This graph has a control token C, which is duplicated into two tokens, one for the True gate and one for the Merge. Assume the token to True gate has longer path than that of the token to Merge. Therefore, the control token will arrive Merge(address 24) earlier than True gate(address 28). The first control token will enter Merge since it is empty initially. Now if second control token arrives Merge still earlier than the first control token to True gate, the second control token will sit on SW3, as illustrated in Figure 33(b). When this happens, the SW3 becomes a dead Switch, designated by X, until Merge is fired. It implies no control token can enter SW3 anymore. Deadlock occurs under this situation. This is because Merge can not fire unless data token arrives address 24. However, this data token can not be generated



(a)



(b)

Figure 33: Deadlock in micro data flow processor (a) data flow graph
(b) architecture causing deadlock

until the first control token arrives address 28. Unfortunately, the first control token can no longer enter address 28 since SW3 is dead(blocked) right now. When deadlock occurs, this processor will no longer function.

This deadlock situation will potentially happen when two paths are not balanced in a data flow graph. When the difference between these two paths are large enough such that the shorter path has accumulated too many tokens(resulting certain Switch is blocked) while the first token for the longer path has not come out yet, deadlock may happen. This is true even for a data flow graph containing neither Merge nor True/False gate. The more fundamental solution to deadlock is to change the processor architecture. According to our network architecture, since each Switch and Arbiter modules are implemented in one stage, it is equivalent to allowing tokens to accumulate on arcs in a data flow graph. This is the reason why deadlock may happen. Therefore, instead of allowing accumulating tokens on arcs, if we could ***save*** the accumulated tokens on some temporary space and release them whenever possible, deadlock will not happen. This is closer to the architecture of a dynamic data flow processor. Some tricks, however, can be used to reduce the possibility of deadlock without changing the architecture. One of them is to balance the data flow graph by inserting Identity operators. Another is to insert some buffers between control(and distribution) network and instruction memory. The more buffers are inserted, the less chance the Switch will be blocked. The price is, of course, more hardware.

5.4 Summary

The micro data flow processor we implemented is an 8 bit processor with respective to its data and address. This processor has four processing units — two operation units and two decision units. After simulation, this processor shows it can generate a new result(addition) every 34.71ns. This is its maximum throughput for a simple addition. A data flow graph with feedback loop, for example, will degradate its performance. A potential problem called deadlock might happen if a data flow graph is not well balanced. In terms of architecture, this is because the temporary unfired tokens stay in the control path to block future tokens to enter. If the unfired tokens can be saved temporarily at some place and keep the control path clean, then the problems could be solved. From this simulation, some throughput patterns are observed. They repeat alternately. This observation could help us to create a performance evaluation model for the circuit using micropipeline implementation.

CHAPTER 6
CONCLUSION AND FUTURE WORK

Up to now, the motivation, concept, implementation technique and simulation results for the micro data flow processor have been shown. It is appropriate to stop here temporarily by drawing some conclusions for our current research. This does not mean that there will be no further exploration necessary. Instead, some important and interesting areas will be pointed out in this chapter for future work.

6.1 Conclusions

Control flow computers do not easily express inherent instruction parallelism due to the fact that instruction execution is guided by the program counter which is increased sequentially. It is possible to extract the instruction parallelism from a sequential program. However, this will need both the aid of a complex compiler and an intricate controller hardware to detect and resolve the dependencies. An attractive alternative is to use data flow computers featuring no program counter. Data flow computers can explore the parallelism among instructions without additional dependency analysis.

In terms of the implementation technique, the synchronous clocked–logic is the dominated design methodology for the current digital system due to its simplicity in circuit realization. This is, however, no longer an important factor. As the clock frequency increases, clock skew will cause wrong signals being latched which leads to incorrect results or even system crashes. An alternative design philosophy called asynchronous(or self–timed) design emerged. It features no clock skew, low noise, zero stand–by power and low heat generation. Moreover, it also provides the property of modularity and composibility for easing module interconnection and environment

interfacing. With these properties, the product life will be prolonged since its performance can be improved easily and efficiently once a critical part of the system is improved.

