A LOW POWER DESIGN FOR ARITHMETIC AND LOGIC UNIT

NG KAR SIN
(B.Tech. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF

ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2004

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to all those who have directly or indirectly provided advice and assistance during the course of my research in the NUS.

Assoc. Prof. Tay Teng Tiow (NUS), who has led me to the proposal of this project. He has provided invaluable guidance, suggestions and support throughout the course of research. During times of difficulties, he has also shown much understanding and patience, which makes this course a memorable part of my life.

Mr Zhu Xiao Ping and Mr Pan Yan, for their times in several constructive discussions over technical and academic problems. These discussions often helped to clarify questions that are related to the research interest.

Miss Rose Seah and Mr Teo King Hock, for their prompt logistic support in the lab, which provided me a conducive environment to work in the lab.

TABLE OF CONTENTS

SUMMARY

The rise of portable devices with wireless network connections has lead to demands on microprocessors to deliver high performance and yet consume low power. This project works on a design for a single-issue 32-bit integer pipelined ALU that comprises two kinds of functional units: one with fast performance and high power consumption and another with slow performance and low power consumption. Both are used to execute instructions, but slow functional units are used whenever possible, for the reason of reducing power consumption.

The ALU architecture comprises a Control Unit, Register File and the mentioned functional units. To make use of this architecture effectively, an offline software instruction scheduler is used to identify and create specific situations for the slow functional unit to be used. The specific situations occur when:

1. there are no subsequent instructions depending on the current instruction;
2. the current instruction has been scheduled for advanced execution;
3. the dependent subsequent instructions are scheduled for a later execution.

When the above situations are identified, slow functional units are used to execute instructions.

However, using two functional units with different levels of performance can cause instruction execution to be in-orderly issued but out-of-orderly executed. As such, instruction execution and retirement have to be properly synchronized to ensure that registers write-backs are performed correctly. This can be achieved by using the

Control Unit to synchronize all instruction issues and executions, and updating the Register File at appropriate timings.

The software instruction scheduler mentioned earlier analyzes and rearranges PIns in the programs, resulting in specific situations being identified or created so that slow functional units are used. After analyzing and rearranging the PIns, the scheduler generates two types of directives for the assembler to work with. The first type of directives indicates selected PIns that can be executed with slow functional units. The assembler uses these directives to compile selected PIns with MIns that are executed with the specified slow functional units. The second type of directives indicates stalls in the pipeline caused by unresolvable instruction dependencies. The assembler uses these directives to embed stall information into opcodes, so that the ALU can delay instruction issue appropriately. In this way, delay instructions such as “NOP” are avoided and the power consumed by fetching and executing such instructions is saved.

Therefore, our proposed ALU consumes power for instruction executions only at run time, since there is no other real time activity happening during operation. Hence, it is therefore capable of attaining low power.

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

C_L	Load Capacitance
ΔV	Voltage Change
V_{DD}	Supply Voltage
f_{clk}	Clock Frequency
α	Activity Factor
F_{0-1}	Low-to-High Transitions
V_{Tn}	Threshold Voltage of NMOS
V_{Tp}	Threshold Voltage of PMOS
$T_{\text{Worst Rise}}$	Worst Rise Time
$T_{\text{Worst Fall}}$	Worst Fall Time

CHAPTER 1

INTRODUCTION

This chapter is divided into four sections: 1.1 Background, 1.2 Related Work, 1.3 Project Proposal, 1.4 Project Overview and 1.5 Project Scope.

1.1 Background

Portable devices with wireless network connections such as Personal Digital Assistants (PDA), cellular phones and Global Positioning System (GPS) navigators have become increasing popular and widely-used over the past few years. One reason for the widespread adoption is their usability such as a transformation to a graphical interface. The ability for such a transformation has much to do with the high performance microprocessors embedded in them. Not only are the microprocessors expected to execute complicated functions, but they also should sustain reasonably long usage times giving rise to a need for low power consumption. This explains why a lot of research effort and technological developments centre on building microprocessors that can deliver high performance and yet consume minimal power.

In this preceding chapter, we will explore briefly some techniques that have been developed to reduce power consumption in microprocessors. A general understanding of the technological development on this front will foster a clearer understanding of the project's objectives and where our ALU design stands in comparison with the techniques of reducing power consumption in microprocessors.

1.2 Related Work

Research on low power microprocessors has mainly been concerned with reducing power consumption while maintaining optimum performance levels. There are different techniques of reducing power consumption in microprocessors. Primarily, it is done either by lowering the supply voltage through hardware in conjunction with software support (e.g. Dynamic Voltage Scaling), or by reducing switching activities during runtime operations with an offline software support (e.g. offline intelligent compiler).

The power consumption of a microprocessor is directly proportional to the level of its performance, so the higher its level of performance, the more power the microprocessor consumes and vice versa (full details of microprocessor power consumption are described in Section 3.1). The technology that has been developed to reduce power consumption in a microprocessor works mainly around this relationship.

One problem arises when supply voltage is lowered to reduce power consumption in the microprocessor; the digital circuits in the microprocessor become more susceptible to noise. In order to ensure the proper function of circuits, the decrease of supply voltage has to be concurrent with lowering the clock frequency [1]. However, performance must not be compromised when clock frequency is reduced.

The Dynamic Voltage Scaling (DVS), is an example of a previously developed technique which meets this requirement. The DVS technique enables optimum performance in a microprocessor, even when supply voltage is lowered to reduce its power consumption [2, 3]. With this technique, a hardware voltage scheduler controls the supply voltage based on data from a feedback register, while clock frequency is regulated with a voltage-controlled oscillator that tunes the frequency as the supply voltage varies. It is this aspect of the technique that ensures the digital circuits function accurately and performance maintain optimally.

Software support for DVS is in the form of a real time process running on the Operating System, which updates data stored in the feedback register**.** This real time process monitors the microprocessor performance and computational load based on slack analysis [4, 5, 6, 7]. Depending on the rise or fall of values recorded on the feedback register, the level of computational demand is adjusted accordingly.

