C-slow based Microprogrammed Finite State Machine for Parallel Thread Execution

By

Muhammad Adeel Akram
2007-NUST-MS PhD-ComE-03

Presented to the Faculty of Department of Computer Engineering at College of Electrical & Mechanical Engineering in Partial Fulfillment of the Requirements for the Degree of

Master of Science
In
Computer Engineering

Thesis Supervisor
Col. Dr. Shoab A. Khan

College of Electrical & Mechanical Engineering, Rawalpindi
National University of Sciences & Technology, Pakistan
August 2010

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

IN THE NAME OF ALLAH ALMIGHTY, THE MOST BENEFICIENT, THE MOST MERCIFUL, AND THE MOST COMPASSIONATE

DEDICATION

To my Parents, Teachers and Friends whose prayers, guidance and encouragement made me capable to do this work and to all those people who went off their way to help me to bring out the best in me and to make me what I am today.

Thank you

ACKNOWLEDGEMENT

In the name of *Allah Almighty*, Most Gracious, Most Merciful who has blessed me with the physical and mental capabilities to complete the assigned project.

I would like to thank my respected *Project Supervisor, Dr*. Shoab A. Khan, and the advisory committee members, whose special interest, kind guidance, encouragement, flexibility, and kind behavior throughout the project helped me a lot in completing the given task.

And my heartiest gratitude to my parents and siblings whose prayers made me pass through the ups and downs of the project. I would like to admit that I owe all my achievements in my truly, sincere and most loving grandparents, parents and whose prayers are a source of determination for me.

It would be incomplete without thanking some of my friends whose sincere help make me succeed in this way. Specially, I am thankful to Mr. Aqeel Ahmad, Mr. Umer Ali and Mr. Yasir Munir who helped me making out of this successfully.

MUHAMMAD ADEEL AHMAD

DECLARATION

I hereby declare and affirm that the thesis titled **“C-slow based Microprogrammed Finite state Machine for Parallel Thread Execution”** is neither whole nor as a part thereof has been copied out from any source (except data,). It is further declared that I developed this report entirely on the basis of my personal efforts made under the sincere guidance of my project supervisor and due to help of ALLAH Almighty.

If any part of this project proved to be copied or found be a part of some other, I shall stand by the consequences.

No portion of this work presented in this report has been submitted in support of any application for any degree and qualification of this or any other university or institute of learning. If found I will stand responsible.

MUHAMMAD ADEEL AKRAM

Abstract

To increase the throughput of system is the requirement of many designs. Normally for FPGA based designs pipelining is used to increase the throughput of the system. Pipelining increases the throughput of the system by introducing multiple inputs in a pipeline. Pipeline increases latency for single computation. But the total number of outputs per unit time increases. Pipelining has advantages for the designs which have only feed forward circuits. The designs where feedback loops also part of the design, pipelining cannot perform well. In those scenarios, c-slow is the better approach. C-slow increases the throughput of the system even in the presence of feedback loops. C-slow increases the throughput by taking the advantage of parallel computation i.e. multiple threads are executed in parallel. For this purpose each register in the design is replaced with C number of registers.

TABLE OF CONTENTS

Chapter 1

Introduction

Motivation

As with the increase in technology, the number of components increases in a great amount on the same area size. So the more and more hardware resources are available to program. Here the task is now how to program in such away so that on one we can utilize the available hardware and to increase the speed without adding any additional hardware. The basic idea behind this research thesis is to develop some technique that can perform the above defined goal in complex applications such as FPGA based designs as well as DSP applications. Another effect of this as when we purchase a hardware, we have already paid for all its components either we use it or not. So then it becomes out basic requirement to develop such techniques in which we can increase performance and speed with maximum utilization of available hardware.

Background

Because this thesis is based on the FPGA based systems and development of FPGA based system is involved a large number of complex transformations and optimizations. A simple FPGA contains of two dimensional arrays of logic cells and programmable switches. A logic cell can be programmed to perform a simple task while the programmable switch can be programmed in such a way that it can provide interconnection between logic cells. A FPGA based digital logic is designed so that the function of each logic cell is specified. In this thesis Spartan 3 family device is used. A basic Spartan 3 devices is given below,

Table 1.1 Devices in the Spartan-3 family

Device	**Number of LCs**	**Number of block RAMs**	**Block RAM bits**	**Number of multipliers**	**Number of DCMs**
XC3S50	1,728	4	72K	4	2
XC3S200	4,320	12	216K	12	4
XC3S400	8,064	16	288K	16	4
XC3S1000	17,280	24	432K	24	4
XC3S1500	29,952	32	576K	32	4
XC3S2000	46,080	40	720K	40	4

XC3S4000	62,208	96	1,728K	96	4
XC3S5000	74,880	104	1,872K	104	4

In above paragraph, a small introduction of FPGAs and the devices that used in this thesis is presented. Now below we discuss some of the FPGA design issues such as speed, area, power etc.

When we use some arbitrary coding methods for designing FPGA based digital systems, optimization tools cannot perform well in those cases because the requirements for those are not met. When the speed is major issue in designs, it is based on problem domain. There are three domains by which we can define speed, throughput, latency and timing. When we process data in a FPGA, throughput is the amount of data that is processed in a single clock cycle. And it is usually measured as bits per second. Latency is the amount of time between input data and processed output and latency is measured in clock cycles. And in the last, Timing is the delays in sequential elements when data pass through them. The basic task behind this thesis is to increase throughput and to reduce latency. Timing is not the major issue during the thesis.

High Throughput

When we are concerned about high throughput designs, i.e. designs in which data rate is constant and less worried about the latency i.e. the time consumed to process a single piece of data. The pipelining is the basic idea to increase the throughput. In pipelining, multiple stages are defined such as when first instruction is at second stage, second instruction enters at first stage and so on. In this case, it may possible that the time required for a single instruction increases but if we consider the number of outputs per unit time, it increases.

Low Latency

A low latency design means which can pass the input data to the output or process input data as quickly as possible by minimizing the intermediate processing delays. As above discussed high throughput design use the concept of pipelining but that increases latency in design. When latency is the major design issue, pipelining in the design is removed and requires some sort of parallelisms in designs and logical shortcuts that can reduce the throughput or maximize the clock speed. The bad thing in removing pipelining stages is the increase in timing or combinational delays.

Timing

Timing refers to the clock speed of a design. The maximum delay between any two sequential elements in a design will determine the max clock speed. The idea of clock speed exists on a lower level of abstraction than the speed/area trade-offs discussed elsewhere. The maximum speed, or maximum frequency, can be defined according to the straightforward and well-known maximum-frequency equation,

Maximum Frequency

$$F_{max} = \frac{1}{T_{clk} + T_{logic} + T_{setup} + T_{routing} - T_{skew}}$$

Where Fmax is the maximum frequency of clock; T_{clk} is the data arrival time; Tlogic is the propagation delay through logic between flip-flops; Tsetup is minimum time data must arrive at D before next rising edge of clock; Trouting is the routing delay between flip-flops; and Tskew is propagation delay of clock between the launch flip-flop and the capture flip-flop.

Area Minimization

Area minimization is based on the selection of correct design method. Design method is based on high level design organization and not on the device level. A design method that refers to the area, must reuses the logic resources to maximum extent possible, often at the expense of speed. Normally this requires the recursive data flow where the output of one stage is fed back to the input to the similar process. It may be implemented with simple loop that flows simply with algorithm or it can be implemented with the help of special controls.

Methodology

As we are developing technique for FPGAs so the programming environment here used is Verilog. For the test purpose of this technique, we developed a simple microprogrammed finite state machine design as well as some filters in which both feed forward and feed backward filters are included. In filters the developed technique is implemented for finite impulse response filters and infinite impulse response filters. The basic idea to increase the speed is to execute multiple programs with the same hardware as for the single program but with just adding that hardware that is available in excessive amount without any expenditure.

For the above defined purpose, I used C-slow retiming with microprogrammed finite state machine design. Microprogrammed state machine will be discussed in the next chapter. Basically the concept of microprogramming is much old, but here we combined the microprogrammed finite state machine with C-slow retiming. The concept of C-slow is to increase the number of registers C times. Now we can implement C different instruction sets that can work in parallel. The idea behind to choose the C-slow is that with the passage of time we have more and more registers are available on FPGAs. So if we increase the number of registers it will not much effect because these registers are available in the same cost either we use them or not.

After obtaining the simulation results to check the timing results, synthesize the code in Xilinx to get the hardware results. Then do the power analysis to verify our technique. And all these results are added in result analysis chapter.

Scope of thesis

The end result of this thesis is a process or technique which can perform the above defined functions. The developed technique is not only for a single program but it must be general purpose. It means we can use this technique in many physical complex applications.
These complex applications include Image processing applications, Signal processing applications, Communication and many other more fields where complex computations are involved. The applications where a lot number of multiplications or additions are used. So in these applications if we want to increase the speed of computation we also have to increase the multipliers and adders which take a lot of space as well as power which is not suitable for area critical as well as power critical designs. So my proposed technique is much more efficient in such cases because in these cases we just have to increase the number of registers and not any other hardware so helps to reduce the cost and area. At the end we will discuss the results by implementing the proposed applications on more than one design architectures and prove that the above claim about this technique is true.

Organization

The thesis is organized in the following sections to easily make the readers understand;

- Chapter 2 elaborates the characteristics in details about the finite state machines.

- Chapter 3 gives an overview of the Finite state Machine and Datapath

- Chapter 4 briefly describes the basic concepts of microprogramming

- Chapter 5 gives the review Microprogrammed state machine and the details of its implementation on a hard ware platform

- Chapter 6 give the concept of C-slow retiming and effects of C-slow retiming on different types of architectures and comparisons of C-slow pipelined and Unfolding architecture is addressed in this chapter

- Chapter 7 discusses the C-slow based microprogrammed FSM and its architecture to execute multiple threads in parallel and its implementation.

- Chapter 6 addresses the analytical simulation results of the C-slow based parallel computing processor architecture

Chapter 2

Finite State Machine

Introduction

The finite state machine (FSM) design is used in cases when a system transits among some finite number of states. And these transitions depend on the present state of the system and input from external source. Unlike the normal sequential circuit in which system transits between the states in a sequential manner i.e. first, second, etc. but in FSM, the state transition do not show a simple repetitive pattern. Its next state logic depends upon many conditions and also known as random logic.

In this chapter we will discuss in details about the finite state machines. The major application of finite state machine is to act as controller in a large number of digital systems in which FSM check the external commands and status and selects the proper control signals to control the operation of data path which is usually composed of regular sequential and combinational circuits.

FSM representation

There are two ways to represent a finite state machine, state diagram and ASM chart (Algorithmic State Machine chart). These two representations shows the same information. The FSM representation is more compact and better for simple applications. The ASM chart representation is somewhat like a flowchart and is better for applications with complex transition conditions and actions.

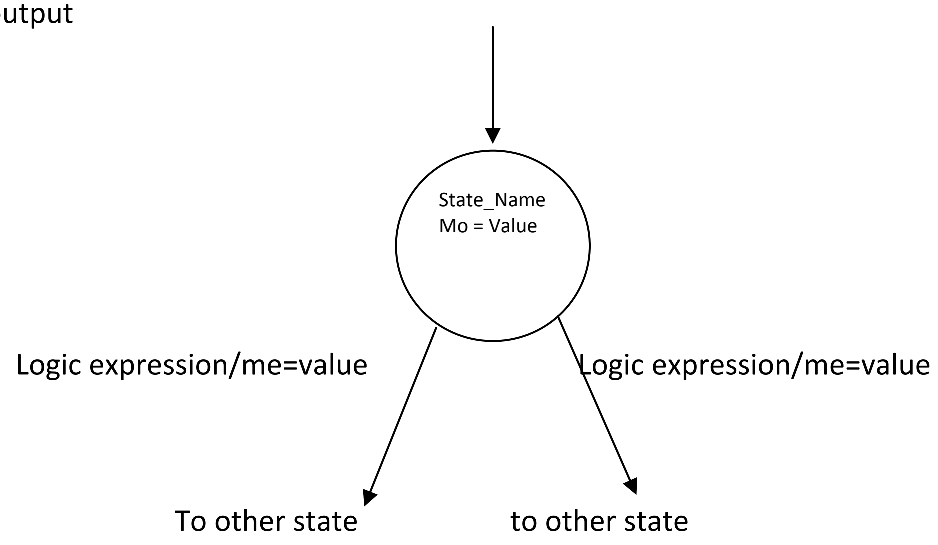
State Diagram

A state diagram is composed of nodes, which represent states and are drawn as circles, and annotated transitional arcs. A single node and its transition arcs are shown in Figure 2.1(a). A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition. The arc is taken when the corresponding expression is evaluated true.

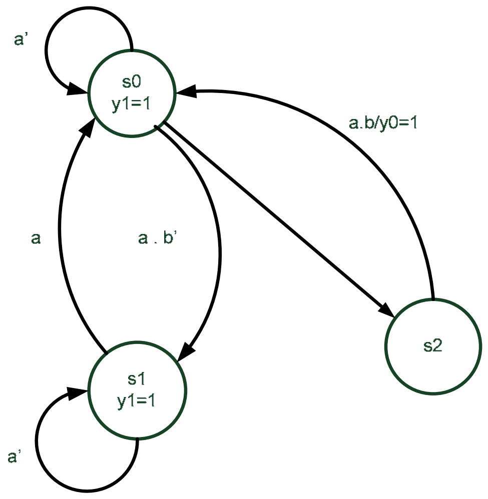
The Moore output values are placed inside the circle since they depend only on the current state. The Mealy output values are associated with the conditions of transition arcs since they depend on the current state and external input. To reduce clutter in the diagram, only asserted output values are listed. The output signal takes the default (i.e., unasserted) value otherwise.

A representative state diagram is shown in Figure 2.1(b). The FSM has three states, two external input signals (i.e., a and b), one Moore output signal (i.e., y1), and one Mealy output signal (i.e., y0). The yl signal is asserted when the FSM is in the SO or s l state. The yo signal is asserted when the FSM is in the SO state and a and b signals are " **1 1** ".

mo: moore output
me: mealy output



(a) A simple node representation



(b) State diagram using nodes

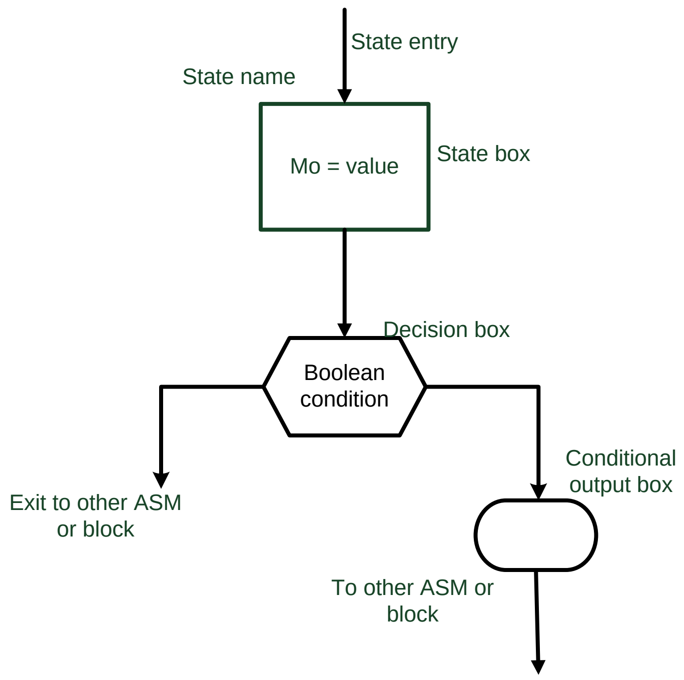
Figure 2.1: State diagram representation

ASM chart

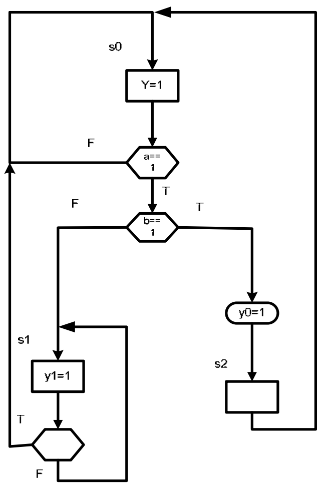
An ASM chart consists of a network of ASM blocks. An ASM block consists of one state box and a network of decision boxes and conditional output boxes. A simple representation of ASM block is shown in Figure 2.2(a).

A state box represents a state in an FSM, and the asserted Moore output values are listed inside the box. Note that it has only one exit path. A decision box tests the input condition and determines which exit path to take. It has two exit paths, labeled T and F, which correspond to the true and false values of the condition. A conditional output box lists asserted Mealy output values and is usually placed after a decision box. It indicates that the listed output signal can be activated only when the corresponding condition in the decision box is met.

A state diagram can easily be converted to an ASM chart, and vice versa. The corresponding ASM chart of the previous FSM state diagram is shown in Figure 2.2(b).



(a) ASM block



(b) ASM chart
Figure 2.2 ASM representation

Control Algorithm interpretation with FSM

The finite state machine (FSM) implementing control algorithm is represented by classical model of sequential circuit, which can be treated as the composition of combinational circuit and register.

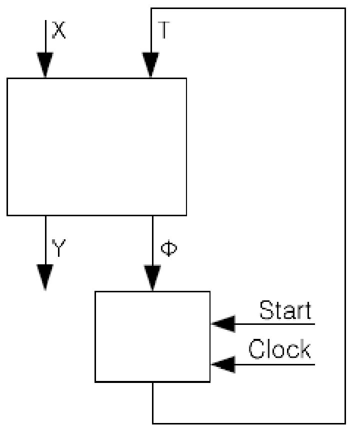


Figure 2.3 FSM structural diagram

Presence of register in this structure can be explained in the following way. A time-distributed microinstruction sequence *Y*(0)*,Y*(1)*, . . . ,Y*(*t*), where *t* is the time determined by synchronization pulse "Clock", appears on the output of FSM. The initial time *t* = 0 is determined by a single-shot pulse "Start". To produce such a sequence, some information about history of the system operation is needed. This sequence is determined by input signals $X(0), \ldots, X(t-1)$ for previous moments of time. It means that output signal $Y(t)$ at time t is determined by the following formula

$$Y(t) = f(X(0), \ldots, X(t-1), X(t))$$

Expressions of this kind are very cumbersome and could not be realized in hardware, especially if they contain cycles with unpredictable number of iterations. In practice, the history is described by special internal states of the FSM, from the set of states $A=\{a_1, \ldots, a_M\}$. States $am \in A$ are encoded by binary codes $K(am)$ having not less than

$$R = [\ \log_2 M\ ]$$

Elements of the set of state variables $T = \{T_1, \ldots, T_R\}$ are used to encode the states of FSM. The code of current state is kept in register RG, which includes R synchronous flip-flops with common timing signal "Clock". The code of initial state $a1 \in A$ is loaded into RG using pulse "Start". The content of RG can be changed by pulse "Clock" on the base of input memory functions, which form the set of input memory functions. As a rule, the register RG is implemented using D flip-flops. In case of the Mealy FSM, output functions Y are represented as

$$Y = Y(T, X)$$

In case of the Moore FSM, output functions depend only on the states variables:

$$Y = Y(T)$$

Synchronous FSM

Most FSM systems are synchronous; that is, they make use of a clock to move from one state to the next. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. Using a clock to control the synchronous movement between one state and the next allows the FSM logic time to settle before the next transition. The block diagram of a synchronous system is shown in Figure 4.2. It consists of the following parts:

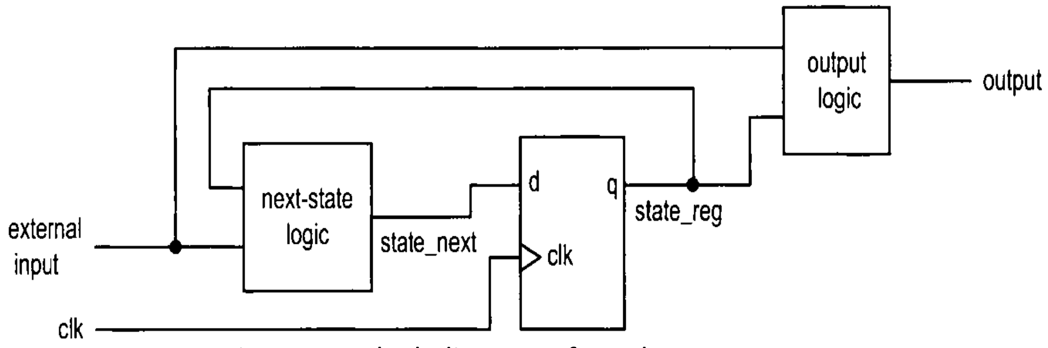


Figure 2.4 Block diagram of synchronous FSM

State register: a collection of D FFs controlled by the same clock signal
Next-state logic: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register.
Output logic: combinational logic that generates the output signal.