Micropipeline is one of the self–timed implementation technique. Due to the similarity between the micropipelines and data flow concept, our research is to implement a data flow processor using micropipelines. We call the processor with these combination a "micro data flow processor". The modularity and composibility property does help in designing this micro data flow processor. For example, to design all the networks in the micro data flow processor, once the switch and arbiter are designed correctly, all the networks can be obtained in a snap by direct interconnection of these two modules. This will reduce a great deal of design time. This research proves that with only some basic event logic modules and simple standard control circuit, micropipelines can easily, readily and efficiently implement a general–purpose data flow processor. It is almost impossible to implement this kind of data flow processor using the traditional asynchronous design methodology. Moreover, micropipeline is especially powerful and suitable for the hierarchical design which is highly desirable.

6.2 Future Work

Our current research is a start in familiarizing ourselves with the self–timed system design using micropipelines. We have also gain many conceptual understanding at the system level as we combine the micropipeline technique with data flow architecture. Although this learning experience is both successful and educated, a great deal of further exploration still needs to be done in order to develop more mature and practical implementation techniques. The following sub–sections summarize the possible

future works categorized in four aspects — data flow processors, micropipelines, simulation and CAD tool development.

6.2.1 Data flow processors

(1) In our research, the data flow architecture we implemented was chosen somewhat arbitrarily. In order to avoid deadlock and to have higher performance, a new architecture should be carefully designed.

(2) Given a static data flow graph, majority of the parallelism will be lost within a feedback loop. Further research in breaking down or unfolding the loop is needed to increase the instruction parallelism.

(3) Currently, only eight primitives are used to construct a data flow graph. Although these primitives are sufficient for the general applications, more primitives could be added to form more powerful basic modules and to ease the data flow graph construction.

(4) In terms of firing rule, the data flow processor we implemented is static. If more parallelism is expected, a dynamic data flow processor should be implemented.

6.2.2 Micropipelines

(1) This thesis concentrated on implementing a general–purpose data flow processor using micropipelines. The other attractive application is to implement special–purpose processors by direct mapping a data flow graph to hardware. A typical application will be the DSP area.

(2) The delay elements in micropipelines are fixed and should be calculated in advance by circuit designer. In considering the processing and environmental variation, these delays are always designed larger than the worst case for safety reason,

leading to performance degradation. A different logic family called Enable/Disable CMOS Differential Logic(ECDL) can be used to remedy this problem[32].

(3) To predict the performance of a circuit made of micropipelines, the performance evaluation model for micropipelines should be established. This model creation will help engineers design higher performance circuits before its actual implementation.

6.2.3 Simulation

In our original design, a micro data flow processor with 128 instructions in memory occupies around 170M bytes for its simulation wirelist file. This big file exceeds our hardware capability. For simulation purpose, the Very High Speed Integrated Circuits Hardware Description Language(VHDL), can be used to replace some basic schematic module design to reduce the size of simulation wirelist file. By using behavior models in VHDL simulation time can be reduced[33].

6.2.4 Computer-Aided Design(CAD) tool development

Eventually, one of our goals is to automate the design process mainly for a special purpose data flow processor. In order to do this, a set of CAD tools will be developed. Followings are the tool specifications which we are interested in.

1> Schematic entry: A schematic capture tool for graphic entry of a data flow graph in structural description. This entry level should also allow to call sub-data flow graph in hierarchical manner. Behavior description of user-defined primitives should also be included to increase the design flexibility.

2> Simulation: After a data flow graph is constructed, it could be simulated at either the token level or the logic level. This tool should have the ability to detect possible deadlocks in a defective data flow graph.

3> Performance analysis: The simulation results should also illustrate the busy and idle cycle for each node in a data flow graph to understand the hardware utilization rate and performance.

4> Implementation: This tool should have the ability to map a data flow graph to micropipelines directly. It should be further mapped down to a lower transistor level in optimized manner by considering the area occupation, power consumption, cost and so forth. If ECDL implementation is desired, auto–synthesis from logic equation to ECDL is necessary.

With these tools, engineers can focus on high level(data flow graph) design without wasting time in detail circuit design. This will also shorten the design cycle.

BIBLIOGRAPHY

[1] J. Backus, "Can Programming Be Liberated From The von Neumann Style? A Functional Style And It's Algebra Of Programs", Comm. of the ACM, 21:8, pp613–641, 1978.

[2] C.V. Ramamoorthy and H.F. Li, "Pipeline Architecture", ACM computing surveys, 9:1, pp61–102, March 1977.

[3] P.M.Kogge, "The Architecture Of Pipelined Computers", Mcgraw–Hill, New York, pp220–225, 1981.