An alternative to a real time process is an offline intelligent compiler, which is another form of software support [8, 9, 10]. It is used to identify program regions where application of voltage scaling is required during compilation. The compiler embeds directives into instructions to update the feedback register during runtime operation. Data stored in the feedback register in turn communicates the level of performance required to meet computation demands to the microprocessor. As with the DVS technique, supply voltage and clock frequency is tuned as data is updated, so the microprocessor’s optimum performance is maintained while reducing power consumption.

Microprocessors designed for portable devices are capable of decreasing supply voltage to reduce power consumption. Some examples of these microprocessors are the ARM11 series and IBM 405LP for portable handheld devices and the Intel Centrino and TransMeta Crusoe series for laptops and notebook personal computers.

In these microprocessors, power consumption reduction also lies in the design of their functional circuits. The functional circuits built into these microprocessors have been specially designed for performance while consuming minimal power. This is evident in the analysis of the circuits' datapath, which reveals how switching activities in these functional circuits have been optimized for low power consumption [11]. Intentionally designed for frequently-used functions like addition [12, 13, 14, 15] and multiplication [16, 17, 18], the circuits are implemented with CMOS logic due to its low power consumption. These two design features of the functional circuits thereby result in switching activities with low power consumption. *More on CMOS logic is described in Section 3.1.*

Software also has a key role in reducing the power consumption of microprocessors. An offline software that is able to analyze programs and rearrange instructions can cut down microprocessor activities like memory accesses and signal switching within circuits to maintain low power consumption [19]. In the case of VLIW based microprocessors, software is commonly used to perform loop unrolling, software cache prefetch and software pipelining on instructions, which reduces pipeline stalls and improves performance of the microprocessor. Drawing on the same approach, software can reduce power consumption by expressly reducing the amount of memory accesses for data fetch [20]. The use of software can also reduce switching activities by rearranging instructions based on Hamming distance [8] and power consumed between instruction transitions [21, 22].

1.3 Project Proposal

While lowering supply voltage and decreasing the frequency of switching activities are prevalent techniques of reducing power consumption in microprocessors, they also have several disadvantages.

First, while supply voltage reduction effectively lowers power consumption, its application is limited to the functional units in the microprocessor circuits. Moreover, the voltage-reduced circuits require additional interfacing circuits to connect them to other circuits that work with different supply voltages.

Second, with voltage reduction during real time operation, the Operating System is required to update the voltage reduction mechanism frequently. Not only does this eat into overheads required by the microprocessor to compute the real time slacks during runtime, it also consumes extra energy to deliver the computations. On the other hand, offline optimization software activities are performed only during the compilation stage on development machines, and no overheads are incurred during runtime.

The project proposes a design for low power consumption ALU that exploits the benefits of offline software, which can work alone in delivering minimum power consumption or work alongside supply voltage reduction technology to deliver even lower power consumption. Our ALU architecture consists of a set of fast and slow functional units. Fast functional units deliver high performance, but consume a considerable amount of power as they use parallel circuits to carry out computations. Slow functional units on the other hand use simpler circuits to perform computations and consume less energy, but take a longer time to complete the computations.

An instruction scheduler was developed to analyze and rearrange instructions to execute with slow functional units before opcode assembly. The instruction scheduler generates directives for the assembler to assemble opcodes executed with slow functional units during runtime, a feature not available in other microprocessors in the market.

There are many advantages and plus points to the design of our ALU. Not only does it consume minimal power during runtime, it does not require real time process to monitor performance. Neither is a hardware circuit needed to tune the supply voltage. Compared with other models operating on the supply voltage reduction principle, the ALU we have designed is far simpler. This is another boon, because the simplicity in design means voltage reduction techniques can be additionally incorporated into the ALU to further reduce power consumption of the microprocessor.

An overview of the ALU design is described in Section 1.4, with full details on the ALU design is described in Chapter 2.

1.4 Project Overview

This project works on a design for a single-issue 32-bit integer pipelined ALU that comprises two kinds of functional units: one with fast performance and high power consumption and another with slow performance and low power consumption. Both are used to execute instructions, but slow functional units are used whenever possible, for the reason of reducing power consumption. An instruction scheduler is used to identify and create specific situations for the slow functional unit to be used.

It has been observed that in a conventional pipeline, instructions are usually executed with fast functional units. Data is processed as quickly as possible and instructions are passed down without stalling the pipeline. However, there are situations where fast functional units are not required to execute instructions. These situations occur when:

1. there are no subsequent instructions depending on the current instruction;
2. the current instruction has been scheduled for advanced execution;
3. the dependent subsequent instructions are scheduled for a later execution.

When instructions do not require immediate execution, slow functional units can be used to reduce power consumption without incurring loss in performance. This applies to the ALU design, when the above situations are identified.

However, using two functional units with different levels of performance can cause instruction execution to be in orderly issued but out of orderly executed [23, 24]. As such, instruction execution and retirement have to be properly synchronized to ensure that registers write-backs are performed correctly. Figure 1 shows an example of a situation when slow functional units are used to execute instructions with the following code sample. The pipeline stages used in Figure 1 are “F” for fetch, “D” for decode, “E” for execute and “W” for write-back. For instructions that require more than one execution stage, “E*n*” is used to indicate execution and *n* is an integer that indicates the number of executing stage.

	Instructions	**Cycles**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Part A	`Mov  ax, bx`	1	F	D	E	W						
	`Add  ax, bx`	1		F	D	E	W					
	`Push bx`	1			F	D	E	W				
	`And  bx, dx`	1				F	D	E	W			
	`Mov  si, bx`	1					F	D	E	W		
	`Pop  bx`	1						F	D	E	W	
Part B	**Instructions**	**Cycles**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	`Mov  ax, bx`	1	F	D	E	W						
	`Add  ax, bx`	4		F	D	E1	E2	E3	E4	W		
	`Push bx`	1			F	D	E	W				
	`And  bx, dx`	1				F	D	E	W			
	`Mov  si, bx`	1					F	D	E	W		
	`Pop  bx`	1						F	D	E	W	

Fig. 1 Instruction execution with slow functional unit

From Figure 1, Part A shows a conventional pipeline with regular stages for all instruction executions. In Part B, since the “`add`” instruction is not depended subsequently, it can be executed with slow functional units without affecting the performance or correctness of the program execution. Hence, arithmetic instructions like “`add`” in the above example can now be implemented with two functional units of different performance. To the programmer, the instructions appear the same since there is no need to know about the underlying instruction execution process. To the ALU, however, all instructions must be unique so the required functional unit is correctly selected for execution. To distinguish instructions for programmer and ALU, the instructions programmers use will be defined as “Programmer’s Instructions” or “PIns”. Instructions that the ALU executes will be defined as “Machine Instructions” or “MIns”.