Asynchronous FSM

There is another kind of FSM, one that does not use a clock to instigate a transition between states. This is known as the asynchronous FSM. In an asynchronous FSM, the transition between states is controlled by the event inputs, so that the FSM does not need to wait for a clock signal input. For this reason, asynchronous FSM are sometimes called 'event-driven' FSMs. FSM will only change state when there is a change of input variable; hence, the event nature of the system. Sometimes, it is desired to change state when there is no input signal change (as has been seen in clocked driven systems). An important feature with event-driven FSM systems is that when the FSM is in a stable state (perhaps waiting for an input event to move to the next state) the power drain is very low in circuits, since there is no repetitive clock to consume power. This allows asynchronous (event) systems to be low power, while also being very fast. This latter point is due to the fact that the event FSM will move to the next state as soon as the relevant event input changes, and is only limited by the propagation delay for its event-driven logic.

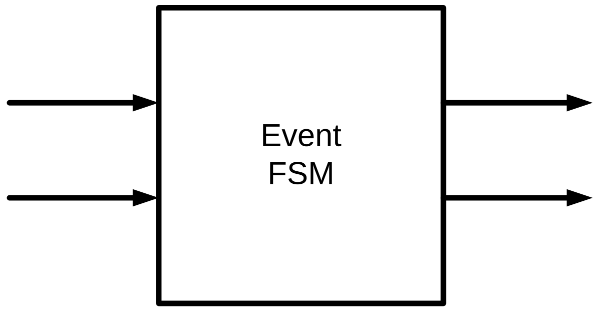


Fig 2.5 Block diagram of Asynchronous FSM

In a clocked FSM, the implementation (synthesis) will make use of some type of flip-flop; the event-driven system needs to make use of memory elements that do not require a clock input.

There can be a number of individual turn-on
Inputs and a number of individual turn-offs
Inputs to the cell

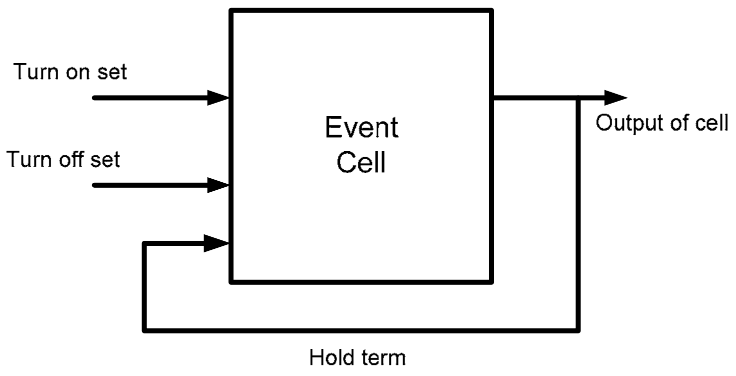


Fig 2.6 Basic Event (Asynchronous) Cell

Chapter 3

Finite State Machine with Datapath

Introduction

An FSMD (finite state machine with data path) combines an FSM and regular sequential circuits. The FSM, which is sometimes known as a *control path,* examines the external commands and status and generates control signals to specify operation of the regular sequential circuits, which are known collectively as a *data path***.** The FSMD is used to implement systems described by *RT (register transfer) methodology,* in which the operations are specified as data manipulation and transfer among a collection of registers.

Simple Register Transfer Operation

An RT operation specifies data manipulation and transfer for a single destination register. It is represented by the notation

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \cdots, r_{srcn})$$

Where r_{dest} is the destination register, r_{srcl}, r_{src2}, and r_{Srcn} are the source registers, and f (.) specifies the operation to be performed. The notation indicates that the contents of the source registers are fed to the f (.) function, which is realized by a combinational circuit, and the result is passed to the input of the destination register and stored in the destination register at the next rising edge of the clock. Following are several representative RT operations:

rl ← 0 A constant 0 is stored in the r1 register.
r1 ← r1 The content of the **rl** register is written back to itself
$r2 \leftarrow r2 \gg 3$ The r2 register is shifted right three positions and then written back to itself.
$r2 \leftarrow r1$ The content of the rl register is transferred to the r2 register.
$i \leftarrow i + 1$ The content of the i register is incremented by 1 and the result is written back to itself

A single RT operation can be implemented by constructing a combinational circuit for the f (.) function and connecting the input and output of the registers. For example, consider the $a \leftarrow a - b + 1$ operation. The f (.) function involves a subtractor and an incrementor. The block diagram is shown in Figure 3.l. For clarity, we use the reg and next suffixes to represent the input and output of a register. Note that an RT operation is synchronized by an embedded clock. The result from the f (.) function is not stored to the destination register until the next rising edge of the clock.

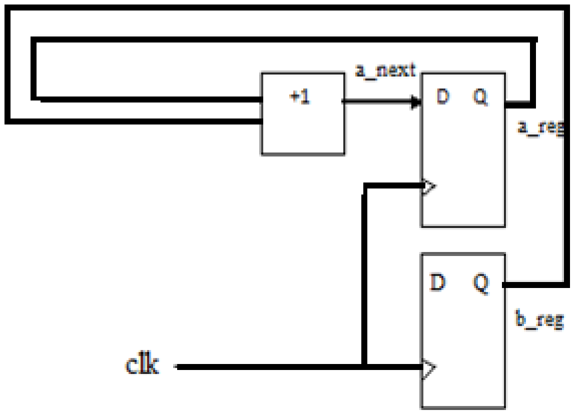


Fig 3.1 A single register transformation operation

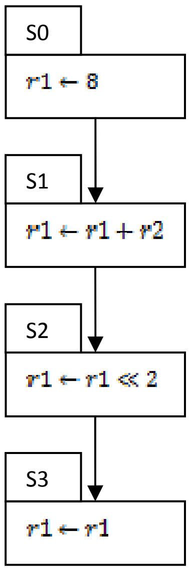
ASMD Chart

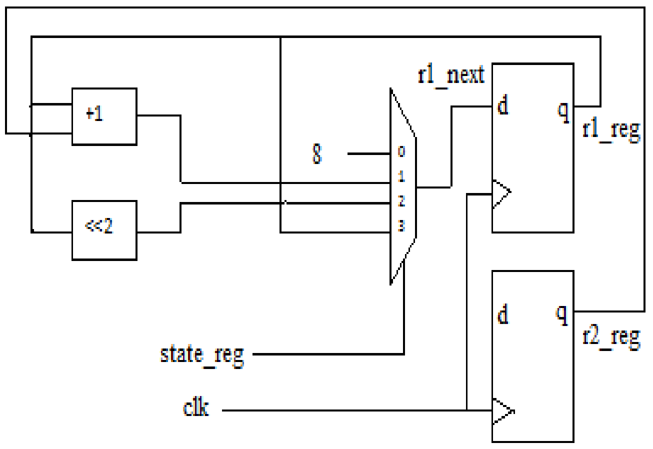
A design that based on the register transfer method defines which register transfer operation will execute at each step. As the register transfer operation is performed on clock by clock basis so its timing is similar to the state transition of an FSM. So the Finite State Machine becomes the first choice for sequencing in a register transfer operation. Thus we extend the Algorithmic State Machine chart to incorporate with register transfer operations and it is called by the name of ASMD chart (ASM with datapath).

A part of an ASMD chart is shown in Figure 3.2(a). It contains one destination register, rl, which is initialized with 8, added with content of the r2 register, and then shifted left two positions. Note that the rl register must be specified in each state. When rl is not changed, the rl← rl operation should be used to maintain its current content, as in the s3 state.

Implementing the RT operations of an ASMD chart involves a multiplexing circuit to route the desired next value to the destination register. For example, the previous segment can be implemented by a 4-to- 1 multiplexer, as shown in Figure 3.2(b). The current state (i.e., the output of the state register) of the FSM controls the selection signal of the multiplexer and thus chooses the result of the desired RT operation.

An RT operation can also be specified in a conditional output box, as the r2 register. Depending on a>b condition, the FSMD performs either r2←r2+a or r2 ← r2+b. Note that all operations are done in parallel inside an ASMD block. We need to realize a>b, r2+a, and r2+b operations and use a multiplexer to route the desired value to r2.





(a) ASMD segment (b) Block diagram

Figure 3.2 ASMD representation

Decision box with a register

The appearance of an ASMD chart is similar to that of a normal flowchart. The main difference is that the RT operation in an ASMD chart is controlled by an embedded clock signal and the destination register is updated when the FSMD exits the current ASMD block, but not within the block.

Consider the FSMD .The r register is decremented in the state box and used in the decision box. Since the r register is not updated until the FSMD exits the block, the old content of r is used for comparison in the decision box. If the new value of r is desired, we should use the output of the combinational logic (i.e., r_next) in the decision box (i.e., replace the r = = 0 expression with r_next==0.

FSMD block diagram

The conceptual block diagram of an FSMD is divided into a data path and a control path, as shown in Figure 6.5. The data path performs the required RT operations. It consists of:

Data registers: store the intermediate computation results
Functional units: perform the functions specified by the RT operations
Routing network: routes data between the storage registers and the functional units

The data path follows the control signal to perform the desired RT operations and generates the internal status signal.

The control path is an FSM. As a regular FSM, it contains a state register, next-state logic, and output logic. It uses the external command signal and the data path's status signal as the input and generates the control signal to control the data path operation. The FSM also generates the external status signal to indicate the status of the FSMD operation.

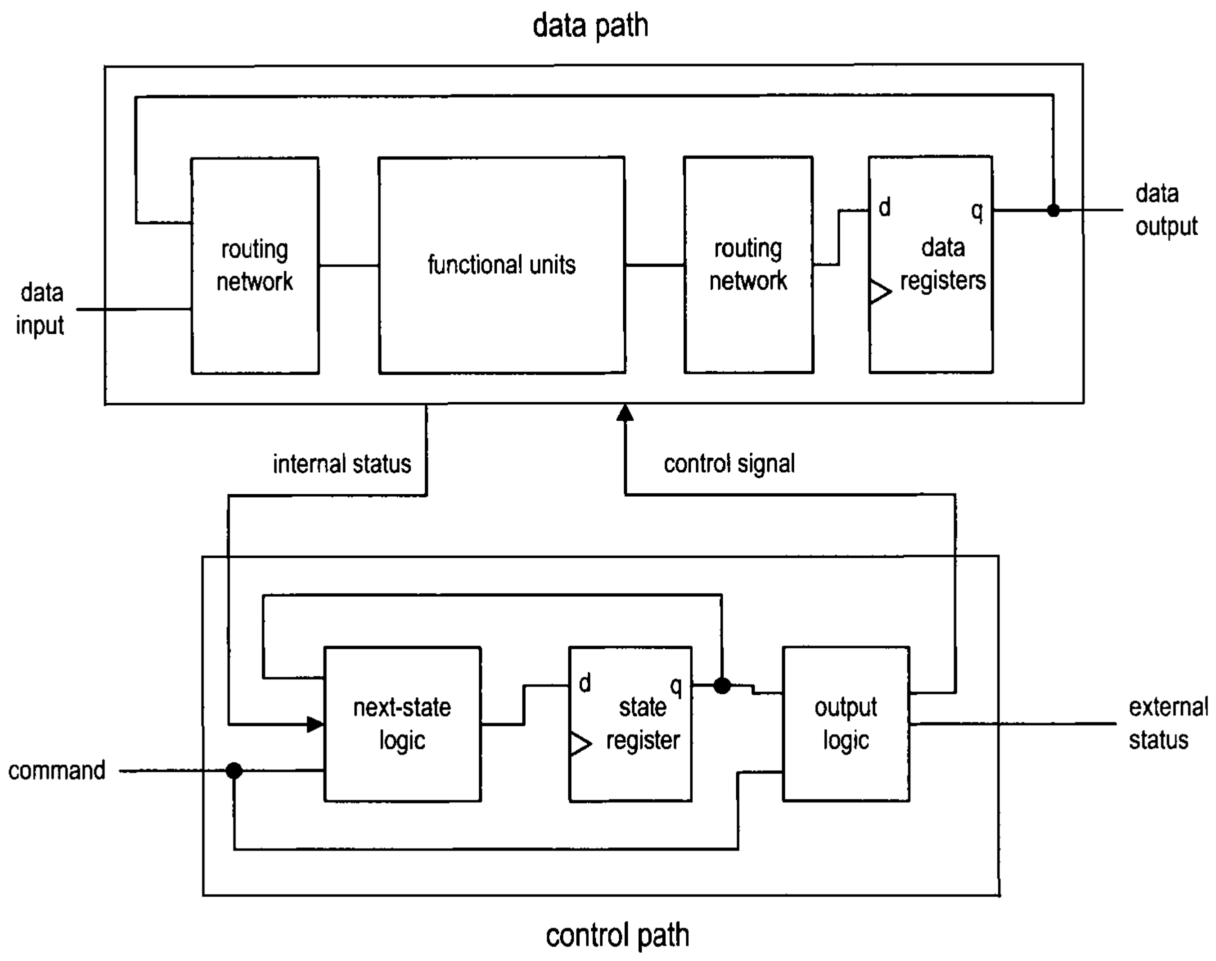


Figure 3.3 Block diagram of FSMD

Although an FSMD consists of two types of sequential circuits, both circuits are controlled by the same clock, and thus the FSMD is still a synchronous system.

Chapter 4

Basic Concepts of Microprogramming

Introduction

This chapter introduces the basic concepts of microprogramming and it includes the principles of micro program controls and behavior of micro programmed control units. Microprogramming is a technique to implement the control logic necessary to execute instructions within a processor. It relies on fetching low-level microinstructions from a control store and deriving the appropriate control signals as well as micro program sequencing information from each microinstruction.

History of Microprogramming

A microinstruction is a string of bits in ROM (read-only memory) and when executed, it generates many enable signals to control the timing of executing a target instruction.

Each bit in a microinstruction provides a single control function. Due to the fact that less hardwired logic is required to execute microinstructions, it is easier to debug the logic. A microinstruction means microcode, and it is occasionally just abbreviated as instruction.

The concept of microprogramming was first introduced by prof. M. Wilkes in early 1950 in his famous work [1]**.** In 1950 microprogramming gain some attention but it was not commercially became popular until IBM first implemented this technique [2] and [3]. The reason of this long delay is due to the speed difference between memory where microinstructions were to be stored and logic circuits which had to control the microinstructions.

The memory technology that was first time developed for microprogranning were capacitive, transformer ROMs discussed in [4].

In early 1970s and 60s, there was a great improvement in semiconductor, there is great increase of microprogramming for commercial use [5**].** This also introduced a new method of dynamic microprogramming in which microcode could not only read but also new microinstructions could be introduced. This is described in [7].

Principle of Microprogramming

The principle of microprogramming was developed by V. Glushkov. According to this principle, any complex operation executed by digital system can be divided into a sequence of small processes. These small processes are named as micro instructions. Special logic is used to control the sequence of execution. An executable microprogram consists of microinstructions and logical conditions.

A control algorithm can be implemented as a program or as a network of logic elements. The principle developed by Glushkov can be explained as follows;

1. Any operation, executed by a digital device, is considered as a complex action, which is represented as the sequence of elementary operations (microoperations) on the words of information (operands).
2. The logical conditions (status signals) are used to control the order of microoperation executions. The values of logical conditions are calculated as some Boolean functions depending on the values of operands.
3. Execution of operations in a digital device is specified by a control algorithm, which is represented in terms of logical conditions and microoperations. It is called microprogram". Microprograms determine an order of testing the values of logical conditions and sequences of microoperations, which are necessary to get proper operation of the device.
4. The microprogram is used as a particular form of specification of the function of a device and determines the structure of digital device and the order of its time operation sequence.

Firmware

While microprograms contain information that control hardware at a primitive level, they are stored in a memory and are executed as stored programs. This gives microprogramming a software as well as a hardware flavor so the appellation "firmware," "a term to designate microprograms resident in the computer's control memory" [8], is most appropriate. This software nature means that microprogramming may be considered independently from machine languages; indeed, the brief description in the previous paragraph of the operation of a microprogrammed computer makes no reference to machine language instructions.

Flynn [9] and some other provide that both software and hardware designers work on algorithms but the realization is different for both of them. If the memory is writeable we refer its contents as software. And if read only we can think it as hardware. In both cases the term “firmware” can describe its functional utilization.

Characteristics of Microprogrammable Architectures

In his paper [6] described some characteristics of microprogrammable architectures. According to that a microprogrammable machine can be characterized on the basis of control store design, microinstruction design, microinstruction implementation and microprogrammability.

Control Store Design

The control store of a microprogrammed computer may be logically organized in a variety of ways :

1. The simplest and most common control store structure is the ordinary memory array in which there is one microinstruction in each control store word. There are several variations of this structure.
2. In one form the number of bits in each control store word is increased so that two microinstructions occupy each control store word. The advantage of this scheme is that two microinstructions are read into the microinstruction register(MIR) simultaneously. This reduces the number of control store references.
3. In another structure, the control store is divided into blocks. In this scheme there are two types of control store addresses-addresses of control store words in the same block as the current microinstruction and addresses of other blocks. As a result of this organization, addresses of other microinstructions in the same block are shorter than in a no blocked structure.
4. In the split control store structure, control store comprises two distinct storage units which have different word sizes. The storage unit with the shorter word size contains microinstructions which move literal data contained in the microinstruction to one or more machine registers, or initiate the execution of a microinstruction which resides in the other storage unit. The second storage unit has many more bits per word and hence can exercise more control over machine resources.