[4] A. Smith and J. Lee, "Branch Prediction Strategies And Branch Target Buffer Design", Computer, 17:1, pp6–22, January 1984.

[5] J.E. Smith, "Dynamic Instruction Scheduling And The Astronautics ZS–1", Computer, 22:7, pp21–35, July 1989.

[6] J.E. Smith and A.R. Plezkun, "Implementing Precise Interrupts In Pipelined Processors", IEEE Trans. on Computers, 37:5, pp562–573, May 1988.

[7] Arvind and R.A. Iannucci, "A Critique Of Multiprocessing von Neumann Style", Proc. of the 10th annual Int'l. symp. on computer architecture, pp426–436, 1983.

[8] S. Dasgupta, "Computer Architecture A Modern Synthesis", volume 2, Wiley, pp281–283, 1989.

[9] W.B. Ackerman, "Data Flow Language", IEEE Computer, pp15–25, February 1982.

[10] J.B. Dennis, "Data Flow Supercomputers", IEEE Computer, pp48–56, November 1980.

[11] Arvind and D.E. Culler, "Dataflow Architectures", Annual Review of Computer Science, Volume 1, pp225–253, Annual Reviews Inc., 1986.

[12] Fumiyasu ASAI et al, "Self–Timed Clocking Design For A Data–Driven Microprocessor", IEICE Trans. Vol. E 74 No.11, pp3757–3764, November 1991.

[13] D. Pountain, "Computing Without Clocks", BYTE, pp145–150, January 1993.

[14] A.L. Davis, R.M. Deller, "Data Flow Program Graphs", IEEE Computer, pp26–41, February 1982.

[15] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data–Flow Processor", The second annual symposium on computer architecture, pp126–132, 1975.

[16] G. Broomell and J. R. Heath, "Classification Categories and Historical Development of Circuit Switching Topologies", Computing Surveys, Vol. 15, No. 2, pp95–133 June 1983.

[17] P.Y. Chen et al., "Interconnection Networks Using Shuffles," Computer, 14:12, pp55–64, 1981.

[18] C.H. Lau, "Data Flow Approach To Self–Timed Logic In VLSI", ISCAS, pp.479–482, 1988.

[19] S. Komori et al, "Data–Driven Microprocessor", IEEE MICRO, pp.45–59,June 1989.

[20] F. Asai et al,"Self–Timed Clocking Design For A Data–Driven Microprocessor", IEICE Transactions Vol. E 74, No.11, pp.3757–3764, November 1991.

[21] T.H Meng,"Synchronization Design For Digital Systems", Kluwer Academic Publishers, 1991.

[22] I.E. Sutherland, "Micropipelines", Communications of the ACM, Vol.32 No.6, pp.720–738, June 1989.

[23] R.E. Muller, "Sequential Circuits", Chapter 10, In Switching Theory, Vol 2, Wiley, NY, 1965.

[24] S.L. Lu and M. Ercegovac, "A Novel CMOS Implementation of Double–Edge–Triggered Flip–Flops", IEEE Journal of Solid–State Circuits, Vol.25, No.4, pp.1008–1010, August 1990.

[25] C.M. Chang and S.L.Lu, "Micro Data Flow Processors,"Proceedings of IEEE Pacific Rim Conference on Comm., Computers and Signal Processing, pp.149–153, 1993.

[26] S.L.Lu and L.Merani, "Micro Data Flow," Proceedings of the 5th IEEE ASIC Conference, Rochester New York, pp.301–304, Sept., 1992.

[27] L.Merani and S.L.Lu, "A Self–timed Approach to VLSI Digital Filter Design," Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp.402–406, 1993.

[28] The HP C34000 Standard Cell Library Data Manual (Draft). Hewlett–Packard Company, Integrated Circuit Business Division, June 1992.

[29] R.F. Sproull and I.E. Sutherland, "Asynchronous Systems," Volume I: Introduction, Sutherland, Sproull & Associates, Inc., Sep. 1986.

[30] VIEW*logic* Work*view* Manual, VIEWlogic Systems, Inc., 1991.

[31] A.V. Oppenheim, A.S. Willsky and I.T. Young, "Signals and Systems," Prentice–Hall, pp.108–117, 1983.

[32] S.L. Lu, "Self–timed Arithmetic Structures in CMOS Differential Logic," Ph.D Thesis, Computer Science Department, UCLA, 1991.