The software instruction scheduler mentioned earlier analyzes and rearranges PIns in the programs, resulting in specific situations being identified or created so that slow functional units are used. After analyzing and rearranging the PIns, selected PIns that can be executed with slow functional units are marked with directives. The directives inform the assembler to compile these PIns with MIns that are executed with the specified slow functional units.

Our ALU design is therefore capable of attaining low power consumption during runtime with a software instruction scheduler, with the exclusion of real time activities supporting the operation.

1.5 Scope of Project

The scope of this project is to develop a low power ALU, both hardware and software. The ALU hardware development would focus on the fast and slow functional units, and the software development would focus on the development of algorithms to rearrange instructions to execute with slow functional units to achieve low power consumption.

The performance and power consumption of our ALU depends on the functional unit operations. The main focus of this project would be on hardware research and development. The study of power consumption of arithmetic circuit and behavior is carried out through simulation works. Details of the power consumption of the circuits are described in Appendix I. Different arithmetic circuits are modeled and synthesized with different performance levels to study on the variation in performance and power consumption. With which, the appropriate circuit would be selected to implement the functional unit. Details on the hardware development of the functional circuits and a summary on the selected circuits are described in Chapter 3.

The other section of this project would focus on the development of the software algorithm to achieve lower power consumption on the ALU, which would include the rearrangement of the instructions. Research on software scheduling is also carried out prior to the development work. Using the developed software, several programs are analyzed and reduction on power consumption is estimated. Details of the development work and a summary on the program analysis and power consumption estimation are described in Chapter 4.

1.6 Thesis Organization

The thesis would be organized in the following order.

Chapter 2 describes the runtime operation, hardware design and software instruction scheduler of our low power 32-bit integer ALU. The runtime operation would describe the method used to achieve lower power with the ALU. Components of the ALU would be presented in the hardware design section. The rearrangement of the instructions for the execution in slow functional units would also be described. A novel method to implement the wait state through rearrangement of software instructions would also be included.

Chapter 3 describes the characteristics of CMOS circuits and the implementation of the 32-bit integer ALU functional units. The power consumption and performance of the circuits will be described in this chapter. Results from the simulation would also be presented and discussed.

Chapter 4 presents the instruction scheduling algorithms used to enhance the performance and reduce power consumption during the ALU runtime. The algorithms at each functional stage would be discussed in detail. Results from the program analysis and power consumption estimation would also be presented and discussed.

Chapter 5 summarizes the research and development work and concludes the project. Possible future work and development would also be recommended.

CHAPTER 2

THE ARITHMETIC AND LOGIC UNIT DESIGN

In this chapter, we describe the runtime operation, hardware design and software instruction scheduler of our low power 32-bit integer ALU, explaining how lower power consumption is achieved during the runtime operation. In addition, we will illustrate how instructions are rearranged for the execution in slow functional units and how to implement wait state using embedded information in instructions.

Components of the ALU will be presented in the hardware design section.

2.1 ALU Design

Unlike a typical ALU which uses only one type of functional unit to execute a particular PIn, this ALU is capable of using either a fast or a slow functional unit to execute the PIn, depending on the situation. Figure 2.1 shows the ALU hardware architecture.

Given the same clock frequency in performing similar functions, the fast functional unit completes the operation in a shorter time than the slow functional circuit, because it has more logic circuits. However, while it is faster, the fast functional unit also consumes more power during the operation compared with the slow functional unit, which takes a longer time for the same operation, but consumes less power.