Microinstruction design

Of primary interest in the design of microinstructions is the number of resources each microinstruction controls. In this respect, microinstructions can be classified into two categories; horizontal microinstruction format, vertical microinstruction format.

Vertical microinstructions effect single operations-load, add, store, branch, etc. They often resemble machine language instructions containing one operation code and two or more operands. Lengths of vertical microinstructions typically range from 12 to 24 bits.

Horizontal microinstructions, in contrast, control many resources which operate in parallel. A single horizontal microinstruction might control the simultaneous and independent operation of one or more ALU's, input from and output to main memory, conditional next address generation, etc.

While horizontal microprogramming has the potential advantage of efficient hardware utilization, developing horizontal microprograms that use resources optimally is a difficult problem. Because horizontal microinstructions control multiple resources, they contain more information than vertical microinstructions and hence have greater length; horizontal microinstructions with 64 bits and more are common. The determining characteristic between vertical and horizontal microinstructions, then, is the number of simultaneously controlled resources.

Microinstruction Implementation

While microinstructions are performed in a general fetch-decode-execute sequence, details of implementations vary greatly. Unlike the- implementation of machine language instructions, the effects of microinstruction implementation are usually not hidden from the microprogrammer. The serial-parallel characteristic of microinstruction implementation measures the amount of overlap between the execution of the current microinstruction and the fetching of the next microinstruction to be executed. In a serial implementation fetching the next microinstruction to be executed does not begin until the execution of the current microinstruction terminates. At the other extreme, the fetch of the next microinstruction to be executed is performed in parallel with the execution of the current microinstruction. The advantage of serial or sequential fetch is simplicity of realization; the hardware need not control execution and fetch simultaneously, and no problems arise in executing conditional branch instructions. The advantage of parallel fetch is the corresponding saving of time; the execution of a microinstruction begins immediately after the completion of the previous microinstruction.

In a mono phase implementation, there are no distinct sub cycles of the basic clock cycle; each microinstruction is affected by a single simultaneous issue of control signals. In a poly phase implementation, each major clock cycle comprises multiple minor clock cycles; the hardware generates control signals at each minor clock cycle. The advantage of mono phase operation is simplicity of realization when race conditions are not present. Poly phase operations allow interaction among resources at the expense of more complicated control.

Micrprogrammabilty

The degree of difficulty in microprogramming a machine once the design and implementation of its microinstructions are understood characterizes the microprogrammability of the machine. The ease with which a machine can be microprogrammed depends on the-realization of control store and the availability of support software to assist in microprogram preparation. The microprogrammability characteristic thus deals with both hardware and software aspects of microprogramming. The microprogrammability spectrum ranges from non microprogrammed (i.e., hardwired) computers to dynamically user microprogrammable machines.

User Microprogramming

User microprogramming considered as a tool for the user in which user can actively change the microinstructions according to its needs. Microprogramming as tool of the user has slowly evolved. Minicomputer manufacturers were probably the first to actively support user participation in the design process by making available facilities to assist in the design and checkout of ROM programs and by producing customized ROM's to meet special needs, particularly in the process control environment. Cost, lead time and relatively difficulty of microprogramming combined to limit this mode of user participation. Despite of cost, there is one other thing due to which most of users do not appreciate user defined microprogramming because most of the users are unaware of the languages and tools required to reprogram the control memory. And also user wants system at less cost, so manufacturers prefer to create fixed microprogram.

Software support available to the user for microprogramming varies greatly. In several machines, users are not given facilities for putting microprograms into control store, even though it may be writable. These machines are considered microprogrammed but not user microprogrammable. Support for microprogrammable computers includes preparing microprograms according to user design specifications and providing assistance to users in creating their own microprograms. This assistance is manifested in the form of support software [6].

Dynamic Microprogramming

Dynamic microprogramming gives us the concept of microprogram memory as not only readable but also writeable. Dynamic microprogrammed machines have been given the capability of swapping of microprogram in and out from microprogram control memory. Thus dynamic microprogramming has great advantages. Sometimes user's need changes to some extent. If the capability of dynamic microprogramming is available, then it is possible to redesign the architecture just by changing the microprogram in the control memory in order to meet the needs of immediate job to be done.
In microprogrammed machines, control store realizations vary from inflexible read only memories (ROM's) through memories with fast read/write capabilities, which allow the microprogrammer to add new microinstructions to control store as the microprogram is

interpreted. Machines in which the control store can be loaded under program control are called dynamically microprogrammable [7].

Writeable Control Store

The introduction of writeable control store, control memory with the capability of read as well write, has given rise to what can truly be described as micro programmable computers, in which not only the designer but the system programmer and even applications programmer can utilize the microprogramming capabilities to assist in problem solving. The first micro programmable machines included ROM for the bulk of the micro program and a smaller amount of write control store to enable the addition of new, special instructions to extend the basic repertoire or perhaps permit the microprogramming of the special routines to speed up highly repetitive and time consuming tasks. Even in these machines however, microprogramming features primarily reflect designer’s intention of implementing a specified target machine. It is easy to write a micro program but it may be difficult to write micro programs to accomplish different tasks from the same machine.
Recently, machines have begun to appear that are fully micro programmable and whose architecture are intended to support general purpose microprogramming. With the equal level of difficulty, they can be micro programmed to accomplish a variety of tasks.

Microprocessors and Microprogramming

The terms of microprocessor and microprogramming are usually somewhat confusing. A microprocessor is generally an integrated circuit on a single chip, and with the addition of clock circuit and other mathematical and logical functions it becomes a microcomputer. it is commonly thought that microprocessors are inherently microprogrammed, i.e., that programming a microprocessor is microprogramming. This notion is not correct; as with minicomputers and large computers, some microprocessors are microprogrammed and some are not. The prefix "micro" in the term "microprocessor," as it is generally used today, means small [6].

Applications of Microprogramming

In [5] some applications of microprogramming are explained. With the development of user microprogrammable minicomputers and microprogrammable bit slice microprocessors, practitioners have facilities to explore new applications. Interpretation of machine' language instructions continues to be the most common application of microprogramming. Here we discuss developments in microprogramming applications and research areas during the last few years.

Emulation

According to the definition of Rosin,"… we use the term "emulator" to describe a complete set of microprograms which, when embedded in a control store, define a machine. We shall call a machine which is realized by an emulator a "virtual machine" and the machine which supports microprograms a "host machine"." By appropriately defining virtual machines, many microprogramming applications may be considered to be instances or examples of emulation.

In emulation of microprocessors and minicomputers, there have been several reports in the literature, [10], [11], [13] because the manufacturers test their machine's working before they actually implement the design.

Operating System

Recent work in the area of microprogramming use in operating systems has followed two approaches:

1. implementing primitives that will be used throughout an operating system in microprograms and
2. Implementing important portions of operating systems partly or entirely in microprograms.

The primitives that are candidates for microprogram implementation are those which are too specific for hardware implementation, unlikely to change with time, and which would suffer from the slower execution rate of software. Brown et al. [12] list fourteen functions that could be implemented as primitives through microprogramming. These include

1. Program synchronizing primitives,
2. Queue manipulation mechanisms, and
3. State switching.

The advantages of this first approach result in operating systems with

1. Increased performance,
2. Decreased program development cost, and
3. Improved security and correctness.

A subsequent report [14] describes how to select the primitives most appropriate for implementation and how to perform an analysis of the tradeoffs between software, firmware, and hardware.

Use of Microprogramming in Architecture Implementation

As logic and memory technology improve, microprogramming is being used in innovative ways to implement many aspects of computers' architectures. Machines have been designed whose microprogram level architecture facilitates implementation of higher level languages. Kogge has discussed the advantages of using microprogramming to implement various pipeline structures [14]. Microprogrammable microprocessors are being used to implement minicomputers and emulate monolithic microprocessors [15], [16], [17] to implement special floating point processors [6], and to implement test equipment [18]. Networks of microprogrammable microprocessors are being used to implement advanced features such as pipelining, multiprocessing, and distributed processing [19] microprogrammable minicomputers are, being used as specialized reconfigurable nodes in networks [20]. In addition microprogramming continues to be used in implementing fault tolerant systems [21] and measurement systems.

Chapter 5

Microprogrammed Finite State Machine

Introduction

The ratio of internal circuit clock frequency and input sample frequency has a great effect on the design. In signal processing applications circuit frequency is much greater than sampling frequency. In this case if we directly map the design, it will not fully utilize the hardware resources. A logical design decision should be to use only required number of hardware computational resources and share them for multiple computations of the algorithm in different clock cycles. The architecture where one functional block is reuse in time to execute different computations of the algorithm is termed as time-shared or folded architecture.

Any synchronous digital design sharing the HW building blocks for different computation in different cycles will require a controller. The controller directs different computations of the algorithm to use the resource in different clock cycles. In time-shared architecture the digital system is divided in two parts: data path and control unit. The data path is the computational engine and consists of registers, multiplexers, de-multiplexers, ALUs, multipliers, shifters, combinational circuits and buses. In time-shared architecture different computational nodes in the data flow graph share these computation units in the data path. This sharing requires a controller directing data path to perform multiple operations on single physical unit in non-overlapping cycles. The controller selects the operations and determines the sequence in which these operations are to be performed. In many applications this sequence of operations may also depend on status of some computation in the data path and the status signals are fed back to the control unit. For hardwired state machine based designs, the controller is implemented as Mealy or Moore Finite State Machine (FSM).

In many applications the algorithms are so complex that either a hardwired state machine based designed is not feasible or is too complicated to design. In other applications several algorithms may need to be implemented for different scenarios. In these applications a flexible state machine based controller is needed which can be reprogrammed easily if required. This chapter describes designs of Micro programmed State Machines with different capabilities and options. The chapter generates motivation by listing equivalent microprogrammed state machine based implementations.

The analysis of few algorithms shows that in many cases a simple counter based Micro-program state machine can be used to implement a controller. The memory contains all the control signals to be generated in a sequence whereas the counter keeps computing the address of the next micro-code stored in the memory. In applications where decision based execution operations is required, this elementary counter-based state-machine is augmented with decision support capability. The execution is based on few condition lines coming either from the data-path or as input, if condition is true the state-machine start executing sequence of instruction stored at address specified in the conditional instruction. Further to this the counter in the state-machine can be replaced by a Program Counter (PC) register, which in normal execution adds a fixed offset to its content to get the address of the next instruction. The chapter then lists cases where a program instead of executing the next micro-code needs to take a jump to execute sequence of instructions stored at different location in the program memory. This requires subroutine support in the state-machine. After executing the subroutine, the state-machine returns to the address of the next instruction from where the subroutine was called.

Microprogrammed Controller

A microprogrammed control unit consists of binary control values stored as a word in memory. Each word in the control contains a microinstruction that performs one or more microoperations for a system. And this sequence of microinstruction is called by the name of microprogram. The contents of microinstruction, stored in memory at a given address, shows the microoperations performed on both datapath and control unit. Memory where the microinstructions are stored is called control memory. If the memory is RAM, then it is called writeable control memory.

Normally, the microprogram is loaded up at the start up of the computer from the non volatile memory to the computer memory. The block diagram of microprogrammed control unit is shown in the fig. 4.1.

The control memory is assumed to be a ROM within which all control information is permanently stored. The control address register specifies the address of the microinstruction. The control data register, which is optional, may hold the microinstruction currently being executed by the datapath and the control unit. One of the functions of the control word is to determine the address of the next microinstruction to be executed. This microinstruction may be the next one in sequence, or it may be located somewhere else in the control memory. Therefore, one or more bits that specify how to determine the address of the next microinstruction must be present in the current microinstruction. The next address may also be a function of status and external control inputs. While a microinstruction is being executed, the next-address generator produces the next address. This address is transferred to the control address register on the next clock pulse and is used to read the next microinstruction to be executed from ROM. Thus, the microinstructions contain bits for activating microoperations in the datapath and bits that specify the sequence of microinstructions executed.

The next-address generator, in combination with the *CAR*, is sometimes called a microprogram sequencer, as it determines the sequence of instructions that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one and loading the control address register. Possible sources for the load operation include an address from control memory, an externally provided address, and an initial address to start control unit operation.

The control data register holds the present microinstruction while the next address is being computed and the next microinstruction is being read from memory. The control data register breaks up a long combinational delay path through the control memory and the datapath. The ROM operates as a combinational circuit, with the address as the input and the corresponding microinstruction as the output. The contents of the specified word in ROM remain on the output lines of the ROM as long as the address value is applied to the inputs. No read/write signal is needed, as it is with RAM. Each clock pulse executes the microoperations specified by the microinstruction and also transfers a new address to the control address register, which, in this case, is the only component in the control that receives clock pulses and stores state information. The next-address generator and the control memory are combinational circuits. Thus, the state of the control unit is given by the contents of the control address register.

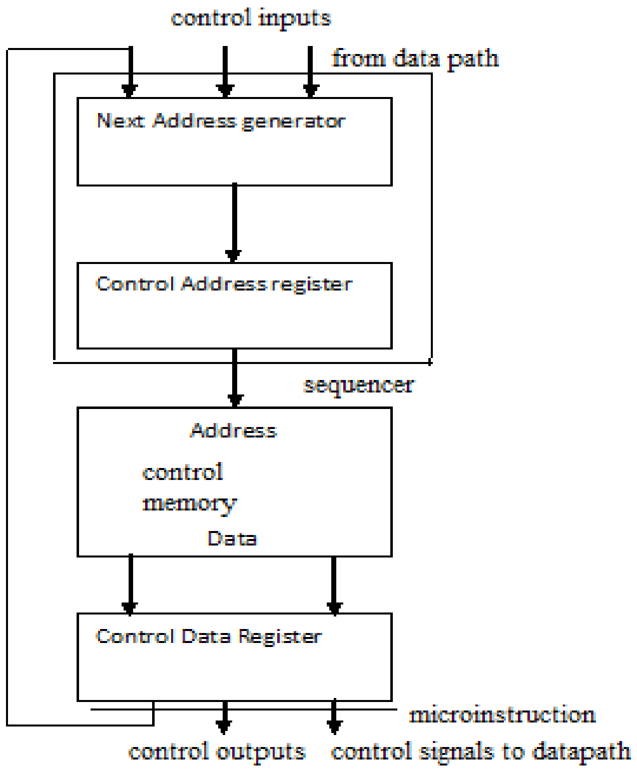


Figure 5.1 Block diagram of microprogrammed control unit

Counter based Microprogrammed FSM Implementation

In many controller designs the FSM generates a sequence of control signals without any input. In these design the Micro-program Memory is sequentially filled with the control signals. The state machine only needs to generate address in a sequence starting from 0 and ending at the last control signal in the sequence. The address can be easily generated with a counter. Thus the counter acts as a state register and generates the address and automatically gets incremented to generate next state for the state register as shown in the fig 5.1.

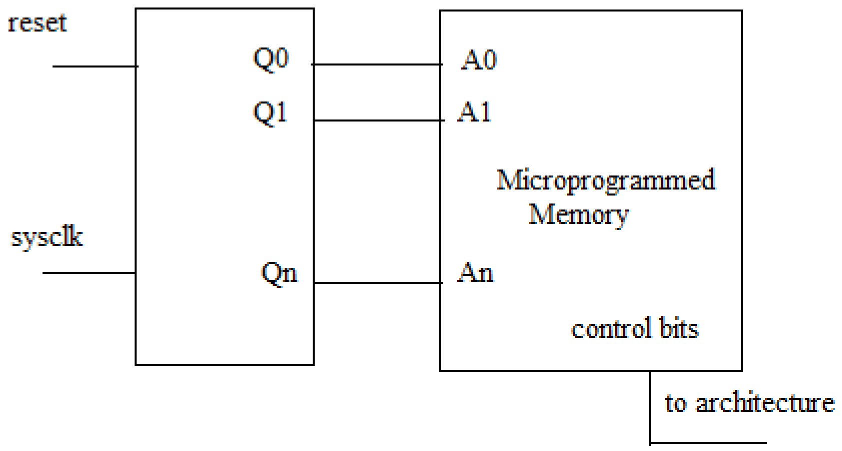


Figure 5.2 Block diagram of microprogrammed control unit

The counter is a complex structure that includes the state variable flip-flops and input combinational network. The output combinational network is implemented with a memory as shown in the figure. The counter is made to count a binary sequence beginning with zero i.e. reset. All data outputs from the memory are available for use in engineering control signals. This machine, like the first one, is of use only generating a single sequence of control signals even through the actual values of the signals as a function of time are completely flexible. Once the memory is filled with desired values, then its outputs will cycle endlessly until reset.

There is another type of counter based FSM toward a generally useful design by introducing a mechanism whereby a count sequence can be changed to begin another sequence under control of the micro program memory. Most of the bits in the micro program are used to generate control signals for some architecture which is not shown.

Counter Based Microprogrammed FSM with Branching

Unconditional Branching

A simple Counter-based Micro-program state machine can only generate control signals in a sequence. Many algorithms once mapped on time-shared architecture may also require some decision making capabilities where the controller while generating a sequence of control signals is capable to jump and start generating control signals from some other location in the Micro-program memory. This flexibility in the design is added by incorporating the address to be branch as part of the instruction and a load-able counter to load this value when load signal is asserted. This branching is termed as unconditional branch provision in the architecture.

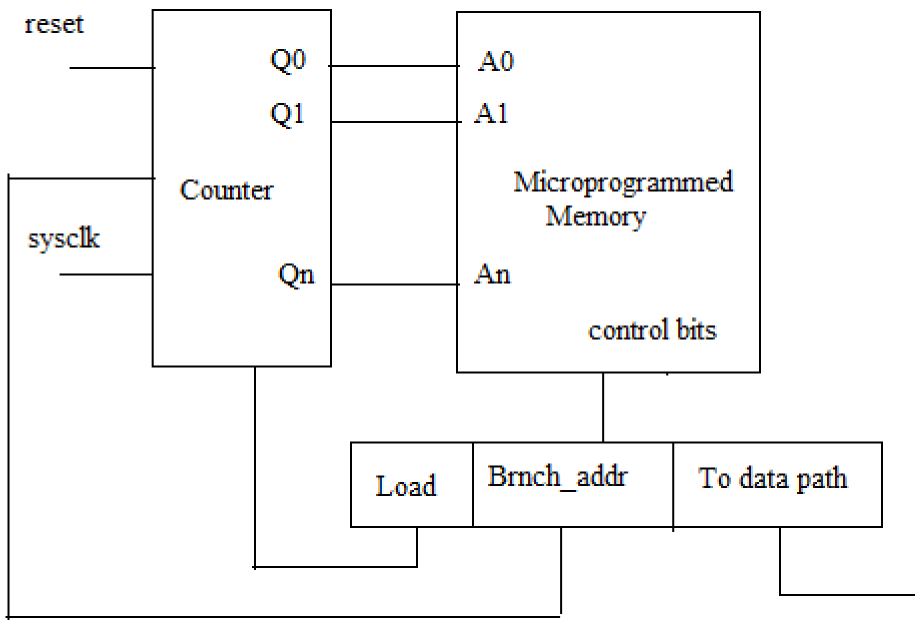


Figure 5.3 Unconditional branching

Conditional Branching

In many applications it is desirable to add conditional branching support in the Micro program state machine. The state machine implements control signals in a sequence and the machine is also capable of transitioning to a new address in PM subject to status of conditional inputs. The control inputs usually come from the data path Status and Control Register (SCR). On the basis of execution of some micro code, the ALU in the data path sets corresponding bits of the SCR. Example of control bits are zero and positive status bits in SCR. These bits are set if the result of previous ALU instruction is zero or positive. These two status bits can allow selection of two conditional branching. In this case the state machine will check if the input control signal from the data path is true of false. The controller will load the branch address in the counter if the conditional input is true, otherwise the controller will keep generating sequential control signals from PM as shown in the fig 5.3.