[33] S.L.Lu and C.M.Chang, "Modelling of a Self–timed DataFlow Processor in VHDL," to appear in the Proceedings of the 6th IEEE ASIC Conference, Rochester New York, Sept., 1993.

APPENDICES

Appendix A Corresponding Names For S1, S2 And S3

Instruction number: I

Address of corresponding instruction register 1: 2 * I

Address of corresponding instruction register 2: 2 * I + 1

SopopN corresponds S1 and S2 in distribution network.

SdedeN corresponds S1 and S2 in control network.

SmumuN corresponds S3 in distribution network.

No S3 in control network.

The N in the above names can be obtained from the following table.

Table 6: Corresponding Names for S1, S2 and S3

N =		First Register	Second Register
I : Even	S1	I + 1	I + 33
	S2	I + 2	I + 34
	S3	(I / 2) + 1	(I / 2) + 17
I : Odd	S1	I + 64	I + 96
	S2	I + 65	I + 97
	S3	((I – 1) / 2) + 33	((I – 1) / 2) + 49

I : Instruction number, 0 <= I <= 31

Appendix B Opcode, Gating Code And Result Tag Assignments

The following table is the opcode $C_3C_2C_1C_0$, gating code and result tag representations for ALU operations.

Table 7: Opcode, gating code and result tag representations

			$C_3C_2C_1C_0$
Operation Unit	Operator	+	0000
		–	0001
		x	0010
Decision Unit	Decider	>	1000
		<	1001
		=	1010
	Boolean	NOT a_0	1100
		NOT b_0	1101
		OR	1110
		AND	1111

The following table is gating code $G_3G_2G_1$ representation.

Gating Code	$G_3G_2G_1$
Cons	100
No	000
True	010
False	110
True_m	011
False_m	111

Result tag t = 0 ==> Value packet(Value flags V)

Result tag t = 1 ==> Control packet(Gate flags F_2F_1)

Boolean Value	F_2F_1
Reset	00
True	01
False	11

Appendix C Basic Module And Gate Delays

Table 8: The basic module and gate delays.

Gates/Modules		TPLH(ns)	TPHL(ns)
1	NOT	0.16	0.13
2	AND(2)	0.27	0.48
3	AND(3)	0.28	0.62
4	NAND(2)	0.26	0.1
5	OR(2)	0.58	0.42
6	OR(3)	0.81	0.46
7	NOR(2)	0.19	0.3
8	D Flip–Flop	0.79	0.89
9	DETFF	1	1
10	XOR	0.48	0.35
11	C–element	2.13	2.13
12	Arbitor	13.33	13.33
13	Call(Req./Ack.)	3.2 / 5.34	3.2 / 5.34
14	Toggle	1	1
15	Select	1	1

Appendix D Instruction Templates For Recursive Equation

	Gating Code	Value Flag	Value	Code	
R1	Gating Flag			Desti. Addr.	Result Tag
R2	Gating Code	Value Flag	Value	Desti. Addr.	Result Tag
	Gating Flag				

4	000	Value Flag	Value	0000		22	000	Value Flag	Value	0010	
	Gating Flag			16	0		Gating Flag			38	0
5	100	Value Flag	0	18	0	23	100	Value Flag	1	FF	0
	Gating Flag						Gating Flag				
6	000	Value Flag	Value	0010		32	000	Value Flag	Value	0000	
	Gating Flag			55	0		Gating Flag			20	0
7	100	Value Flag	1	FF	0	33	100	Value Flag	0	22	0
	Gating Flag						Gating Flag				
12	000	Value Flag	Value	0001		36	000	Value Flag	Value	0000	
	Gating Flag			34	0		Gating Flag			54	0
13	000	Value Flag	Value	32	0	37	000	Value Flag	Value	FF	0
	Gating Flag						Gating Flag				
16	000	Value Flag	Value	0010		38	000	Value Flag	Value	0000	
	Gating Flag			36	0		Gating Flag			13	0
17	100	Value Flag	1	FF	0	39	000	Value Flag	Value	FF	0
	Gating Flag						Gating Flag				
18	000	Value Flag	Value	0010		52	000	Value Flag	Value	0000	
	Gating Flag			37	0		Gating Flag			4	0
19	100	Value Flag	1	FF	0	53	100	Value Flag	0	6	0
	Gating Flag						Gating Flag				
20	000	Value Flag	Value	0010		54	000	Value Flag	Value	0000	
	Gating Flag			39	0		Gating Flag			12	0
21	100	Value Flag	1	FF	0	55	000	Value Flag	Value	FF	0
	Gating Flag						Gating Flag				