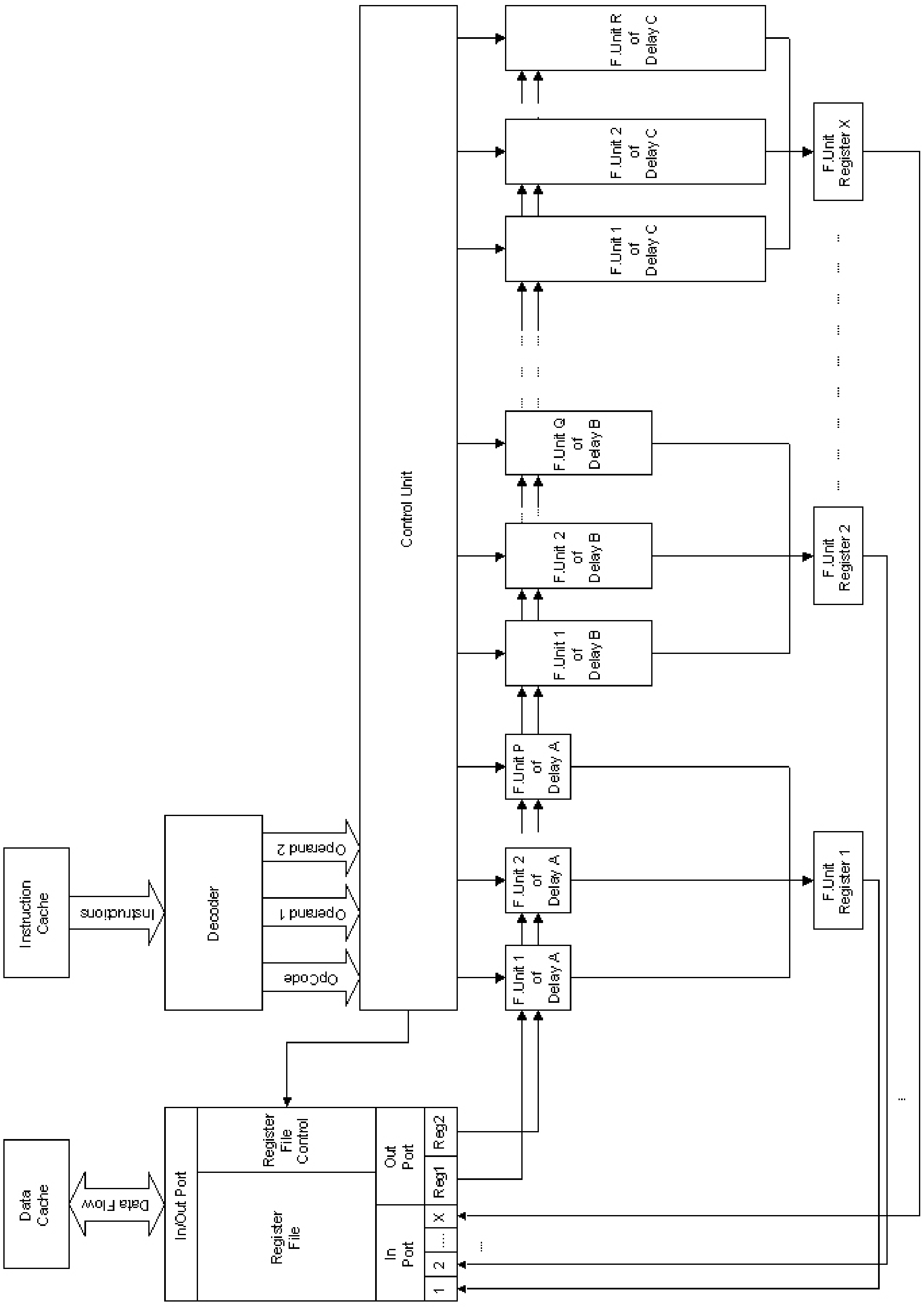


Fig. 2.1 ALU Architecture

The amount of time a functional unit takes to perform an operation is specified in term of number of clock cycles. Different functional units require a different number of clock cycles to perform their operations. As such, the PIns are issued in order from the Control Unit but may be completed in a different order.

With our ALU design structure, a software instruction scheduler analyses an input program and selects a suitable functional unit to perform the PIns. This differentiating feature in the structure of our ALU ensures power-efficient runtime without causing loss in performance.

In processors that use the conventional ALU, PIns are compiled into MIns by an assembler, with one MIn mapped to one PIn. When the proposed ALU is employed in processors, PIns may be realized with different MIns, which in turn trigger different functional units to perform the PIns.

The task of mapping of MIns to PIns for this proposed ALU is achieved with a software instruction scheduler. The scheduler analyzes the independence of PIns in the program and performs the mapping based on performance or power consumption criteria. The ultimate objective is to sustain optimal performance in the microprocessor while consuming minimal power. Optimal performance in achieved when there are no stalls in the pipeline during runtime while low power consumption is attained when slow functional units are used to execute PIns for most of the operations.

Before the scheduler performs its task, the PIns are analyzed and divided into segments, based on the control flow of the programs. Control PIns are used to mark the start and end points of segments. Within the segments, the PIns are reordered to ensure that the control flow of the PIns is correct after reordering. The objective of reordering the PIns is to work around constraints due to dependencies in PIns to enhance performance and reduce power consumption at runtime. After the scheduler has worked on the PIns, a list of directives is generated for the assembler to map MIns to PIns with the appropriate functional units.

The function of the hardware components and software scheduler are described in the following sections.

2.2 Hardware Components

The hardware architecture is designed to be lean and simple. It consists of a Decode and Control Unit, Register File and several functional units of different performance levels. With this architecture, power is consumed during the operation of the Decode and Control Unit for MIns issue, Register File write-backs and when functional units are enabled by the Control Unit for MIns execution. The components and their functions are described as follows.

2.2.1 Decode and Control Unit

The Decode Unit is responsible for fetching and interpreting MIns from the memory system before passing them on to the Control Unit. The Control Unit is designed to be a simple state machine that synchronizes the ALU activities like any other Control Unit in conventional microprocessors. It is responsible for issuing the MIns for execution and synchronizing register write-back for MIns that are orderly issued, but are executed out of order, because functional units of different performance levels are used.

At every clock cycle during runtime, the Decode Unit reads the MIns and relays relevant information like register operands and the functional unit required to the Control Unit. The Control Unit in turn triggers the appropriate functional unit, selects the required registers in the Register File and places the register contents on the input bus of the functional units. When MIns are executed with functional units requiring more than one clock cycle, the following happens: the Control Unit synchronizes MIns executions and register write-backs between the functional units and Register File. It does this by deferring write-backs for the number of clock cycles that the functional units require to run.

For the unused functional units, the clock signal is gated off. These functional units are thus in static state. However, because CMOS circuits are used in the functional units, static power consumption is negligible. An analysis of CMOS circuit power consumption is described in Appendix I.

2.2.2 Functional Units

The functional units are circuit blocks that operated on integer data stored in the Register File. The Control Unit selects the registers and the stored data for the functional units to perform the operations for a particular MIn.

As shown in Figure 2.1, the functional units are organized such that units requiring the same amount of time (in terms of number of clock cycles) to perform their operations are grouped together. In a conventional ALU, each functional unit has a register to store the processed data. However, with the proposed ALU, each group of functional units shares a register to store processed data. Therefore, there are fewer registers required in the ALU to support the functional units. Registers used to store processed data for a group of functional units are called the Common Output Registers.

Even though there is only one Common Output Register available to several functional units within a group, conflicts would not arise when the functional units attempt to write to this register, as the Control Unit issues only one instruction every clock cycle. The workings of the functional unit circuits are described in Chapter 3.

2.2.3 Register File

The Register File control reads selected registers and places the contents on the functional units' input bus. The Control Unit in turn issues instructions and updates selected registers with the content in the Common Output Registers.

The Register File comprises these components:

1. Registers that are available to the programmers,
2. An in-port for updating the registers,
3. An out-port for placing selected register contents on the functional units' input bus,

4. And control circuits that select registers for reading or writing via control signals from the Control Unit.

The Register File is designed to perform multiple register writes within a clock cycle. Because functional units of different performance levels are used, MIns may be orderly issued but may be completed out of order. And when MIns are completed out of order, this allows for several MIns to be concurrently executed within a clock cycle. As such, the Register File must be able to perform multiple register write-backs within a clock cycle, so that the executed MIns are properly retired.