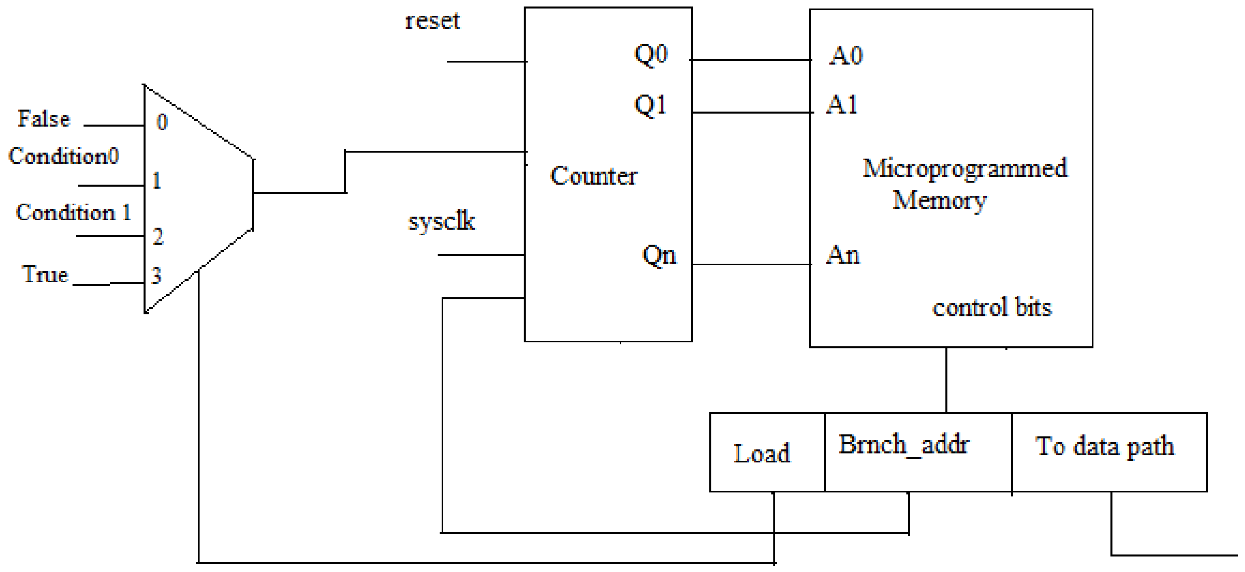


Figure 5.4 Conditional Branching

Register Based Microprogrammed FSM

The counter in counter-based micro programmed machines, is divided into two parts i.e. an incrementer and register. The microprograms counter register. Instead of taking an address into the micro program memory directly from the state variable outputs as we did in the counter based controller, we will impose a multiplexer, MUX, between the state variables and micro program memory as shown in fig 5.4.

The path from the branch address field to the micro program memory via the parallel load input of the counter is changed so that it passes through the MUX directly to the memory address inputs. This removes one level of clocked element from the path, hence the one clock delay that we experienced in our counter based implementation. The path from the micro programmed counter register outputs through the MUX to the PC register inputs via the incrementer modifies the flow along the path in a parallel loadable counter. In the counter, the parallel data path goes to directly to the register depending upon the load input. In this new design, the value applied to the address inputs of the micro programmed memory will always be incremented and placed in micro programmed register at the end of the current SYSCLK cycle no matter what the source address is.

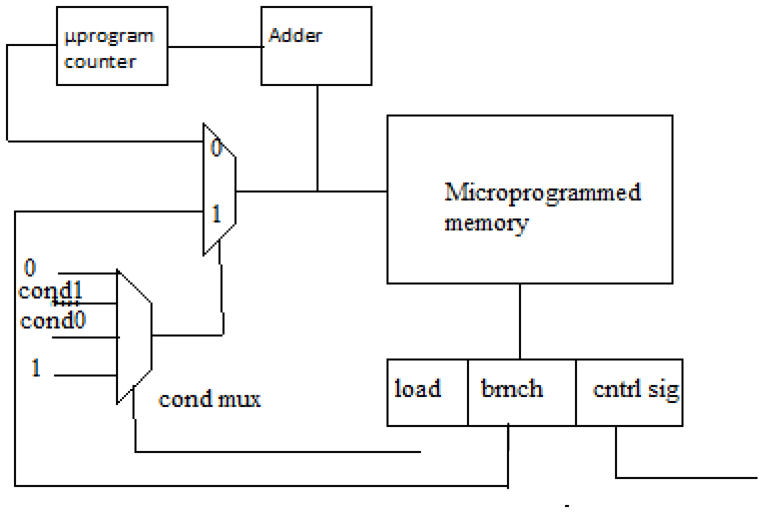


Figure 5.5 Register based microprogrammed controller

Microprogrammed FSM with subroutine support

In many applications it is desired to repeat a set of microcode instructions in generating sequence of control signals to the data path. Adding subroutine capability in the Microprogram State Machine optimizes these types of design. Those microcode instructions that are repeating can be placed in a subroutine and call to this subroutine is made from the required locations in the microcode. The state machine after executing the sequence of operations in the subroutine needs to return to the next microcode from where the call to the subroutine is made. . This requires a register storing the contents of micro PC at instance when the state machine decodes the call to subroutine microcode as in this cycle the micro PC contains the address of the next microcode in the sequence. While executing the return microcode, the next address select logic directs the next address MUX to select the address store in SRA register. Though the micro PC has the address of the next instruction but return address gets to the address bus and the microcode at return address is read for execution. In the next clock cycle the micro PC stores an incremented value of the next address thus keeps executing sequence of micro-codes from there on. All this is shown in the fig below.

Chapter 6

C-Slow Retiming

Introduction

This chapter is the most important because to understand the thesis, the required concepts will be discussed in this chapter. In first we will discuss some basic concepts in which pipelining, retiming, folding, unfolding, timeshared vs hardwired and feedback vs feed forward systems. Then we will discuss the concept of c-slow retiming and why we need c-slow retiming in the present of such algorithms. In the last, some examples of c-slow and their results are discussed.

Pipelining is a method that is used to increase the throughput by adding register stages between logic groups of design. A well-designed module can usually be pipelined by adding additional register stages and only cost is total latency with a small amount in area.

Leiserson et al. [22] were the first to propose retiming, an process to reposition pipeline stages to balance a design. Their algorithm can rebalance a design so that the critical path is optimally reduced. In addition, two modifications, repipelining and C-slow retiming, can add additional pipeline stages to a design to further improve the critical path.

The key idea is simple: If the number of registers in every clock cycle in the design does not change, the end-to-end design does not change. Thus retiming tries to solve two primary objectives: All paths longer than the desired critical path are registered, and the number of registers around every cycle is unchanged.

This optimization is useful for conventional FPGAs but absolutely essential for fixed-frequency FPGA architectures, which are devices that contain large numbers of registers and are designed to operate at a fixed, but very high, frequency, often by pipelining the interconnect as well as the computation.

To meet the fixed frequency, a design must ensure that every path is properly registered. Repipelining or *C*-slow retiming enables a design to be transformed to meet this objective. Without repipelining or *C*-slow retiming, the designer must manually ensure that all pipeline objectives are met by the design.

Retiming operates by determining an optimal placement for existing registers; while repipelining and *C*-slowing add registers before the beginning of retiming. After retiming, the design should balanced, with no pipeline stage requiring significantly more time than any other stage.

Background

In many applications, there is a requirement of digital data processing. These applications include mobile radio, satellite communications, speech processing, video and image processing, biomedical applications. Real-time implementations of these systems require design of hardware that can match the application sample rate to the hardware processing rate (which is related to the clock rate and the implementation style). Thus, real-time does not always mean high speed. Real-time designs are capable of processing samples as they have received from the signal source, without storing them in buffers for later processing. Furthermore, real-time designs operate on an infinite time series (since the number of the samples of the signal source is so large that it can be considered infinite). While speech and sonar applications require lower sample rates, radar and video image processing applications require much higher sample rates. The architecture cannot be choosing on the basis of sample rate. The algorithm complexity is also an important factor. For example, a very complex and computationally expensive algorithm for a low-sample-rate application and a computationally simple algorithm for a high-sample-rate application may require similar hardware speed and complexity. These ranges of algorithms and applications motivate us to study a wide variety of architecture styles.

Usually programmable digital signal processors can be prototyped much fast. These prototyped systems can prove very effective in fast simulation of computation-expensive algorithms (such as those encountered in speech recognition, video compression, and signal processing). After standards are determined, it is more useful to implement the algorithms using dedicated hardware.

Designing of dedicated circuits is not a simple task. Dedicated circuits have limited or no programming flexibility. Their requirement is less area and consumes less power. However, due to the low production volume, high design cost, and long time are some of the difficulties associated with the design of dedicated systems.

Successful designing of dedicated circuits is required to choose careful algorithm and architecture. Thus, some of these may be suitable for a particular application while other may not be able to meet the sample rate requirements of the application. The lower-level architecture can be implemented in a word-serial or word-parallel method. The arithmetic functional units can be implemented in bit-serial or digit-serial or bit-parallel method. The synthesized architecture may be implemented with a dedicated data path or shared data path.

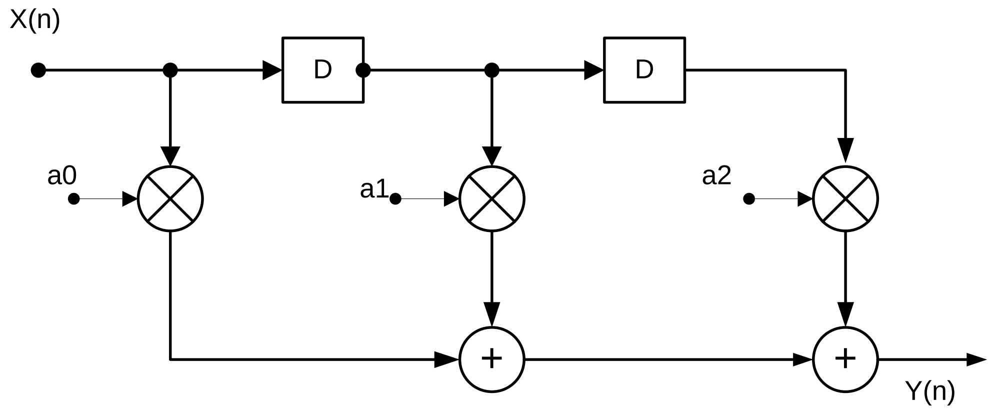
Algorithm transformations play an important role in the design of dedicated architectures (Parhi, 1989). This is because the transformed algorithms can be made to operate with better performance (where the performance may be measured in terms of speed, area, or power). Examples of these transformations include pipelining, parallel processing, retiming, unfolding, folding, and look-ahead, associativity, and distributivity. These transformations and other architectural concepts are described in detail in subsequent sections.

Pipelining

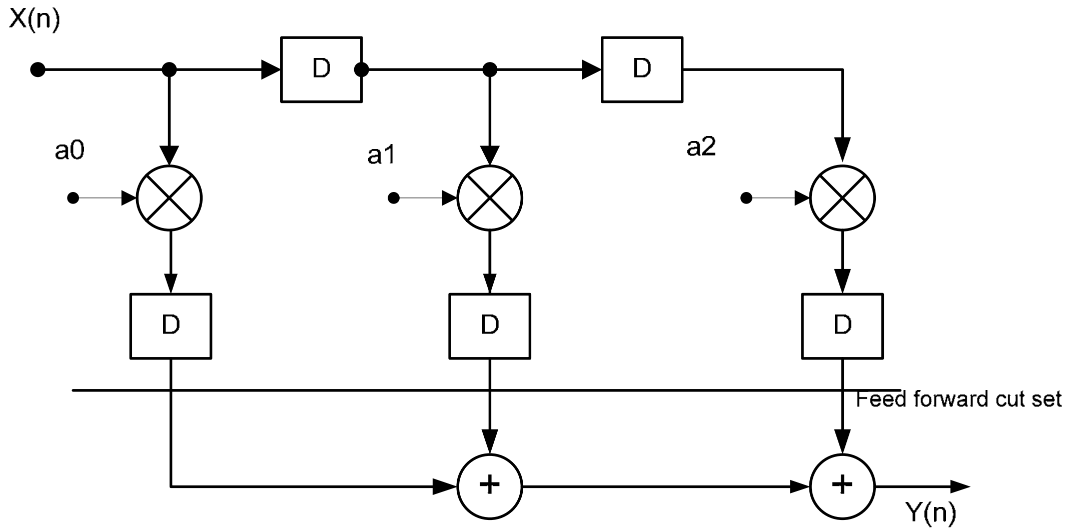
Pipelining is a common design method that is used to increase the throughput of digital circuits. Pipelining is especially important for field-programmable gate array (FPGA) circuits due to the long combinational delay of logic functions and interconnects. Most high-performance FPGA circuits have some form of pipelining. In most cases, pipelining can be implemented without using additional FPGA resources.

Pipelining can increase the amount of the number of tasks performed simultaneously in an algorithm. Pipelining is accomplished by placing registers at intermediate points in a data flow graph that describes the algorithm. The registers can be placed at feed-forward cut sets of the data flow graph. In pipelining, different instances of programs are partially overlapped during execution. Depending on what is considered as a basic building block of a program, three forms of pipelining are most often considered: sub operational (just pipelining), control loop pipelining (software pipelining, folding) and functional pipelining. The program execution in sub operational pipelining is an operation, in control pipelining, program loop has to be pipelined and in functional pipelining only the instance is program which is executed iteratively.

In synchronous hardware implementations, pipelining increases the clock rate of the system (and therefore the sample rate). The drawbacks of pipelining are the increase in no. of clock cycles and the increase in the number of registers. For many digital signal processing (DSP) designs, however, simply increasing the clock speed does not significantly improve the design. In many medium sample rate applications, the design goal is to minimize the total system area by mapping the design onto the smallest (i.e., cheapest) possible FPGA device. To show that the speed increases by using pipelining, consider the second-order three-tap finite impulse response (FIR) filter shown in Figure 6.1(a). The signal x(n) in this system can only be sampled at a low rate due to the throughput of one multiplication and two additions. For simplicity, if we assume the multiplication time is two times more than the addition time, the effective sample or clock rate of this system is 1/4Tadd. By placing registers as shown in Figure 6.1(b) at the cut set shown by the line, the sample rate can be improved to the rate of one multiplication or two additions. While pipelining can be easily applied to such algorithms where there are no feedback loops by the appropriate placement of registers, it cannot easily be applied to algorithms with feedback loops. This is because the cut sets in feedback algorithms have both feed-forward and feedback data flow and cannot be considered only feed-forward cut sets.



(a) A three Tap, second order non-recursive digital filter



(b) Equivalent pipelined digital filter

Figure 6.1 Representation of pipelining

An important parameter of every pipelined system is the input introduction time (*δ*). In software pipelining, this parameter is named as the initiation interval [23]. For digital circuits, this parameter is measured in clock cycles and specifies the number of clock cycles between the initiations of sequential iterations of the pipelined computation. An input introduction time of δ = 1represents a fully pipelined circuit with a new computation initiated every clock cycle [24].

Pipelined circuit modules are characterized with two parameters. The first parameter is the module latency (*λm*), which is the number of clock cycles to perform a single computation. The second parameter is the module input introduction time (*δm*), which is the number of clock cycles separating the next input to the module. These two parameters completely describe the timing behavior of pipelined circuit modules for the scheduling process. Fully pipelined circuit modules are those where $\delta m = 1$ and $\lambda m > 1$ (i.e., a new computation can be initiated every clock cycle). Partially pipelined modules occur when $1 < \delta m < \lambda m$. Non pipelined modules occur when $\lambda m = \delta m$ [24].

Pipelining of algorithms can increase the sample rate of the system. Sometimes, for a constant sample rate, pipelining can also reduce the power consumed by the system. This is due to the fact that the data paths in the pipelined system can be charged or discharged with lower supply voltage. Achieving low power can be important in many battery-powered applications.

The presence of feedback reduces the ability to increase throughput using pipelining. Each cycle *C* within a dependence graph must be broken by a sample delay node or an edge representing data dependence between different iterations of the pipelined computation.

The use of pipelined circuit modules increases the possibilities of resource sharing. A pipelined multi cycle circuit module can increases the sharing since a new operation may be started on the module before the previous operation has completed. A multi cycle pipelined circuit module may be shared if the input introduction time of the module is less than or equal to half the data introduction interval of the global pipeline (i.e., $\delta m \leq (1/2)\delta 0$). The maximum number of operators that can share a pipelined module is [24].

$$share_{max} = \frac{\delta}{\delta_m}$$

As shown in above equation, by the initiation time δ we can calculate how much sharing may be possible for each module. For small values of δ, there are limited sharing possibilities. A pipelining in which $\delta = 1$ will result in a circuit with dedicated resources and no opportunities for sharing. With larger values of δ, the data introduction time of the modules reduces than the global data introduction interval. Several projects have shown resource sharing within pipelining, but these approaches are limited to single-cycle or non pipelined circuit modules [26], [27], [28], [29].

Unfolding

How to efficiently and effectively design iterative or recursive algorithms is an important problem in VLSI high level synthesis For example, for a given signal flow graph for any filter (which have many cycles), we need to know how to get a schedule such that a resultant hardware can achieve the highest pipeline rate.

The input algorithm can be described as a *data-flow graph* (*DFG*), which is widely used in many fields; for example, in circuitry [30], in program descriptions [31], [32], [45], etc. In a DFG, nodes represent operations and edges represent relationships. The graph *G* in Fig. 2a is an example of DFG, where the number attaches to a node is the time it needs to compute. A DFG is called a unit-time DFG if the computation time of its every node is one unit. A certain delay count is attached with each edge to represent inter iteration precedence. Unfolding can reduce the cycle period of a DFG. However, the process of unfolding is time-consuming and space-consuming.

The unfolding transformation is like the process of loop unrolling. In J-unfolding, each node is replaced by J nodes and each edge is replaced by J edges. The J-unfolded data flow graph executes J iterations of the original algorithm (Parhi, 1991).

The unfolding transformation can show the hidden parallelism in a data flow program. The achievable iteration period for a J-unfolded data flow graph is 1/J times the length of critical path of the unfolded data flow graph. Unfolding can lead to a lower iteration period in the context of a software programmable multiprocessor implementation.

The unfolding transformation can also be applied in the context of hardware design. If we apply an unfolding transformation in an algorithm which have no feedback, the resulting data flow graph represents a simply parallel algorithm that processes multiple samples or words in parallel every clock cycle. If we apply two-unfolding to the three-tap FIR filter in Figure 6.1(a), we can obtain the data flow graph of Figure 6.2 (Richard C. Drof, 2006).