Appendix E Command File For Recursive Equation Simulation

```
| This is the simulation program for the example of general difference recursive equation in
chapter 5.

vector d1e d1e[39:0]
vector d2e d2e[39:0]
vector dae dae[11:0]
vector dbe dbe[11:0]
vector dce dce[11:0]
vector dde dde[11:0]
vector o1p o1p[37:0]
vector o2p o2p[37:0]
vector oap oap[17:0]
vector obp obp[17:0]
vector ocp ocp[17:0]
vector odp odp[17:0]
vector ia  ia[39:0]
vector ib  ib[39:0]
vector ic  ic[39:0]
vector id  id[39:0]
vector dwe dwe[11:0]
vector dxe dxe[11:0]
vector dye dye[11:0]
vector dze dze[11:0]
vector owp owp[17:0]
vector oxp oxp[17:0]
vector oyp oyp[17:0]
vector ozp ozp[17:0]

vector sdede sdede[128:1]
vector sopop sopop[128:1]
vector smumu smumu[64:1]
assign sdede 0
assign sopop 0
assign smumu 0

|inst. 0
vector im0g  im0g[3:1]          |g3g2g1
vector im1g  im1g[3:1]
```

```
assign im0g  000
assign im1g  000

|inst 1
vector im2g  im2g[3:1]      |g3g2g1
vector im3g  im3g[3:1]
assign im2g  000
assign im3g  000

|inst 2
vector im2i  im2i[38:23]    | two input operands for instruction 2
vector im4o  im4o[8:1]      | operation input operand for register 1
vector im5o  im5o[8:1]      | operation input operand for register 2
vector im4d  im4d[2:1]      | decision input operand for register 1
vector im5d  im5d[2:1]      | decision input operand for register 2
vector im4g  im4g[3:1]      | gating code for register 1 g3g2g1
vector im5g  im5g[3:1]      | gating code for register 2
vector va5   im2i[30:23]    | second input operand for instruction 2
vector c2    im2i[22:19]    | opcode c3c2c1c0
vector dd2   im2i[18:3]     | two destination addresses
vector t2    im2i[2:1]      | two result tags
l     va5
h     sopop4
assign im4g  000
assign im5g  100
assign c2    0000
assign dd2   1012\h
assign t2    00

|inst 3
vector im3i  im3i[38:23]
vector im6o  im6o[8:1]
vector im7o  im7o[8:1]
vector im6d  im6d[2:1]
vector im7d  im7d[2:1]
vector im6g  im6g[3:1]      |g3g2g1
vector im7g  im7g[3:1]
vector va7   im3i[30:23]
vector c3    im3i[22:19]    |c3c2c1c0
vector dd3   im3i[18:3]
vector t3    im3i[2:1]
l     va7
h     im3i23
```

```
assign im6g  000
assign im7g  100
assign c3    0010
assign dd3   37ff\h
assign t3    00

linst. 4
vector im8g  im8g[3:1]       |g3g2g1
vector im9g  im9g[3:1]
assign im8g  000
assign im9g  000

linst. 5
vector im10g  im10g[3:1]      |g3g2g1
vector im11g  im11g[3:1]
assign im10g  000
assign im11g  000

linst. 6
vector im6i  im6i[38:23]
vector im12o  im12o[8:1]
vector im13o  im13o[8:1]
vector im12d  im12d[2:1]
vector im13d  im13d[2:1]
vector im12g  im12g[3:1]      |g3g2g1
vector im13g  im13g[3:1]
vector va13   im6i[30:23]
vector c6    im6i[22:19]     |c3c2c1c0
vector dd6   im6i[18:3]
vector t6    im6i[2:1]
assign im12g  000
assign im13g  000
assign c6    0001
assign dd6   2220\h
assign t6    00

linst. 7
vector im14g  im14g[3:1]      |g3g2g1
vector im15g  im15g[3:1]
assign im14g  000
assign im15g  000

linst. 8
vector im8i  im8i[38:23]
```

```
vector im16o  im16o[8:1]
vector im17o  im17o[8:1]
vector im16d  im16d[2:1]
vector im17d  im17d[2:1]
vector im16g  im16g[3:1]      |g3g2g1