Figure 2.2 illustrates an example of such situations in a pipeline:
Part A shows a regular 4-stage pipeline where only one instruction retires in every clock cycle.

Part B and C show pipeline cases with functional units with operation time that is longer than 1 clock cycle. In Part B, the pipeline has execution stages that vary between 1 to 2 clock cycles. It is observed that for the worst case, there were 2 instructions retiring within a clock cycle. In Part C, the pipeline has execution stages that vary between 1 to 3 clock cycles. In the worst case scenario observed, 3 instructions retired within a clock cycle.

In general, we observed that in functional units requiring different lengths of operation time (measured in number of clock cycles), the maximum number of instructions that retire simultaneously within a clock cycle, *n*, is equal to the operation time (measured in number of clock cycles) of the slowest functional unit.

When a worst-case situation like this occurs, all the Common Output Registers in the ALU will be updated with the processed data from the functional units. The Register File must also update *n* registers respectively within that clock cycle.

	Cycles	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	1	F	D	E	W							
	1		F	D	E	W						
Part A	1			F	D	E	W					
	1				F	D	E	W				
	1					F	D	E	W			
	1	F	D	E	W							
	1		F	D	E	W						
	2			F	D	E1	E2	W				
Part B	2				F	D	E1	E2	W			
	1					F	D	E	W			
	1						F	D	E	W		
	1							F	D	E	W	
	1								F	D	E	W
	1	F	D	E	W							
	1		F	D	E	W						
	2			F	D	E1	E2	W				
Part C	3				F	D	E1	E2	E3	W		
	3					F	D	E1	E2	E3	W	
	2						F	D	E1	E2	W	
	1							F	D	E	W	
	1								F	D	E	W

Fig. 2.2 MIns concurrent retirement

Multiple writes within the Register File may be implemented using multiple ports for the registers [26] or multiple banks of registers [27]. However, multiple writes within the Register File can be simpler using one port and bus for the registers, by implementing very fast writes in sequence.

For example, if one register-to-register write operation requires 3ns to perform, then a maximum of three registers can be updated sequentially within a clock cycle of 10ns with a bus in the Register File. If the registers are implemented with two ports, six registers can be updated within the same write operation time and clock cycle.

2.3 Software Instruction Scheduler

In conventional ALU, hardware circuits like Reservation Stations and Scoreboard Logics [28] are used during runtime to maintain peak performance, while the Dynamic Voltage Scaling [29] system is used to reduce power consumption. The proposed ALU system, however, does not employ these complicated hardware circuits. In place of these, is an offline software instruction scheduler.

The scheduler's objective is to ensure that PIns are rearranged offline to use the slow functional units that consume low power, without suffering any penalty in performance. A list of directives is generated by the scheduler to map PIns with appropriate MIns, as seen in the scheduling results.

Before the scheduler works on the PIns, the PIns pass through a conditioning phase in preparation for the scheduling. During this phase, empty lines and comments are removed from the PIns and they are segmented based on the control flow of the programs. Control PIns mark the start and end points of the segments. Within segments, the PIns are reordered to ensure that the control flow of the PIns is correct after reordering. After segmentation, the PIns are translated into a generic form that the scheduler recognizes.

The scheduler works on the PIns in two phases. In the first phase, the scheduler removes data hazards among the PIns that may stall the pipeline. It does this by analyzing data dependencies among the PIns. When data dependencies are found, the PIns are reordered with the assumption that all functional units require only one clock cycle to execute. This ensures that the PIns are pre-scheduled for optimal performance, before the scheduler proceeds to work, under power-efficient conditions.

In the second phase, the scheduler reanalyzes the pre-scheduled PIns to correct the assumption in first phase. The pre-scheduled PIns are reordered again using the correct number of clock cycles that the functional units required. With this step – analyzing dependencies and reordering the PIns – in place, the scheduler creates or identifies the situations mentioned in Section 1.3, to ensure that slow functional units are used.

When any of the mentioned situations are either found or created, directives will be generated with the scheduling results to provide information for the assembler. The implementation of the software instruction scheduler is described in Chapter 4.

2.3.1 Avoiding Hazards with Wait States

Wait states are still required on occasion to resolve pipeline hazards – even though the scheduler is mainly responsible for this task, which it achieves by reordering the PIns. These exceptions occur when the PIns happen to depend closely on each other, or when there are insufficient independent instructions available for reordering to avoid pipeline hazards. An example of a PIn commonly used in such situations, is the "NOP", which is found in Intel processors.

The "NOP" is technically an empty instruction as nothing is accomplished with its execution. But like other instructions, it is processed as per normal – fetched from memory, decoded and issued by the Control Unit and executed as "`XCHG AX, AX`", as in the case of Intel processors. As such, power [30] is still consumed in the process of fetch, decode, issue and execution of the "`NOP`" PIn.

An alternative method of resolving pipeline hazards, without incurring power consumption, is to implement the delay without explicitly using the "`NOP`" instruction.

Under the assumption that there are available unused bits in the MIns, the scheduler will generate delay directives for the assembler – when the scheduler detects un-resolvable pipeline hazards in the PIns. Upon receiving the delay directives, the assembler embeds delay information [31] into MIns for the stalled PIns.

After the Decode Unit deciphers this delay information, it relays signals to the Control Unit to cease issuing MIns for the required number of clock cycles as indicated by the delay information.

This achieves the effect of using the "`NOP`" instruction in the implementation of wait states, without incurring power for fetching, decoding and executing it.

2.4 Chapter Summary

The components used in the design of the proposed ALU differentiate it from conventional ALU. Conventional ALU use hardware circuits like Reservation

Stations and Scoreboard Logics [28] to sustain peak performance during runtime and Dynamic Voltage Scaling to reduce power consumption.

With the proposed ALU design, both fast and slow functional units are used to execute MIns, along with a Control Unit and a Register File to support simultaneous retirement of instructions during runtime operation.