Because the unfolding algorithm is based on graph theory approach, it is possible to apply this at the bit level. Thus, unfolding of a bit-serial data flow program by a factor of J leads to a digit-serial program with digit size J. The digit size represents the number of bits processed per clock cycle. In the digit-serial design, clocked at the same rate as the bit-serial. Because the digit-serial program processes J bits per clock cycle the effective bit rate of the digit-serial design is J times higher. In both ways, the unfolding transformation can be applied to both word level and bit level simultaneously to generate word-parallel, digit-serial architectures. Such architectures process multiple words per clock cycle and process a digit of each word (not the entire word).

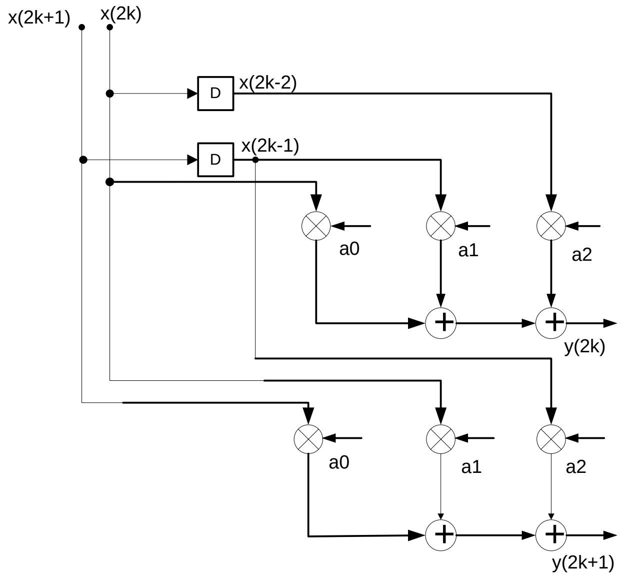


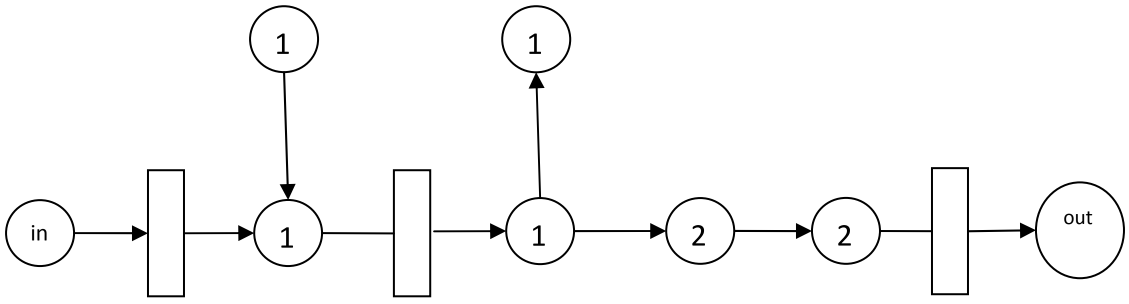
Figure 6.2 Two Fold Parallel Realization of three Tap digital filters

Retiming

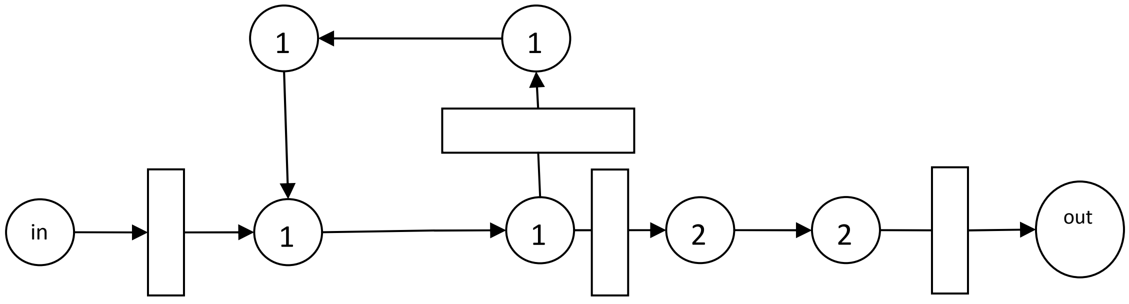
Concepts

Retiming is similar to pipelining but still different in many ways (Leiserson et al., 1983). Retiming is the process of moving the delays around in the data flow graph. Removing one delay from all input edges of a node and putting of one delay to each outgoing edge of the same node is the simplest example of retiming. Unlike pipelining, retiming does not increase the latency of the system. However, retiming changes the number of delay elements in the system. Retiming can reduce the critical path of the data flow graph. As a result, it reduces the clock period in hardware implementations or critical path of the acyclic precedence graph or the iteration period in programmable software system implementations. One effect of changing the locations of the delays is the reduction in combinational rippling that allows the circuit to be clocked at a higher rate. Reducing combinational rippling also reduces the power dissipation in the circuit [33] and allows the circuit to be operated with a reduced supply voltage, both of which lead to low-power implementations [46]. Another effect of changing the locations of delays is that the number of delay elements required can be reduced, which is an area-efficient implementations. In addition to retiming for high speed, low power, and low area implementations, retiming can also an important step towards scheduling for high-level synthesis [47]–[49].

Figure 6.4 shows a simple example. In this design, the nodes represent logic delays (a), with the inputs and outputs passing through fixed registers. The critical path is 5, and the input and output registers cannot be moved. Figure 6.4(b) shows the same graph after retiming. The critical path is reduced from 5 to 4, but the I/O registers have not been changed, as three cycles are still required for a datum to proceed from input to output.



(a) A graph before retiming



(b) Graph after retiming

Figure 6.4 Representation of retiming

As can be seen, the initial design has a critical path of 5 between the internal register and the output. If the internal register could be moved forward, the critical path would be reduced to 4. However, in that case the feedback loop would not work properly. Thus, in addition to moving the register forward, another register is required to be added to the feedback loop, resulting in the final design. Now, even if the last node has been removed, critical path lower cannot be reduced than 4 because of the feedback loop. There is no method that can reduce the critical path of a single-cycle feedback loop by moving registers: Only additional registers can speed up such a design.

The initial state of a circuit is determined by the initial values of the registers in the circuit. For a limited set of applications, e.g., for the data path circuitry in DSP type circuits, the initial state is not important and retiming without additional constraints can be applied [48]. But, in many microprocessor and controller type applications, the initial state is very important part of the behavior of the machine. A retimed circuit has an initial state equivalent to an initial state in the original circuit if for any input sequence applied to both circuits. One circuit started in the initial state, the other in same state, the same sequence of results is produced. One method to design circuit that a corresponding initial state can be found in the retimed network is to only move registers forward in the network [44].

Pipelining with retiming is an important transformation in DSP hardware design. Pipelining with retiming can be defined as the retiming of the original algorithm with a large number of delays at the input edges. Thus, we can increase the system latency to any amount and remove the appropriate number of delays from the inputs after the transformation.

Algorithm

The main objective of retiming is to automate this process: For a graph representing a circuit, with combinational delays as nodes and integer weights on the edges, the requirement is to find a new assignment of edge weights that meets a targeted critical path or fail if the critical path cannot be met. Leiserson's retiming algorithm is guaranteed to find such an assignment, if it exists, that both minimizes the critical path and ensures that around every loop in the design the number of registers always remains the same. It is this second constraint, ensuring that all feedback loops are unchanged, which ensures that retiming doesn't change the semantics of the circuit.

Leiserson's algorithm takes the graph as input and then adds an additional node, with appropriate edges added to account for all I/Os. This additional node is necessary to ensure that the circuit's global I/O registers are unchanged by retiming.

Two matrices are then calculated, W and D, that represents the number of registers and critical path between every pair of nodes in the graph. These matrices are important because retiming operates by ensuring that at least one register exists on every path that is longer than the critical path in the design. Each node also has a lag value r that is calculated by the algorithm and used to change the number of registers that will be placed on any given edge. Conventional retiming does not change the design semantics: All input and output timings remain unchanged while minor design constraints are imposed on the use of FPGA features. More details and formal proofs of correctness can be found in Leiserson's original paper [43].

According to the fig. 6.1, No edge will have a negative number of registers, while every cycle will always contain the original number of registers. All I/O passes through the intermediate node, to ensure that input and output timings do not change. These constraints must be modified so that a particular line will contain no registers, or a minimum number of registers, to meet architectural requirements without changing the complexity of the equations. But it is the final requirement, that all critical paths above a predetermined delay *P* are registered, that gives this optimization its effectiveness.

Memories themselves can be retimed similarly to any other element in the design, in memories with dual ports, considered as a single node for retiming purposes.

Some FPGA designs have registers with predefined initial values. If retiming is allowed to move these registers, the proper initial values must be calculated so that the circuit still produces the same behavior.

Another important thing is how to process with multiple clocks. If the interfaces between the clock domains are registered by clocks from both domains, it is a simple process to retime the domains separately. Yet without this design requirement, retiming with multiple clock domains is very difficult.

Limitations

The major problem with retiming is the limitation of benefit to a well balanced design. If the clock cycle is defined by a single cycle feedback loop, retiming cannot improve the architecture, as just moving the register along the feedback loop produces no effect. A familiar example of this is The AES encryption algorithm which consists of a single cycle feedback loop.

Nevertheless, retiming can still be a best approach if the design contains of multiple feedback loops (such as the synthetic microprocessor datapath) or an initially unbalanced pipeline. Still, for balanced circuits, even complex designs, retiming is often only a smart benefit.

Repipelining

For most reconfigurable designs, this means that the placement can also be changed to add the new registers. As the new placement may not have the same timing characteristics as the original placement. This results the reduction in the accuracy of the retiming, by slowing the circuit down instead of speeding it up [42].

Repipelining is a small extension to retiming that can change the clock frequency for feed forward computations at the cost of additional pipeline registers. Unlike *C*-slow retiming, repipelining is only better when a computation's critical path contains no feedback loops. Feed forward computations, those that contain no feedback loops, are commonly seen in DSP algorithms. For example, the discrete cosine transforms (DCT), the fast Fourier transforms (FFT), and finite impulse response filters (FIRs) can all be constructed as feed forward pipelines.

Repipelining is obtained from retiming in one of two ways, both of which produce same results. The first is done by adding additional pipeline registers to the start of the computation and allowing retiming to rebalance the registers and create an absolute number of additional stages. The second is to decoupling of the inputs and outputs to allow the retimer produce pipelining. Although these techniques operate in somewhat different ways, they both provide extra registers for the retimer to then move and they produce almost similar results.

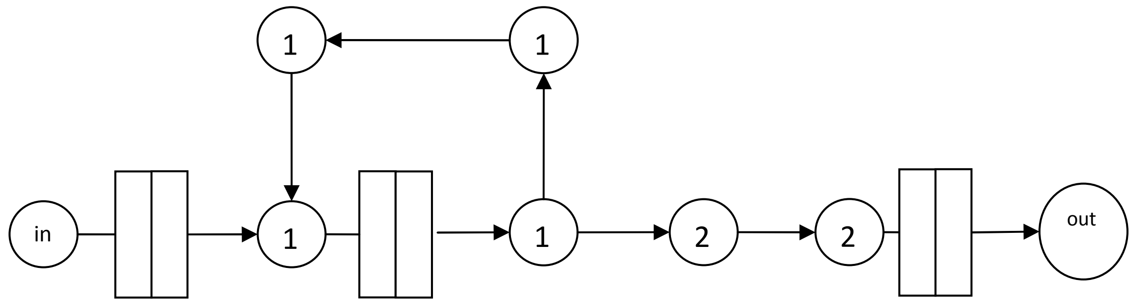
Repipelining produces additional clock cycles and thus increases latency to the design. Thus, it produces the same results and the same relative timing on the outputs. It is only the data-in to data-out timing that is affected.

Repipelining is designed to improve throughput, but will almost always make overall latency worse. Although the increased pipelining will boost the clock rate and thus reduce some of the delay from unbalanced clocked paths, the delay from additional flip-flops on the input-to-output paths typically overwhelms this improvement and the resulting design will take longer to produce a result for an individual input.

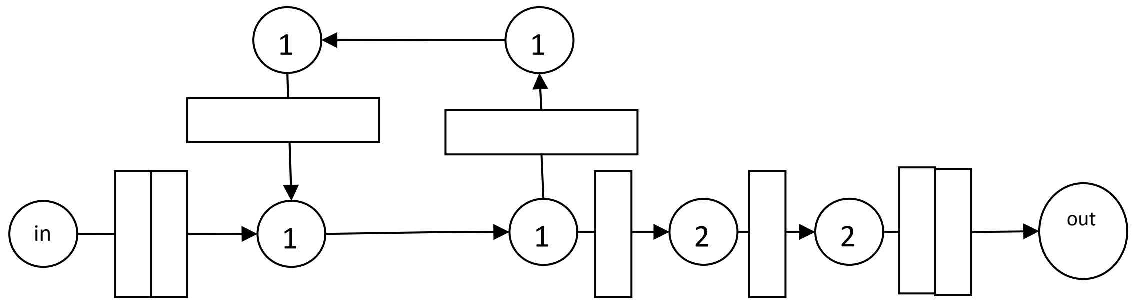
C-Slow Retiming

Unlike repipelining, C-slow retiming can enhance designs that contain feedback loops. C-slowing enhances retiming simply by replacing every register with a sequence of C separate registers before retiming occurs; the resulting design performs C separate execution tasks. [41**]** The effect of C-slow retiming is to ensure pipelining of the critical path, even in the presence of feedback loops. To take advantage of this increased throughput however, there needs to be parallelism in the tasks. This process will slow any single task but the average throughput will be increased by interleaving the resulting computation. C-slow style retiming has long been used within the cryptographic community when implementing block ciphers [40], [39].

In the example below, this is 2-slow, the design interleaves between two computations. On the first clock cycle, it accepts the first input for the first stream of execution. On the second clock cycle, it accepts the first input for the second stream, and on the third it accepts the second input for the first stream. Because of the interleaved nature of the design, the two streams of execution will never interfere. On odd clock cycles, the first stream of execution accepts input; on even clock cycles, the second stream accepts input.



(a) 2-slow operation of fig 6.4



(b) After retiming the same designs
Figure 6.5 C-slow operational diagram

The easiest way to utilize a C-slowed block is to simply multiplex and de-multiplex C separate datastreams. However, a more very good interface is required depending on the application. One possible interface is to register all inputs and outputs of a C-slowed block. Because of the additional edges retiming creates to track I/Os, every stream of execution presents all outputs at the same time, with all inputs registered on the next cycle. If part of the design is C-slowed, but all parts operate on the same clock, the result can be retimed as a complete whole and still preserve all other constraints.

C-Slow and Memory Design

One major issue in C-slow operation is the need to handle random access memories within a C-slowed block. [34]. In cases where the C-slowed design is used to support *N* independent computations, one needs to create the illusion that each stream of execution is completely independent and unchanged. To create such illusion, the memory capacity must be increased by a factor of *C*. This ensures that each stream of execution enjoys a completely separate memory space. There are two different types of memory which need to be addressed when C-slowing a design, either automatically or manually. The first is isolated memory, where each thread of execution has its own view of memory. The second is shared memory, where all streams of execution share a common memory view.

C-Slow and Throughput

C-slowing significantly improves throughput, but it can only apply to tasks where there are at least *C* independent threads of execution and where throughput is the primary goal. The reason is that *C*-slowing makes the latency much worse. Latency is a property of the design and computation whereas throughput is a property derived from cost. Both repipelining and *C*-slow retiming can be applied only when there is sufficient task-level parallelism, in the form of either a feed forward pipeline (repipelining) or independent tasks (*C*-slowing).

Latency can be improved only up to a given point for a design through conventional retiming. Once the latency limit is reached, no amount of optimization, save a major redesign or an improvement in the FPGA, has any effect. This is because independent task throughput can be added via replication, creating independent modules that perform the same function, as well as *C*-slowing. When sufficient parallelism exists, and costs are not constrained, simply throwing more resources at the problem is sufficient to improve the design to meet desired goals.

C-Slowing As Multithreading

There have been many multi-threaded architecture designs, but all of them have a common property: increasing system throughput by enabling multiple streams of execution, or threads, to operate simultaneously. These architectures generally can be divided into four classes: context switching always without bypassing [38],[36], context switching on event [37], interleaved multi-threaded, and symmetric multi-threaded [35]. The ideal goal of all of them is to increase system throughput by operating on multiple streams of execution.

The general concept of *C*-slow retiming can be applied to highly complex designs, including microprocessors. Unlike a simple FIR filter bank or an encryption algorithm, it is not a simple matter of inserting registers and balancing delays. Nevertheless, the changes necessary are comparatively small and the benefits substantial: producing a simple, statically scheduled, higher clock rate, multi-threaded architecture that is semantically equivalent to an interleavedmulti-threaded architecture, alternating between a fixed number of threads in a round-robin fashion to create the illusion of a multiprocessor system.

The biggest complications in *C*-slowing a microprocessor are selecting the implementation semantics for the various memories through the design. The first type keeps the traditional *C*-slow semantics of complete independence, where each thread sees a completely independent view, usually by duplication. This applies to the register file and most of the state registers in the system. This occurs automatically if *C*-slowing is performed by a tool, because it represents the normal semantics for *C*-slowed memory.

The second is completely shared memory, where every thread sees the same memory, such as the caches and main memory of the system. Most such memories exist in the non-*C*-

slowed portion. The third is dynamically shared, where a hardware thread number is tagged to each entry, with only the valid tags used. This breaks the automatic *C*-slow semantics and is best employed for branch predictors and similar caches. Such memories need to be constructed manually, but offer potential efficiency advantages as they do not need to increase in size. Because they cannot be constructed automatically they may be subject to interference or synergistic effects between threads.

The biggest architectural changes are to the register file: It needs to be increased by a factor of *C*, with a hardware thread counter to select which group of registers is being accessed. Now each thread will see an independent set of registers, with all reads and writes for the different threads going to separate memory locations. Apart from the thread selection and natural enlargement, the only piece remaining is to pipeline the register access.

Cost of C-Slow

There are four primary costs associated with C-slow retiming as discussed in [34]. These are increased latency for single computations, greater circuit area, increased power consumption, and the costs of interacting with single-instance. The increased latency is the biggest concern, caused by unbalanced pipeline stages and the setup and hold time from the additional pipeline stages. The unbalanced stages can be reduced through the use of conventional retiming, but some imbalance will undoubtedly remain.

The second major concern is circuit area, and is a direct function of how aggressively the design is retimed. With modern FPGAs possessing as many registers as 4-LUTs, most designs underutilized the registers available. Thus, for low levels (2-4 slow) retiming, the resulting circuit cost is usually very low as the registers can be combined with the associated logic blocks.

Power consumption is naturally increased due to the higher clock rates and larger number of elements involved. If lower power is desired, the higher-throughput design can be modified to save power by reducing the clock rate and operating voltage. Although the finer pipelining allows the frequency and the voltage to be scaled back to a significant degree while maintaining throughput, the activity factor of each signal may now be considerably higher. Because each of the *C* streams of execution is completely independent, it is safe to assume that every wire will probably have a significantly higher activity factor that increases power consumption.