vector im17g  im17g[3:1]
vector va17   im8i[30:23]
vector c8     im8i[22:19]     |c3c2c1c0
vector dd8    im8i[18:3]
vector t8     im8i[2:1]
l     va17
h     im8i23
assign im16g  000
assign im17g  100
assign c8     0010
assign dd8    24ff\h
assign t8     00

|inst. 9
vector im9i   im9i[38:23]
vector im18o  im18o[8:1]
vector im19o  im19o[8:1]
vector im18d  im18d[2:1]
vector im19d  im19d[2:1]
vector im18g  im18g[3:1]      |g3g2g1
vector im19g  im19g[3:1]
vector va19   im9i[30:23]
vector c9     im9i[22:19]     |c3c2c1c0
vector dd9    im9i[18:3]
vector t9     im9i[2:1]
l     va19
h     im9i23
h     sopop74
assign im18g  000
assign im19g  100
assign c9     0010
assign dd9    25ff\h
assign t9     00

|inst. 10
vector im10i  im10i[38:23]
vector im20o  im20o[8:1]
```

```
vector im21o  im21o[8:1]
vector im20d  im20d[2:1]
vector im21d  im21d[2:1]
vector im20g  im20g[3:1]      |g3g2g1
vector im21g  im21g[3:1]
vector va21   im10i[30:23]
vector c10    im10i[22:19]    |c3c2c1c0
vector dd10   im10i[18:3]
vector t10    im10i[2:1]
l     va21
h     im10i23
assign im20g  000
assign im21g  100
assign c10    0010
assign dd10   27ff\h
assign t10    00

|inst. 11
vector im11i  im11i[38:23]
vector im22o  im22o[8:1]
vector im23o  im23o[8:1]
vector im22d  im22d[2:1]
vector im23d  im23d[2:1]
vector im22g  im22g[3:1]      |g3g2g1
vector im23g  im23g[3:1]
vector va23   im11i[30:23]
vector c11    im11i[22:19]    |c3c2c1c0
vector dd11   im11i[18:3]
vector t11    im11i[2:1]
l     va23
h     im11i23
h     sopop76
assign im22g  000
assign im23g  100
assign c11    0010
assign dd11   26ff\h
assign t11    00

|inst. 12
vector im24g  im24g[3:1]      |g3g2g1
vector im25g  im25g[3:1]
```

```
assign im24g  000
assign im25g  000

linst. 13
vector im26g  im26g[3:1]      |g3g2g1
vector im27g  im27g[3:1]
assign im26g  000
assign im27g  000

linst. 14
vector im28g  im28g[3:1]      |g3g2g1
vector im29g  im29g[3:1]
assign im28g  000
assign im29g  000

linst. 15
vector im30g  im30g[3:1]      |g3g2g1
vector im31g  im31g[3:1]
assign im30g  000
assign im31g  000

linst. 16
vector im16i  im16i[38:23]
vector im32o  im32o[8:1]
vector im33o  im33o[8:1]
vector im32d  im32d[2:1]
vector im33d  im33d[2:1]
vector im32g  im32g[3:1]      |g3g2g1
vector im33g  im33g[3:1]
vector va33   im16i[30:23]
vector c16    im16i[22:19]    |c3c2c1c0
vector dd16   im16i[18:3]
vector t16    im16i[2:1]
l     va33
h     sopop18
assign im32g  000
assign im33g  100
assign c16    0000
assign dd16   1416\h
assign t16    00

linst. 17
vector im34g  im34g[3:1]      |g3g2g1
vector im35g  im35g[3:1]
```

```
assign im34g  000
assign im35g  000

linst. 18
vector im18i  im18i[38:23]
vector im36o  im36o[8:1]
vector im37o  im37o[8:1]
vector im36d  im36d[2:1]
vector im37d  im37d[2:1]
vector im36g  im36g[3:1]      lg3g2g1
vector im37g  im37g[3:1]
vector va37   im18i[30:23]
vector c18    im18i[22:19]    lc3c2c1c0
vector dd18   im18i[18:3]
vector t18    im18i[2:1]
assign im36g  000
assign im37g  000
assign c18    0000
assign dd18   36ff\h
assign t18    00

linst. 19
vector im19i  im19i[38:23]
vector im38o  im38o[8:1]
vector im39o  im39o[8:1]