To achieve low power consumption, PIns are arranged to use slow functional units for execution of PIns, without affecting performance. In place of hardware circuits, a software instruction scheduler is developed to analyze and rearrange PIns to be executed with slow functional units.

The analysis by the software instruction scheduler will reveal how closely dependent the PIns are on each other, and whether wait states are necessary to resolve dependencies. Should delays be required, the necessary information will be embedded in the MIns, and subsequently be decoded by the Control Unit. As such, delay PIns like “NOP” that consume unnecessary power are avoided.

These components in the proposed ALU design differentiate it from conventional ALU, enabling it to sustain optimal performance at low power consumption.

CHAPTER 3

THE ARITHMETIC AND LOGIC UNIT HARDWARE

In this chapter, we will describe the characteristics of CMOS circuits and the implementation of the 32-bit integer ALU functional units. We will also discuss the results of the simulations conducted. Specifically, we will talk about the power consumption and performance of the circuits

3.1 CMOS Circuits

The functional units used in the ALU are implemented with CMOS circuits, which are widely used in low power consumption designs [32]. In the following sections, we will briefly describe the characteristics of CMOS circuits as well as their power consumption behaviour.

3.1.1 Circuit Design

3.1.1.1 CMOS Logic

CMOS circuits use both N-type and P-type MOSFETs (Metal Oxide Semiconductor Field Effect Transistors) to realize logic functions. Figure 3.1 shows some basic circuits for CMOS and Pass transistor logic.

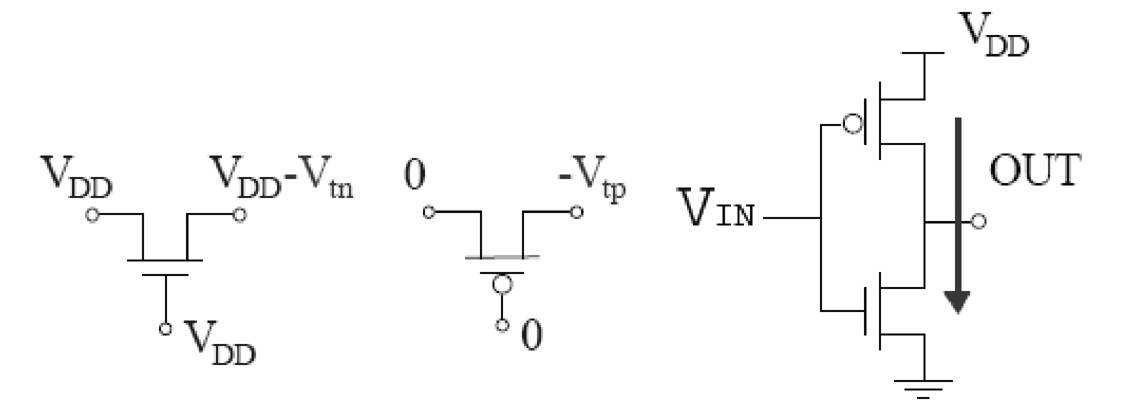


Fig. 3.1 Pass transistor (left and center) and CMOS circuit (right)

Pass transistor logic uses either a NMOS or PMOS (see Figure 3.1, left and center circuit) as a switch to gate electrical signals. Input signal is connected to the transistor gate to create a conductive channel to pass the signal that is connected to the source. This caused a threshold voltage drop across the conducted signal and the output logic signal is degraded [33]. Degraded logic signals may cause the subsequent connected circuits to consume static power due to subthreshold conduction (more details is covered in Appendix Section A1.2).

Contrary to pass transistor logic circuits, CMOS circuits (see Figure 3.1, right circuit) generate rail-to-rail output signals. CMOS circuits use NMOS as pull-down and PMOS as pull-up devices in the logic network. With appropriate input signals connected to the transistor gate, the PMOS transistor charge up output load to the supply voltage level and the NMOS transistor discharge the output load to the ground. As such, CMOS circuits do not incur static power consumption as much as the pass transistor logic circuits. This makes CMOS circuits more suitable for low power circuit designs.

3.1.1.2 Circuit Size

Due to both PMOS and NMOS transistors are used to realize digital logic functions, there are usually a large number of transistors in CMOS circuits. In particular, when many transistors are connected serially in the circuit the parasitic capacitance in the signal path increases. In turn, this increases delay the of the output signal. To counter this problem, buffers or inverters are added along the signal path to increase output drive and reduce the delay. However, this further increases the transistor count in the circuits and the circuit size becomes larger.

3.1.1.3 Simulation

Signal delays in CMOS circuits can be accurately simulated with various delay models and equations. The output signal delay of CMOS circuits may be expressed as a function of the intrinsic delay, parasitic capacitance and load capacitance. The intrinsic delay is determined by parameters in the transistor fabrication process as well as operating conditions. The load capacitance is dependant on the circuit design, while the parasitic capacitance is the sum of the gate capacitance of other connected transistors. In addition to signal delays, power consumption can also be accurately simulated with models and equations.

3.1.2 Power Consumption

There are three types of power consumption in CMOS circuits: dynamic switching power, short circuit power and leakage current power.

Dynamic switching power occurs when load and parasitic capacitances in the circuit are changed or delayed as a result of changes in states. It is the dominant component in CMOS circuit power consumption. Short-circuit current power is energy consumed as a result of the finite turnover time between the rise and fall of input signals. In the third aspect of CMOS circuit power consumption, power is consumed when current leaks through reverse-biased diodes or via sub-threshold conductions.

CMOS circuits have lower power consumption compared with NMOS or bipolar transistor circuits. While NMOS and bipolar junction transistor circuits consume power even when signals are not switching, static (leakage) power consumption for CMOS circuits can be negligible, depending on the channel length of the MOSFETs.

For channel length larger than 0.15um, static power consumption is negligible. For channel length smaller than 0.15um, static power consumption increase exponentially with decreasing channel length. Figure 3.2 shows a simulated plot for static power through an inverter circuit against decreasing channel (gate) length [34].