The final cost of C-slow retiming results from the increased complexity of interacting with other blocks in the design. If there is a feedback loop between a C-slowed block and a different block, then the different block should be included within the C-slow retiming process. Otherwise, one simply needs to interleave and de interleave the data streams coming to and from the C-slowed block.

Examples of C-slow Retiming

IIR Implementation

IIR filters are very important in digital signal processing systems. Now we design IIR filter without c-slow retiming and then with 2-slow. After that we will compare results of IIR and FIR as the example circuits of feed forward and feedback circuits.

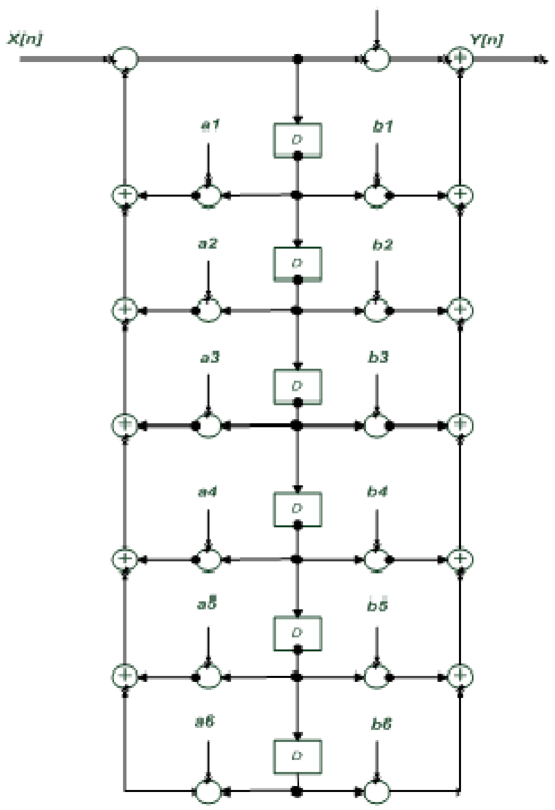


Figure 6.9 Simple 8-tap IIR filter

Chapter 7

C-Slowed Microprogrammed Finite State Machine

Introduction

In this section, we will design a simple micro programmed processor with a simple, and then the same architecture is C-Slowed and checks the performance results. In first case it can execute one instruction per clock cycle. After 2 C-Slow, it executes two instructions per clock cycle or it can be said it can execute two streams parallel. There are two major parts of the thesis. One is Processor and the other is micro programmed control unit that is used perform different tasks on the stored data.

Design of Control Unit

The main purpose of the control unit is to translate or decode instructions and generate appropriate enable signals to accomplish the desired operation. Based on the contents of the instruction register, the control unit sends the selected data items to the appropriate processing hardware at the right time. The control unit drives the associated processing hardware by generating a set of signals that are synchronized with a master clock.

The control unit performs two basic operations: instruction interpretation and instruction sequencing. In the interpretation phase, the control unit reads (fetches) an instruction from the memory addressed by the contents of the program counter into the instruction register. The control unit inputs the contents of the instruction register. It recognizes the instruction type, obtains the necessary operands, and routes them to the appropriate functional units of the execution unit (registers and ALU). The control unit then issues the necessary signals to the execution unit to perform the desired operation and routes the results to the specified destination.

In the sequencing phase, the control unit generates the address of the next instruction to be executed and loads it into the program counter. To design a control unit, one must be familiar with some basic concepts such as register transfer operations, types of bus structures inside the control unit, and generation of timing signals.

Basic Concepts

Register transfer notation is the fundamental concept associated with the control unit design. For example, consider the register transfer operation.

$R1 \leftarrow R2$

The symbol ← is called the transfer operator. However, this notation does not indicate the number of bits to be transferred. An enable signal usually controls transfer of data from one register to another. The enable input may sometimes be a function of more than one variable.

Micro Programmed Control Unit Design

As mentioned earlier, a micro programmed control unit contains programs written using microinstructions. These programs are stored in a control memory normally in a ROM inside the CPU. To execute instructions, the microprocessor reads (fetches) each instruction into the instruction register from external memory. The control unit translates the instruction for the microprocessor. Each control word contains signals to activate one or more micro operations. A program consisting of a set of microinstructions is executed in a sequence of micro-operations to complete the instruction execution. Generally, all microinstructions have two important fields:

1. Control word
2. Next address

The length of a microinstruction is directly related to the following factors: The number of micro-operations that can be activated simultaneously, this is called the "degree of parallelism." And the method by which the address of the next microinstruction is determined. In my design the micro programmed control unit contains the following components:

1. Micro program counter register
2. Incrementer
3. Micro programmed Memory (ROM/RAM)
4. Next Address selection logic
5. Control Data Register

Control Data register has three parts, control signals, branch address and condition select. Control signals to use to control the data transfer through data path. While branch address and condition select are used to select the next instruction in the sequence. If there is no select condition then micro instruction executes in a sequence otherwise it will jump at branch address as shown in the fig. 7.1. Now we discuss different components here:

Micro program Counter (MPC)

The MPC holds the address of the next microinstruction to be executed. It is initially loaded from an external source to point to the starting address of the micro program. The MPC is similar to the program counter (PC). The MPC is incremented after each microinstruction fetch. If a branch instruction is encountered, the MPC is loaded with the contents of the branch address field of the microinstruction.

MUX (Multiplexer)

The MUX is a condition select multiplexer. It selects one of the external conditions based on the contents of the condition select field of the microinstruction fetched into the CWR. Here is an example of 2-bit condition select MUX.

Condition Select Field		Interpretation
0	0	No Branching
1	0	Branch if Z=0
0	1	Unconditional Branch

In micro programmed controller, the length of Instruction word is 24 bit. In this 4 bits are for condition selection, for branch address, there are 7 bits and 13 bits specified for control signals i.e. c0 to c12 that are used to control data path of machine.

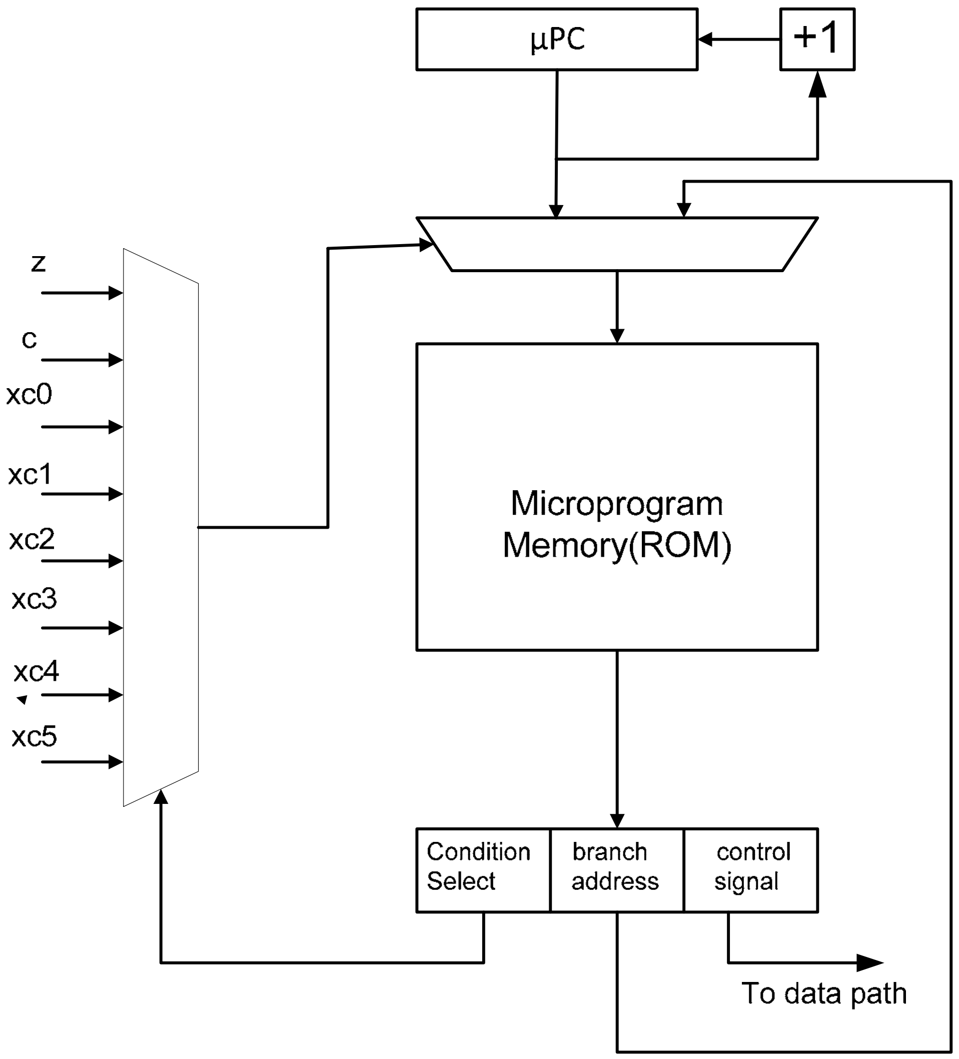


Figure 7.1 Microprogrammed Control Unit

Design of Micro programmed CPU

Next, Micro programmed CPU design is illustrated. The CPU contains two registers:

1. An 8-bit register A
2. A 2-bit flag register F

The flag register holds only zero (Z) and carry (C) flags. All programs and data are stored in the 256 x 8 RAM. The detailed hardware schematic of the data-flow part of this processor is shown in Figure 7.2.

It can be seen that the hardware organization includes four more 8-bit registers, PC, IR, MAR, and BUFFER. These registers are transparent to a programmer. The 8-bit register BUFFER is used to hold the data that is retrieved from memory. In this system, only a restricted number of data paths are available. These paths are controlled by the control inputs C0 through C9.

The proposed instruction set contains 11 instructions. The first 7 instructions are classified as memory reference instructions, since they all require a memory address (which is an 8-bit number in this case). The last 4 instructions do not require any memory address; they are called non memory reference instructions. Each memory reference instruction is assumed to occupy ***2*** consecutive bytes in the RAM. The first byte is reserved for the op-code, and the second byte indicates the 8-bit memory address. In contrast, a non memory reference instruction takes only one byte of storage. This instruction set supports only two addressing modes: implicit and direct. Both branch instructions are assumed to be absolute mode branch instructions. The op-code encoding for this instruction set is carried out in a logical manner. If **I3** = 1, it is a memory reference instruction (MRI), otherwise it is a non memory reference instruction (NMRI). Within the memory reference category, instructions are classified into four groups, as follows:

GROUP NO.	INSTRUCTIONS
0	Load and store
1	Add and subtract
2	Jumps
3	Logical

As mentioned before, the instruction execution involves the following steps:

MICROOPERATION	COMMENT
C0: PC ← 0	Clear PC to zero
C1: PC ← PC+1	Advance the PC
C2C5C6: PC← M ((MAR))	Read the data from the memory and save it in PC
C3C4: MAR ← PC	Transfer the contents of PC
C5C6C7: Buffer ← M(MAR)	Read the data from the memory and store the result in buffer
C3C4: MAR ← Buffer	Insert the buffer contents into memory address register
C5C6C8: IR ← M(MAR)	Read the data from memory and the result into IR
C9: A ← F	Transfer the ALU output into the register A
C5C6: M(MAR) ← A	Store the contents of register A into memory location specified in MAR

The eight ALU operations performed by C10C11C12 are given below:

Cl0	**C11**	**Cl2**	**F**
0	0	0	0
0	0	1	R
0	1	0	L+R
0	1	1	L-R
1	0	0	L+ 1
1	0	1	L- 1
1	1	0	L AND R
1	1	1	NOT L

Step1: Fetch the instruction.
Step2: Decode the instruction to find out the required operation.
Step3: If the required operation is a halt operation, then go to Step6 otherwise continue.
Step4: Retrieve the operands and perform the desired operation.
Step5: Go to Step 1.
Step6: Execute an infinite LOOP.

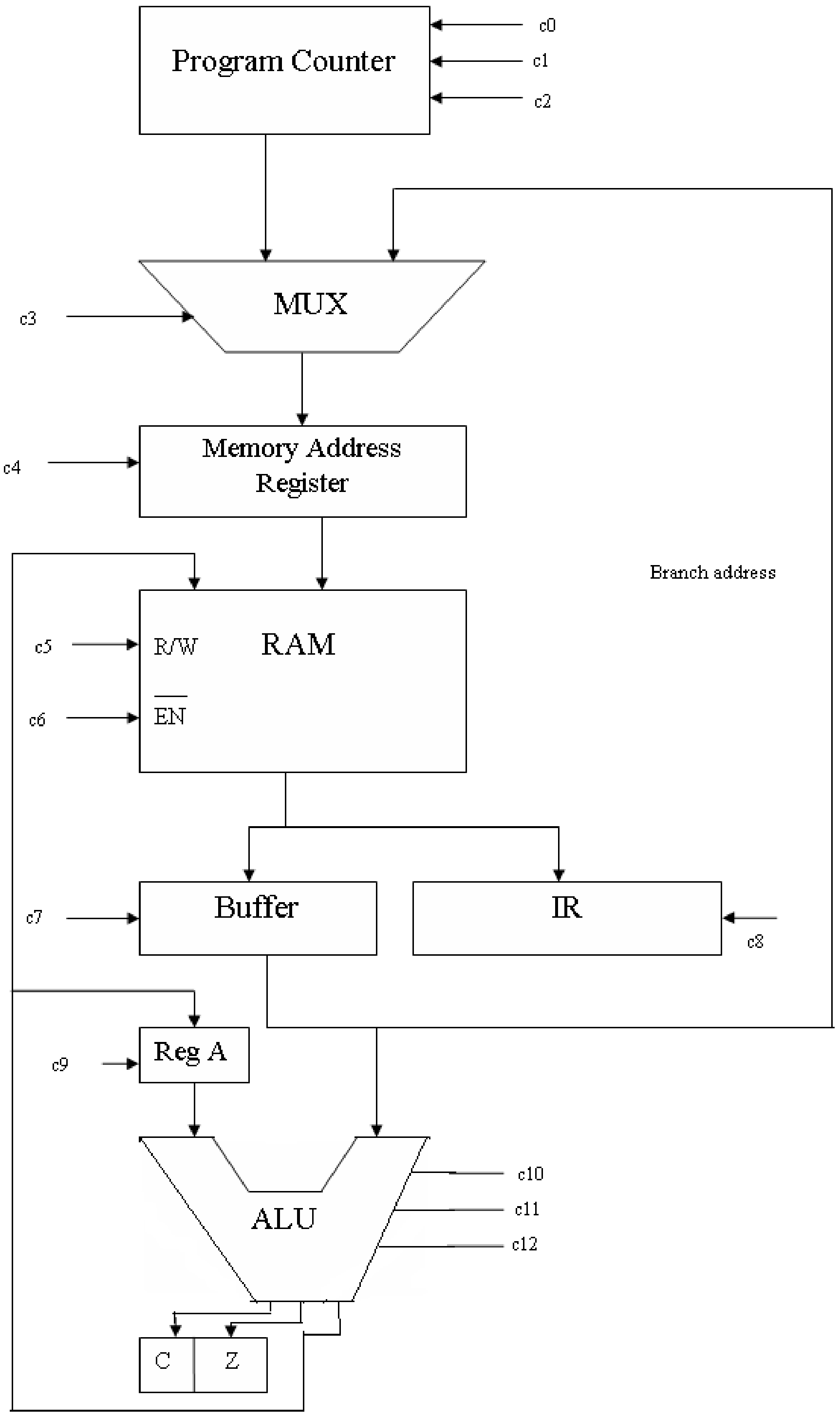


Figure 7.2 A simple Processor design

C-Slowed Micro programmed Controller and Processor

C-slowed micro programmed counter is shown in the figure. As discussed earlier, in c-slow register increased c times, and also the memory. So in control unit, there are two micro programmed counters for each micro program. Which instruction is to execute is control by multiplexers and de multiplexers.

As shown in the figure, selection of control signals is based upon the clock. At clock0, control signals from micro programmed memory 1 are generated while at clock 1, control signals are from memory 2. Thus 2 micro instructions completes once the one clock cycle completes.