vector im38d  im38d[2:1]
vector im39d  im39d[2:1]
vector im38g  im38g[3:1]      lg3g2g1
vector im39g  im39g[3:1]
vector va39   im19i[30:23]
vector c19    im19i[22:19]    lc3c2c1c0
vector dd19   im19i[18:3]
vector t19    im19i[2:1]
assign im38g  000
assign im39g  000
assign c19    0000
assign dd19   0dff\h
assign t19    00

linst. 20
vector im40g  im40g[3:1]      lg3g2g1
vector im41g  im41g[3:1]
```

```
assign im40g  000
assign im41g  000

linst. 21
vector im42g  im42g[3:1]      lg3g2g1
vector im43g  im43g[3:1]
assign im42g  000
assign im43g  000

linst. 22
vector im44g  im44g[3:1]      lg3g2g1
vector im45g  im45g[3:1]
assign im44g  000
assign im45g  000

linst. 23
vector im46g  im46g[3:1]      lg3g2g1
vector im47g  im47g[3:1]
assign im46g  000
assign im47g  000

linst. 24
vector im48g  im48g[3:1]      lg3g2g1
vector im49g  im49g[3:1]
assign im48g  000
assign im49g  000

linst. 25
vector im50g  im50g[3:1]      lg3g2g1
vector im51g  im51g[3:1]
assign im50g  000
assign im51g  000

linst. 26
vector im26i  im26i[38:23]
vector im52o  im52o[8:1]
vector im53o  im53o[8:1]
vector im52d  im52d[2:1]
vector im53d  im53d[2:1]
vector im52g  im52g[3:1]      lg3g2g1
vector im53g  im53g[3:1]
vector va53   im26i[30:23]
vector c26    im26i[22:19]    lc3c2c1c0
vector dd26   im26i[18:3]
```

```
vector t26   im26i[2:1]
l    va53
h    smumu14              |data input set s3=1
assign im52g  000
assign im53g  100
assign c26   0000
assign dd26   0406\h
assign t26   00

|inst. 27
vector im27i  im27i[38:23]
vector im54o  im54o[8:1]
vector im55o  im55o[8:1]
vector im54d  im54d[2:1]
vector im55d  im55d[2:1]
vector im54g  im54g[3:1]      |g3g2g1
vector im55g  im55g[3:1]
vector va55   im27i[30:23]
vector c27    im27i[22:19]    |c3c2c1c0
vector dd27   im27i[18:3]
vector t27    im27i[2:1]
assign im54g  000
assign im55g  000
assign c27    0000
assign dd27   0cff\h
assign t27    00

|inst. 28
vector im56g  im56g[3:1]      |g3g2g1
vector im57g  im57g[3:1]
assign im56g  000
assign im57g  000

|inst. 29
vector im58g  im58g[3:1]      |g3g2g1
vector im59g  im59g[3:1]
assign im58g  000
assign im59g  000

|inst. 30
vector im60g  im60g[3:1]      |g3g2g1
vector im61g  im61g[3:1]
```

```
assign im60g  000
assign im61g  000

linst. 31
vector im62g  im62g[3:1]      |g3g2g1
vector im63g  im63g[3:1]
assign im62g  000
assign im63g  000

wave mdfp32–case2.wfm im4o9 im4o0 im6o9 im6o0 im12o9  im12o0 im13o9 im13o0
im16o9 im16o0 im18o9 im18o0  im20o9 im20o0 im22o9 im22o0 im32o9 im32o0 im34o9
im34o im34o0 im36o9 im36o0 im37o9 im37o0 im38o9 im38o0 im39o9 im39o0 im52o9
im52o im52o0 im54o9 im54o0 im55o9 im55o0
wfm reset @0ns=1 @100ns=0

assign im52o 01\h

sim 200ns

h  sopop3 sopop73 sopop17 sopop75   linitial token control set–up

l im4o
h im4o1
break im4o0 1 do(r im4o)

l im18o
h im18o1
break im18o0 1 do(r im18o)

l im32o
h im32o1
break im32o0 1 do(r im32o)

l im22o
h im22o1
break im22o0 1 do(r im22o)

assign sopop27 1
break im52o0 ? do(assign im52o < im52o.dat; assign sopop27 < sopop27.dat)

break im34o9 1 do(after 2.13ns do(h im34o0))
break im34o9 0 do(after 2.13ns do(l im34o0))

sim 11000ns
```