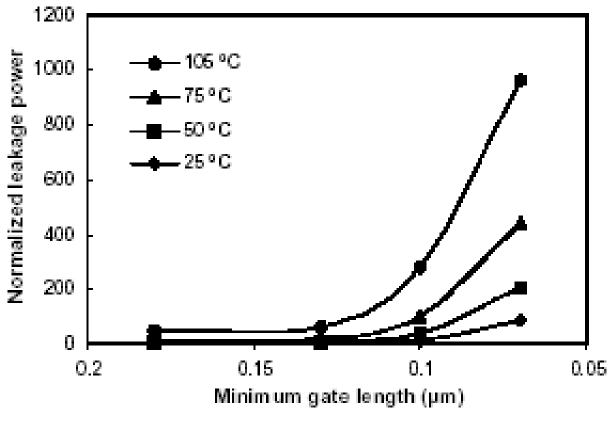


Fig. 3.2 Static (leakage) power against channel (gate) length

Extracted from [34], Figure 1 of "Drowsy caches: simple techniques for reducing leakage power" by Krisztian Flautner et al

When channel length is below 0.15um, the leakage current consists of subthreshold leakage, reverse-bias diode leakage, gate leakage and other smaller leakage components. With such a short channel length, the subthreshold (source/drain) leakage and reverse-bias diode (drain/substrate) leakage current are amplified by the short channel effects and lower threshold voltage respectively [35].

In general cases, the leakage current is dominated by the subthreshold leakage because the depletion layers at the source and drain could be very close to each other due to short gate channel length. However, for advanced technology devices, where gate oxide thickness is very thin (1.8nm or below), gate leakage can dominate the leakage current.

We describe in greater details the three aspects of CMOS circuit power consumption in the following sub sections:

3.1.2.1 Dynamic Switching Power

For every low-to-high output signal transition in the circuits, a voltage change of ΔV occurs across the output load capacitance C_L. To effect this change, energy equivalent to $C_L \Delta V V_{DD}$ joules needs to be drawn from the supply voltage V_{DD}. On the other hand, a high-to-low output signal transition results in the energy stored on C_L to be dissipated into the NMOS transistors and pulls the output low. Figure 3.3 shows the various sources of capacitance seen in an inverter circuit.

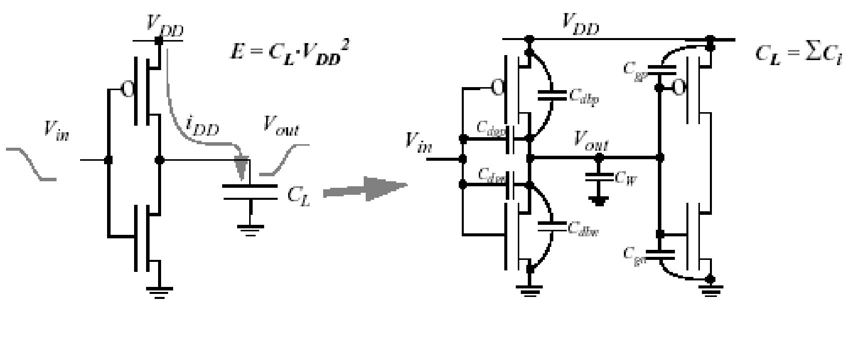


Fig. 3.3 Dynamic switching power consumption; sources of capacitance

Extracted from [1], Figure 2.3 of “Energy-Efficient Processor System Design” by Thomas D. Burd

The basic capacitor elements of C_L shown in Figure 3.3, consists of the gate capacitance of subsequent inputs attached to the inverter output (C_{gp}, C_{gn}), interconnect capacitance (C_W), and the diffusion capacitance on the drains of the inverter transistors (C_{dbp}, C_{dbn}, C_{dgp}, C_{dgn}) [1].

The dynamic switching power consumption is the product of the energy consumed per transition at the rate of low-to-high transitions, $F_{0\text{-}1}$. The value of $F_{0\text{-}1}$ is usually difficult to quantify as it is dependent on the state of the system and the input test vectors. In the absence of a transistor-level circuit simulation, $F_{0\text{-}1}$ can be calculated via statistical analysis of the circuit, or by using a high-level behavioural model with benchmark software to determine a mean value.

Since most digital CMOS circuits are synchronous with a clock frequency f_{clk}; an activity factor, $0 < \alpha < 1$, is used to denote the average fraction of clock cycles in which a low-to-high transition occurs, such that $F_{0-1} = \alpha f_{clk}$. For a circuit with N switching nodes, the dynamic switching power can generally be expressed as,

$$\text{Dynamic Switching Power} = V_{DD} f_{clk} \sum_{i=1}^{N} \alpha_i C_{Li} \Delta V_i \quad \text{.....................(Eq. 1)}$$

From the equation, dynamic switching power may be lowered by reducing V_{DD}. As mentioned in Chapter 1, if V_{DD} is reduced, the operating f_{clk} must be proportionally reduced, as signals in the circuits become more susceptible to noise interference.

3.1.2.2 Short-Circuit Current Power

Short-circuit current power consumption occurs when the output signal of the CMOS circuit is transitioning, while the input signal is still in the middle of transition.

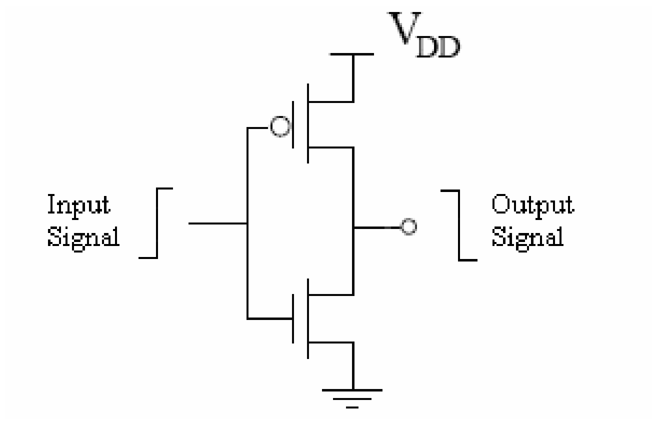


Figure 3.4 Two transistor inverter circuit

In an ideal inverter circuit shown in Figure 3.4, when a step input is given, the PMOS and NMOS transistors should switch states immediately with one turned on and the other turned off. This inhibits the conduction of power from V_{DD} to the ground through the transistors and eliminates short circuit power consumption.