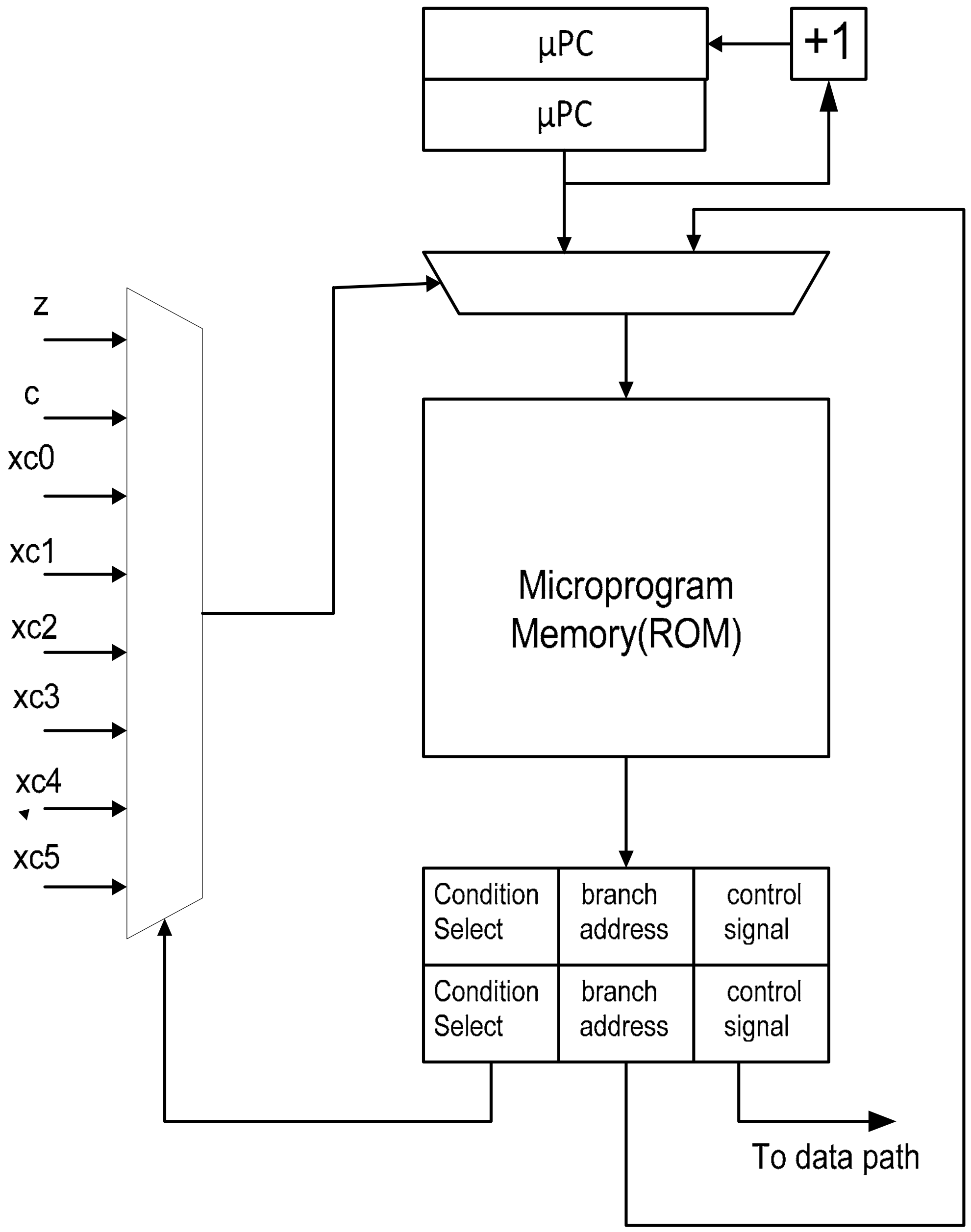


Figure 7.3 C-Slowed Microprogrammed Controller

Chapter 8

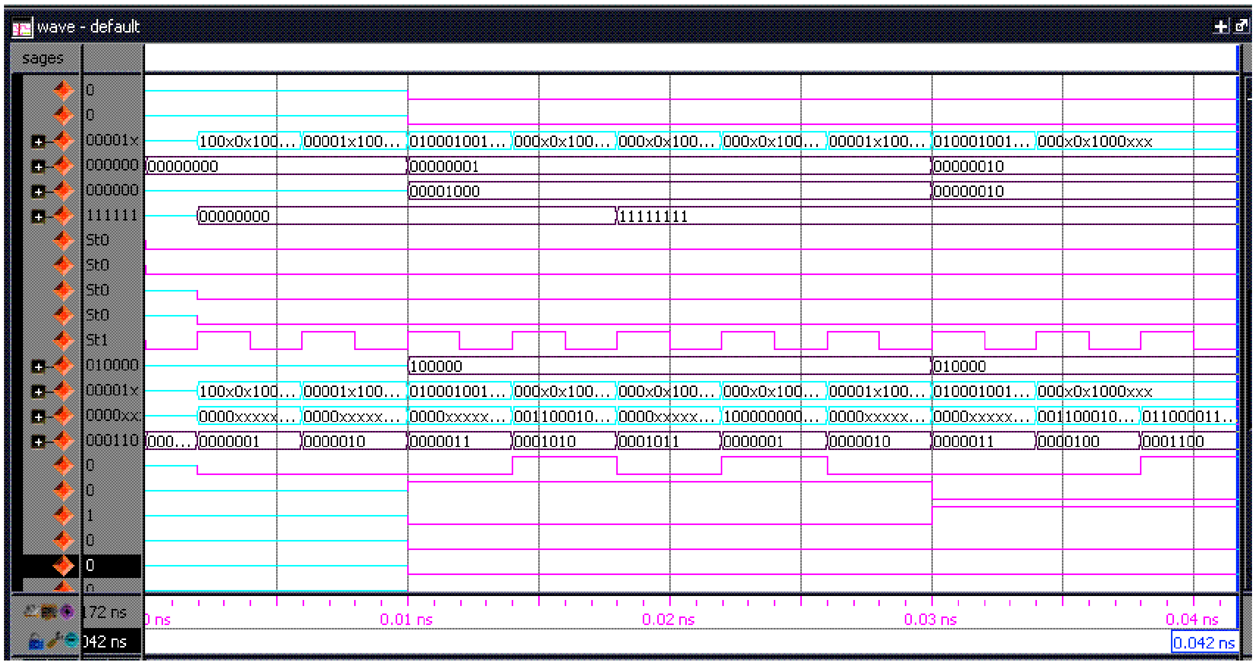
Results Analysis

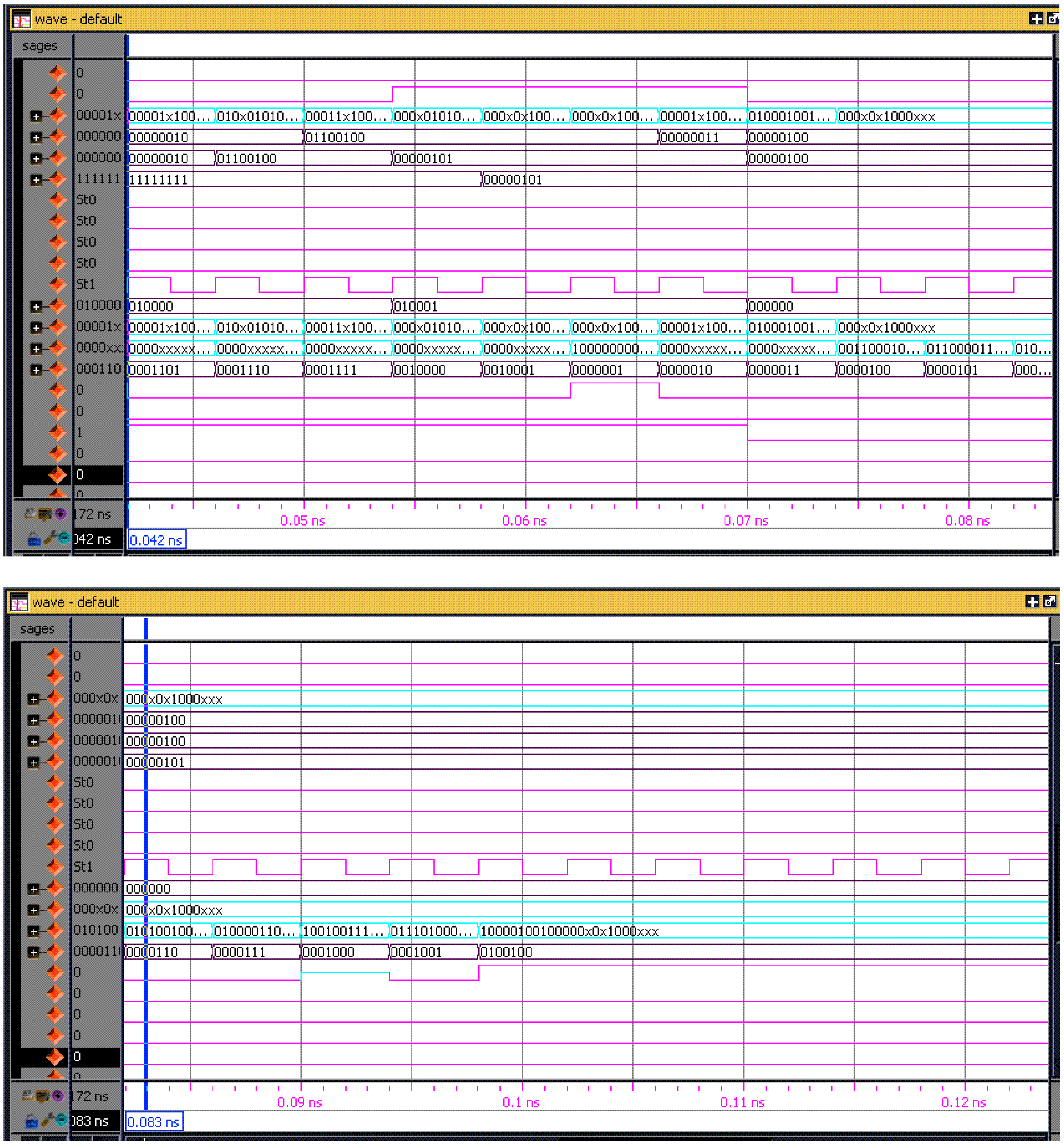
Simulation Results Analysis

No. of cycles required to process without c-slow = 25 clock cycles

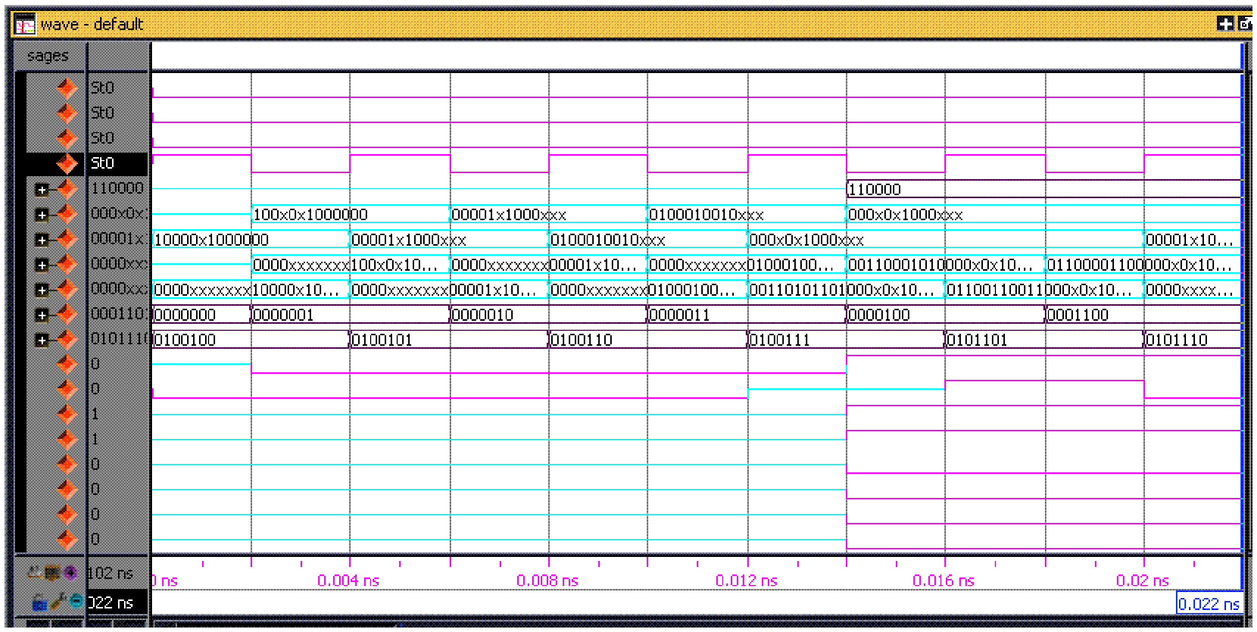
No. of clock cycles required to process with c-slow = 10 clock cycles

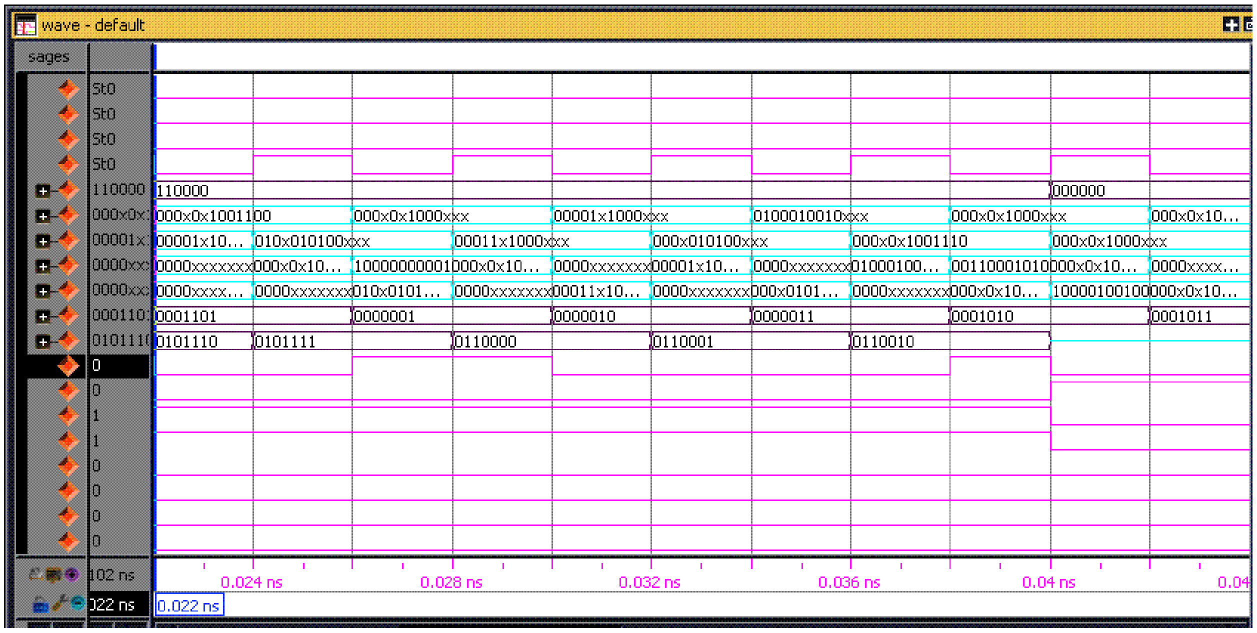
Timing diagrams of both of these results are shown in the fig.7.4 in which (a) show the results without c-slow while (b) shows the results in the presence of c-slow.





(a) Timing results without C-slow





(b) Results with C-slow

Figure 7.4 (a) Timing results without C-slow (b) Results with C-slow

Synthesis Results without C-slow

Table 8.1Device utilization summary

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	107	6,624	7%

Number of 4 input LUTs	444	6,624	5%
Logic Distribution			
Number of occupied Slices	778	3,312	13%
Number of Slices containing only related logic	778	778	100%
Number of Slices containing unrelated logic	0	778	0%
Total Number of 4 input LUTs	454	6,624	5%
Number used as logic	444		
Number used as a route-thru	0		
Number of bonded IOBs	9	333	2%
Number of GCLKs	2	8	25%

Synthesis Results with C-slow

Table 8.2 Device utilization summary

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	4,270	26,624	16%
Number of 4 input LUTs	3,166	26,624	11%
Logic Distribution			
Number of occupied Slices	3,708	13,312	27%
Number of Slices containing only related logic	3,708	3,708	100%
Number of Slices containing unrelated logic	0	3,708	0%
Total Number of 4 input LUTs	3,167	26,624	11%
Number used as logic	3,167		
Number used as a route-thru	1		
Number of bonded IOBs	10	333	3%
Number of GCLKs	2	8	25%

Table 8.3 Timing Summary

	Microprogrammed FSM	**2-slow Microprogrammed FSM**
Number of clock cycles required to process two tasks	25	10
Minimum period	29.976ns	11.558ns
Minimum input arrival time before clock	29.985ns	5.458ns
Maximum Frequency	33.360MHz	86.520MHz
Maximum output required time after clock	7.165ns	7.165ns

Conclusion

According to the results obtained, it is clear that C-slow increases the throughput of the system. In the sixth chapter, we have discussed that C-slow increase the throughput at the cost of increased latency in case of feed forward circuits but if there are feedback loops in the design, then there is not much effect. It is also discussed that as in C-slow, we increase the number of registers so there is an increase in area of the design also. Now we will discuss all these things in the presence of obtained results.

The first thing due to which we have to apply the c-slow is the throughput. And according to the simulation and synthesis results this is obtained in a good factor. From the simulation results, total number of clock cycles required to process two independent instruction sets is 25. But on the other hand if we simulate the design after applying C-slow then the same results are obtained within just 10 clock cycles. So the throughput of the system increases by decreasing the number of clock cycles required. And there is another effect that the frequency of the system increases by decreasing the minimum time required to process i.e. at faster clock speed. These are the results what we obtain from synthesizing the design. In first case, instructions are executed one by one. Micro program processor design that is without C-slow, executes one micro instruction at one clock cycle. As in the simulation results it executes two instructions one is for invert and second is addition of stored data to the ALU register. And the same case is applied to the design when micro programmed architecture is c-slowed, these micro instructions are executed in parallel i.e. inverting and addition instructions are executed at the same time i.e. time to simulate both instructions is equal to the time required to execute the single instructions.

On the other hand, if we analyze the results on the basis of hardware used then we see a clear picture in which c-slowed architecture utilize more hardware. As discussed c-slowed in filter design example, in the cases when the architecture is simple or when there is no feedback architecture, c-slowed simply becomes an example of repipelining. Synthesis results in the previous chapter show when we c-slow the feed forward architecture, it increase the frequency of execution much more than without c-slow but we have to face a situation in which output time for a single instruction increases, same behavior as in pipelining.

But when we apply same phenomena on the complex architecture, then we have a very different picture. In these cases, after c-slow the frequency of execution does not increase but somewhat increases because in this case as two instructions execute, we assume our frequency becomes double. At the same time, output clock period time is reduced about to half, thus the through put becomes double.

Another effect after C-slow retiming is the increase in the area. Now we see what the effect of this. From the above synthesis results, the area constraint or the number of FPGA slices, flip-flop slices, and the number of Look Up Tables (LUT) are shown. From these results it is clear that C-slow retiming increases the throughput at the cost of area used. But on the other hand if we analyze the area used, we obtain information that before C-slow the percentage of area used is seven percent for slice flip-flops, and after C-slow the percentage area used is sixteen percent. It means we have a lot number of resources available for C-slowing the design and still we have resources.

In this simple processor design we just implement 2-Slow design in which the area constraint does not effect much because the number of available resources is very large. But in case we design a circuit which is 4-Slow or more then area/throughput factor becomes worse. Because in that case the number of resources used is very large.

References

[1] M. V. Wilkes, *"The best way to design an automatic calculating machine,"* Report of the Manchester University Computer Inaugural conference, Electrical Engineering Department of Manchester University, Manchester, England, July 1951, pp. 16-18, reprinted in Earl E. Swartzlander, Jr, ed, Computer Design Development- Principal Papers, Hayden Book Co, Rochelle Park, NJ, 1976, pp. 266-270

[2] P. Fagg et al., "*IBM System/360 engineering,*" in 1964 Fall Joint Comput. Conf. Proc., AFIPS Press, Montvale, NJ, pp. 205-231.] [R. L. Horton, J. Englade, and G. McGee, "*12L takes bipolar integration a significant step forward*," Electronics, pp. 83-90, Jan. 23, 1975.

[3] *"The growth of interest in microprogramming: A literature survey,"* Comput. Surveys, vol 1, no. 3, pp. 139-145, Sept. 1969

[4] S.S.Husson, *Microprogramming Principles and Practices*. Englewood Cliffs, NJ: Prentice-Hall, 1970

[5] *Microprogramming: A Tutorial and Survey of Recent Developments* TOMLINSON G. RAUSCHER, MEMBER, IEEE, AND PHILLIP M. ADAMS

[6] *Microprogramming: A tutorial and survey of recent developments* Tomlinson G. Rauscher and Philip M. adams, IEEE transactions on computers vol. c-29 No.1, Jan 1980

[7] A. K. Agrawala and T. G. Rauscher, Foundations *of Microprogramming- Architecture, Software, and Applications*. New York: Academic Press, 1976**.**

[8] A. Opler, "*Fourth-generation software*," Datamation, vol. 13, pp. 22-24, Jan. 1967

[9] *Microprogramming: an introduction and viewpoint*. IEEE transaction on computers July 1971:727-731

[10] J. C. Demco and T. A. Marsland, "*An insight into PDP-11 emulation,"* SIGMICRO Newslett., vol. 7, pp. 20-26, Sept. 1976

[11] B. L. Ehlers et al., *"Reflective emulations of the HP2100A and Varian 72 minicomputers,"* SIGMICRO Newslett., vol. 7, pp. 26-42, June 1976.

[12] R. A. Belgard, "*A generalized virtual memory package for B1700 interpreter writers,"* SIGMICRO Newslett., vol. 7, no. 4, pp. 31-45, Dec. 1976.

[13] E. G. Mallach, *"Emulator architecture,"* Computer, vol. 8, pp. 24-32, Aug. 1975.

[14] *"Operating system enhancement through firmware,"* SIGMICRO Newslett., vol. 8, pp. 119-133, Sept. 1977

[15] G. Reyling, Jr., *"Extend LSI-Processor capabilities with microprogramming,"*Electron. Des, vol. 22, pp. 90-95, Oct. 25, 1974.

[16] G. DesRochers*, "Microprogramming helps squeeze more from your equipment dollar,"* EDN, pp. 102-105, Sept. 20, 1976

[17] J. Clymer, *"Use 4-bit slices to design powerful microprogrammedprocessors,"* Electron. Des., vol. 25, pp. 62-71, May 10, 1977

[18] D. A. Patterson, *"Strum: Structured microprogram development system for correct firmware,"* IEEE Trans. Comput, vol. C-25, pp. 974-985, Oct. 1976

[19] R. R. Ramseyer and A. van Dam, *"A multimicroprocessor implementation of a general purpose pipelined CPU,"* in Proc. 4th Annu. Symp. Computer Architecture, Mar. 1977, pp. 29-34

[20] S. Davidson, *"A network of dynamically microprogrammable machines,"* SIGMICRO Newsletter., vol. 6, pp. 27-31, Dec. 1975.

[21] T. F. Storey, *"Design of a microprogram control for a processor in an electronic switching system,"* Bell Syst. Tech. J, vol. 55, no. 2, pp. 183-232, Feb. 1976

[22] C. Leiserson, F. Rose, J. Saxe. *Optimizing synchronous circuitry by retiming.* Third Caltech Conference On VLSI, March 1993

[23] B. R. Rau, *"Iterative modulo scheduling: An algorithm for software pipelining loops,"* in Proc. 27th Annu. Int. Symp. MICRO, 1994, pp. 63–74.

[24] *FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing* Welson Sun, Student Member, IEEE, Michael J. Wirthlin, Senior Member, IEEE, and Stephen Neuendorffer. IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 26, NO. 2, FEBRUARY 2007 pp. 254

[25] B. R. Rau, *"Iterative modulo scheduling: An algorithm for software pipelining loops,"* in Proc. 27th Annu. Int. Symp. MICRO, 1994, pp. 63–74.

[26] S. Raje and R. Bergamaschi, *"Generalized resource sharing,"* in Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. Dig. Tech. Papers, Nov. 1997, pp. 326–332

[27] N. Park and A. Parker, "Sehwa: *A software package for synthesis of pipelines from behavioral specifications,"* IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 7, no. 3, pp. 356–370, Mar. 1988

[28] S. Bakshi, D. Gajski, and H. Juan, *"Component selection in resource shared and pipelined DSP applications,"* in Proc. EURO-DAC/EUROVHDL, 1996, pp. 370–375

[29] S. O. Memik, G. Memik, R. Jafari, and E. Kursun, "*Global resource sharing for synthesis of control data flow graphs on FPGAs,"* in Proc. Des. Autom. Conf., Jun. 2003, pp. 604–609

[30] C.E. Leiserson and J.B. Saxe, *"Retiming Synchronous Circuitry,"* Algorithmica, vol. 6, pp. 5–35, 1991

[31] L.-F. Chao and E. H.-M. Sha, "*Retiming and Unfolding Data-Flow Graphs,"* Proc. Int'l Conf. Parallel Processing, vol. II, pp. 33–40, St. Charles, Ill., Aug. 1992

[32] A.L. Davis and R.M. Keller, *"Data Flow Program Graphs,"* Computer, vol. 15, no. 2, pp. 26–41, Feb. 1982.

[33] J. Monteiro, S. Devadas, and A. Ghosh, *"Retiming sequential circuits for low power,"* in Proc. IEEE Int. Conf. Computer Aided Design, 1993, pp. 398–402

[34] *The Effects of Datapath Placement and C-slow Retiming on Three Computational Benchmarks*, Nicholas Weaver, nweaver@cs.berkeley.edu , John Wawarzynek , johnw@cs.berkeley.edu UC Berkeley Computer Science Division UC Berkeley Computer Science

[35] D. M. Tullsen, S. J. Eggers, H. M. Levy. *Simultaneous multi-threading: Maximizing on-chip parallelism*. Proceedings 22nd Annual International Symposium on Computer Architecture, June 1995

[36] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. *The Tera computer system.* Proceedings of the 1990 International Conference on Supercomputing, 1990

[37] Intel Corporation. *The Intel IXP network processor*. Intel Technology Journal 6(3), August 2002

[38] B. J. Smith. *Architecture and applications of the HEP multiprocessor computer system. Advances in laser scanning technology*. SPIE Proceedings 298, Society for Photo-Optical Instrumentation Engineers, 1981

[39] S.Wolter, H. Matz, A. Schubert, and R. Laur. *The vlsi implementation of the international data encryption algorithm idea.* Proceedings of the IEEE International Symposium on Circuits and Systems, vol. 1, pp. 397– 400, 1995

[40] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. *The tera computer system.* In Proceedings of the ACM International Conference on Supercomputing, June 1990

[41] *PostPlacement Cslow Retiming for the Xilinx Virtex FPGA* Nicholas Weaver_ UC Berkeley Berkeley, CA

[42] *Simultaneous Retiming and Placement for Pipelined Netlists* Ken Eguro and Scott Hauck Department of Electrical Engineering

[43] C. Leiserson, F. Rose, J. Saxe. *Optimizing synchronous circuitry by retiming.* Third Caltech Conference On VLSI, March 1993.

[44] S. Dey, M. Potkonjak, and S.G. Rotweiler, *"Performance optimization of sequential circuits by eliminating retiming bottlenecks,"* in Tech. Dig. Papers Int. Con$ Computer-Aided Design, Santa Clara, CA, Nov. 1992

[45] K.K. Parhi and D.G. Messerschmitt, "*Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding,"* IEEE Trans. Computers, vol. 40, no. 2, pp. 178–195, Feb. 1991

[46] A. Chandrakasan, S. Sheng, and R. Brodersen, *"Low-power CMOS digital design,"* IEEE J. Solid-State Circuits, vol. 27, pp. 473–484, Apr. 1992.

[47] T. C. Denk and K. K. Parhi, "Exhaustive scheduling and retiming of digital signal processing systems," IEEE Trans. Circuits Syst. II, vol. 45, pp. 821–838, July 1998

[48] V. Singhal, C. Pixley, R. Rudell, and R. Brayton, *"The validity of retiming sequential circuits,"* in Proc. 32nd Design Automation Con$, San Fransisco, CA, June 12-16, 1995, pp. 316-321

[49] C.-Y. Wang and K. K. Parhi, *"High-level DSP synthesis using concurrent transformations, scheduling, and allocation,"* IEEE Trans. Computer-Aided Design, vol. 14, pp. 274–295, Mar. 1995

Appendix A

Verilog Code

```
module iir_8(clk,rst,d_in,d_out);
parameter order = 8;
parameter w_in = 8;
parameter w_out = 2*w_in+2;

         //filter coefficients
      parameter b0= 8'd2;
       parameter b1= 8'd5;
        parameter b2= 8'd8;
        parameter b3= 8'd11;
  parameter b4= 8'd14;
  parameter b5= 8'd11;
        parameter b6= 8'd8;
      parameter b7= 8'd5;
       parameter b8= 8'd2;


      parameter a1= 8'd1;
      parameter a2= 8'd1;
      parameter a3= 8'd1;
       parameter a4= 8'd1;
      parameter a5= 8'd1;
       parameter a6= 8'd1;
      parameter a7= 8'd1;
      parameter a8= 8'd1;

      output [w_out-1:0] d_out;
       input [w_in-1:0] d_in;
      input clk,rst;

        reg [17:0]
tap_in1,tap_in2,tap_in3,tap_in4,tap_in5,tap_in6,tap_in7,tap_in8;
       reg [17:0]
tap_out1,tap_out2,tap_out3,tap_out4,tap_out5,tap_out6,tap_out7,tap_out8;
      wire [w_out-1:0] d_forward;
      wire [w_out-1:0] d_back;

 assign d_forward = (b0*d_in) +(b1*tap_in1) +(b2*tap_in2) +(b3*tap_in3)
+(b4*tap_in4) + (b5*tap_in5)+(b6*tap_in6)+(b7*tap_in7)+(b8*tap_in8);
assign d_back = (a1*tap_out1)+ (a2*tap_out2)+ (a3*tap_out3)+
(a4*tap_out4)+ (a5*tap_out5) + (a6*tap_out6)+(a7*tap_out7)+(a8*tap_out8);

       assign d_out = d_forward+d_back;

       always @ (posedge clk)
       begin
       if (rst == 1)

       begin
            tap_in1 <= 0;
            tap_in2 <= 0;
             tap_in3 <= 0;
            tap_in4 <= 0;
            tap_in5 <= 0;
             tap_in6 <= 0;
            tap_in7 <= 0;
             tap_in8 <= 0;
            tap_out1 <= 0;
             tap_out2 <= 0;
            tap_out3 <= 0;
            tap_out4 <= 0;
            tap_out5 <= 0;
             tap_out6 <= 0;
             tap_out7 <= 0;
             tap_out8 <= 0;
             end
            else
             begin
            tap_in1 <= d_in;
            tap_out1 <= d_out;

             tap_in2 <= tap_in1;
            tap_in3 <= tap_in2;
            tap_in4 <= tap_in3;
            tap_in5 <= tap_in4;
            tap_in6 <= tap_in5;
             tap_in7 <= tap_in6;
            tap_in8 <= tap_in7;

            tap_out2 <= tap_out1;
            tap_out3 <= tap_out2;
            tap_out4 <= tap_out3;
             tap_out5 <= tap_out4;
             tap_out6 <= tap_out5;
            tap_out7 <= tap_out6;
            tap_out8 <= tap_out7;

      end
endmodule
```

Testbench

```
      module stim;
            reg [7:0] d_in;
            reg clk,rst;
             wire [17:0] d_out;

             iir_8 f3(clk,rst,d_in,d_out);

             initial
            begin
             rst=1;
            d_in=8'b00000001;
            clk=1;

             #4 rst=0;
            #34 d_in=8'b00000000;
```

```
             #2 $finish;
end
            always
            begin
            #2 clk=~clk;
end
endmodule
```

2-slow IIR design

Verilog Code

```
module slow_iir(clk_sub,clk_main,rst,impulse,unitstep,d_out);
parameter order = 8;
    parameter w_in = 8;
    parameter w_out = 2*w_in+2;

    //filter coefficients
    parameter b0= 8'd2;
    parameter b1= 8'd5;
    parameter b2= 8'd8;
    parameter b3= 8'd11;
    parameter b4= 8'd14;
    parameter b5= 8'd11;
    parameter b6= 8'd8;
    parameter b7= 8'd5;
    parameter b8= 8'd2;


    parameter a1= 8'd1;
    parameter a2= 8'd1;
    parameter a3= 8'd1;
    parameter a4= 8'd1;
    parameter a5= 8'd1;
    parameter a6= 8'd1;
    parameter a7= 8'd1;
    parameter a8= 8'd1;

    output [w_out-1:0] d_out;
    input [w_in-1:0] impulse,unitstep;
    input clk_main,clk_sub,rst;
   // reg [w_out-1:0] d_out;
   reg [w_in-1:0] d_in;

    reg [17:0]
tap_in1,tap_in2,tap_in3,tap_in4,tap_in5,tap_in6,tap_in7,tap_in8;
    reg [17:0]
temp_in1,temp_in2,temp_in3,temp_in4,temp_in5,temp_in6,temp_in7,temp_in8;
    reg [17:0]
tap_out1,tap_out2,tap_out3,tap_out4,tap_out5,tap_out6,tap_out7,tap_out8;
    reg [17:0]
temp_out1,temp_out2,temp_out3,temp_out4,temp_out5,temp_out6,temp_out7,tem
p_out8;
    wire [w_out-1:0] d_forward;
    wire [w_out-1:0] d_back;

    //integer k;
    assign d_forward =
(b0*d_in)+(b1*tap_in1)+(b2*tap_in2)+(b3*tap_in3)+(b4*tap_in4)+(b5*tap_in5
)+(b6*tap_in6)+(b7*tap_in7)+(b8*tap_in8);
    //assign d_forward =
(b0*d_in)+(b1*tap_in1)+(b2*tap_in2)(b3*tap_in3)+(b4*tap_in4)+(b5*tap_in5)
+(b6*tap_in6)+(b7*tap_in7)+(b8*tap_in8);
     assign d_back =
(a1*tap_out1)+(a2*tap_out2)+(a3*tap_out3)+(a4*tap_out4)+(a5*tap_out5)+(a6
*tap_out6)+(a7*tap_out7)+(a8*tap_out8);

     assign d_out = d_forward+d_back;

    always @ (posedge clk_sub)
    begin
      if (rst == 1)
      //for (k = 1; k <= order; k = k+1)
      begin
       tap_in1 <= 0;
       tap_in2 <= 0;
       tap_in3 <= 0;
       tap_in4 <= 0;
       tap_in5 <= 0;
       tap_in6 <= 0;
       tap_in7 <= 0;
       tap_in8 <= 0;
       tap_out1 <= 0;
       tap_out2 <= 0;
       tap_out3 <= 0;
       tap_out4 <= 0;
       tap_out5 <= 0;
       tap_out6 <= 0;
       tap_out7 <= 0;
       tap_out8 <= 0;
       temp_in1 <= 0;
       temp_in2 <= 0;
       temp_in3 <= 0;
       temp_in4 <= 0;
       temp_in5 <= 0;
       temp_in6 <= 0;
       temp_in7 <= 0;
       temp_in8 <= 0;
       temp_out1 <= 0;
       temp_out2 <= 0;
       temp_out3 <= 0;
       temp_out4 <= 0;
       temp_out5 <= 0;
       temp_out6 <= 0;
       temp_out7 <= 0;
       temp_out8 <= 0;
      // d_out <= 0;
       end
      else
        begin
          tap_in1 <= d_in;
          tap_out1 <= d_out;

       tap_in1 <= temp_in1;
```

```
        temp_in2 <= tap_in1;
        tap_in2 <= temp_in2;
        temp_in3 <= tap_in2;
        tap_in3 <= temp_in3;
        temp_in4 <= tap_in3;
        tap_in4 <= temp_in4;
        temp_in5 <= tap_in4;
        tap_in5 <= temp_in5;
        temp_in6 <= tap_in5;
        tap_in6 <= temp_in6;
        temp_in7 <= tap_in6;
        tap_in7 <= temp_in7;
        temp_in8 <= tap_in7;
        tap_in8 <= temp_in8;


       tap_out1 <= temp_out1;
        temp_out2 <= tap_out1;
        tap_out2 <= temp_out2;
        temp_out3 <= tap_out2;
        tap_out3 <= temp_out3;
        temp_out4 <= tap_out3;
        tap_out4 <= temp_out4;
        temp_out5 <= tap_out4;
        tap_out5 <= temp_out5;
        temp_out6 <= tap_out5;
        tap_out6 <= temp_out6;
        temp_out7 <= tap_out6;
        tap_out7 <= temp_out7;
        temp_out8 <= tap_out7;
        tap_out8 <= temp_out8;

          temp_in1 <= d_in;
          temp_out1 <= d_out;
       // d_out <= d_forward+d_back;
       end
       if (clk_main==0)
        begin
        d_in <= impulse;
       // d_out1=d_out;
    end
        else
        begin
        d_in <= unitstep;
        //d_out2 = d_out;
    end


  end
endmodule
```

Testbench

```
      module stimnew;
      reg [7:0] impulse,unitstep;
      reg clk_sub,clk_main,rst;
       wire [17:0] d_out;
      slow_iir cslow(clk_sub,clk_main,rst,impulse,unitstep,d_out);

       initial
      begin
        rst=1;
        impulse=8'b00000001;
        unitstep=8'b00000001;
        clk_sub=1;
        clk_main=1;

       #2 rst=0;
       #2 impulse=8'b00000000;
#36 $finish;
end
    always
begin
#1 clk_sub=~clk_sub;
end
always
begin
#2 clk_main=~clk_main;
end
endmodule
```

Synthesis Results

Table A-1 Device Utilization Summary

	IIR filte	2-slow IIR filter	3-Slow IIR filter	4-slow IIR filter	
Logic Utilization	Used	Used			Available
Number of Slice Flip Flops	208	424	634	717	26,624
Number of 4 input LUTs	315	323	326	453	26,624
Logic Distribution					
Number of occupied Slices	285	393	500	575	13,312
Number of Slices containing only related logic	285	393	500	575	285
Number of Slices containing unrelated logic	0	0	0	0	285
Total Number of 4 input LUTs	317	325	328	455	26,624
Number used as logic	315	323	326	389	
Number used as a route-thru	2	2	2	2	
Number of bonded IOBs	28	37	36	36	221

Number of MULT18X18s	5	5	5	5	32
Number of GCLKs	1	1	1	1	8

Table A-2 Timing Summary

	IIR filter without C-slow	**2-slow IIR filter**
Number of clock cycles required to process two tasks	18	9
Minimum period	22.384ns	22.384ns
Maximum Frequency	44.674MHz	44.674MHz
Minimum input arrival time before clock	19.747ns	5.900ns
Maximum output required time after clock	28.702ns	28.702ns

Appendix B

Micro Program Instruction Set

Address		Symbolic Instruction
0		pc ← 0
1	Fetch	MAR ← pc
2		IR ← M(MAR) ; pc ← pc+1
3	Decode	I3 = 1? go to MEMREF
4		XC0 = 1? Go to CMA
5		XC1 = 1? Go to INCA
6		XC2 = 1? Go to DCA
7		go to HALT
8	CMA	A ← $\bar{A}$
9		go to Fetch
10	INCA	A ← A+1
11		go to Fetch
12	DCRA	A ← A-1
13		go to Fetch
14	MEMREF	if XC0 = 1, LDSTO
15		if XC1 = 1, ADDSUB
16		if XC2 = 1, JUMP
17	AND	MAR ← pc
18		Buffer ← M(MAR), pc ← pc+1
19		MAR ← Buffer
20		Buffer ← M(MAR)
21		A ← A & Buffer
22		go to Fetch
23	LDSTO	MAR ← pc
24		Buffer ← M(MAR); pc ← pc +1
25		MAR ← Buffer
26		if I0 = 1 go to STO
27	LOAD	Buffer ← M(MAR)
28		A ← Buffer
29		go to Fetch
30	STO	M(MAR) ← A
31		go to Fetch
32	ADSUB	MAR ← pc
33		Buffer ← M(MAR); pc ← pc+1
34		MAR ← Buffer

35		Buffer ← M(MAR)
36		if I0 = , go to SUB
37	ADD	A ← A + Buffer
38		go to Fetch
39	SUB	A ← A- Buffer
40		go to Fetch
41	JUMP	MAR ← pc
42		if I0 =0, go to JOZ
43		if I0 =1, go to JOC
44	JOZ	if z=1 go to LOADPC
45		pc ← pc+1
46		go to Fetch
47	JOC	if c=1 go to LOADPC
48		pc ← pc+1
49		go to Fetch
50	LOADPC	pc ← M(MAR)
51		go to Fetch
52	HALT	go to HALT