However, in real circuits, parasitic capacitance exists along the signal path. This causes the input signals to have a finite rise and fall time. As long as the conditions $V_{Tn} < V_{in} < V_{DD} - |V_{Tp}|$ and $0 < V_{out} < V_{DD}$ remain in place for the input and output signals, a conductive path will connect V_{DD} to the ground as both PMOS and NMOS transistors are turned on. The slower the rise and fall times of the input signal, the longer the short-circuit current will continue to flow.

Figure 3.5 shows a plot for following signals from a switching inverter circuit shown in Figure 3.4. From the plot, the horizontal axis indicates time and the vertical axis indicate the magnitude of voltage or power for the respective signals.

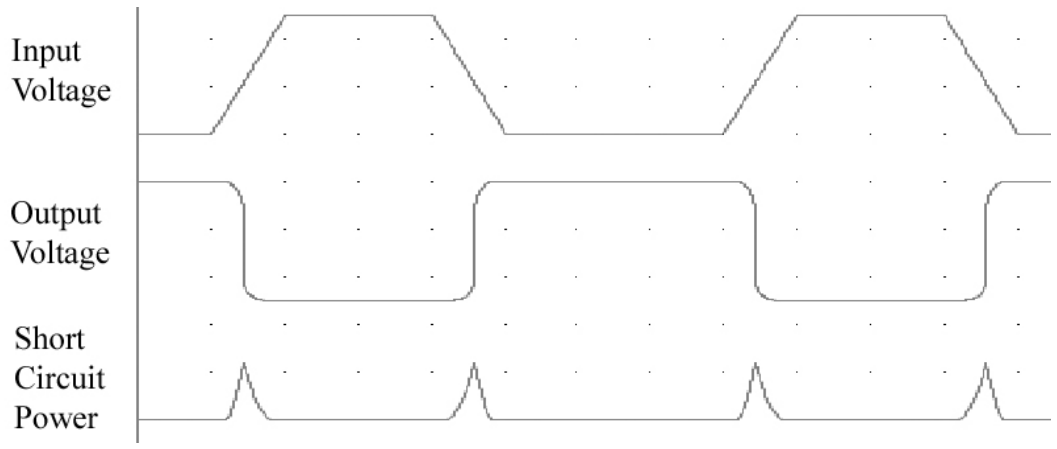


Fig. 3.5 Inverter circuit electrical signals

From Figure 3.5, we can observe short circuit power occurring around every signal transitions.

Short-circuit power consumption scales along with V_{DD}. Theoretically, it can be eliminated if V_{DD} is lowered to the point below the sum of the thresholds of the transistors, $V_{DD} < V_{Tn} + |V_{Tp}|$ because both PMOS and NMOS cannot be turned on at the same time.

3.1.2.3 Leakage Current Power

The current leakages in CMOS circuits are due to the reverse-bias diode leakage and sub-threshold leakage through the channel of a MOSFET that is turned off. The magnitude of these currents is set predominantly by the processing technology and total number of transistors.

Reverse-bias diode leakage

Diode leakage occurs when one transistor is turned off, and another active transistor charges up, or down, the drain with respect to the former's bulk potential. For a static CMOS inverter cross-section shown in Figure 3.6, with a low input voltage, the output voltage will be high because the PMOS transistor is on. The NMOS transistor will be turned off, but its bulk-to-drain voltage will be equal to the supply voltage, $-VDD$. The resulting diode leakage current will be approximately $I_{LD} = A_D.J_{SD}$, where A_D is the area of the drain diffusion, and J_{SD} is the leakage current density of the diffusion, set by the fabrication process technology.

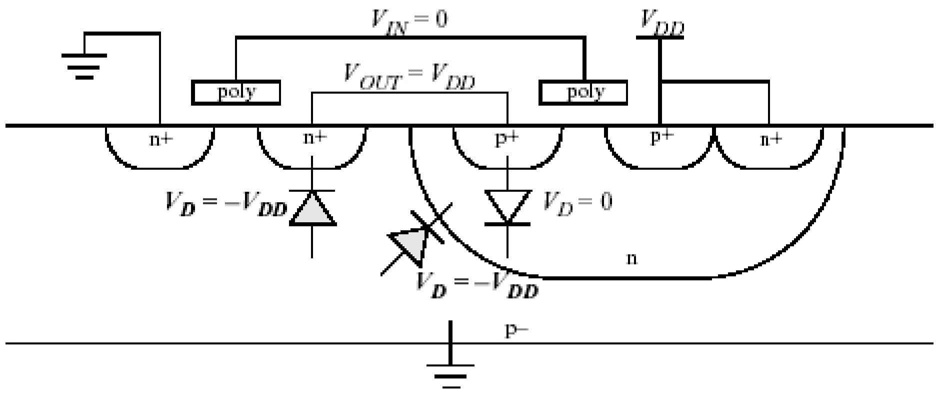


Fig. 3.6 Reverse-bias diodes in CMOS inverter circuit

Extracted from [1], Figure 2.5 of "Energy-Efficient Processor System Design" by Thomas D. Burd

Since the diode reaches maximum reverse-bias current for relatively small reverse-bias potential (< 100mV), the leakage current is roughly independent of supply voltage.

In an nwell process, such as that depicted in Figure 3.6, the nwell-substrate reverse-biased diode also has leakage current. Since a diode leakage current is primarily determined by the more lightly doped side of the junction, which is the p- substrate, the leakage current density is similar to that of the NMOS drain-substrate diode [36]. Because the well area, A_W, is an order of magnitude larger than the diffusion area, this current will dominate reverse-biased diode leakage in an n-well process. The current is $I_{LW} = A_W.J_{SW}$, where J_{SW} is the leakage current density of the well, also set by the technology.

Subthreshold leakage

Subthreshold leakage occurs under similar conditions as the diode leakage. In Figure 3.6, the NMOS was turned off, but even for V_{GS} = 0V, there is still current flowing in the channel due to the V_{DS} potential of V_{DD}.

The magnitude of the subthreshold current is both a function of process, device sizing (W/L), and supply voltage [37]. The process parameter that predominantly affects the current value is V_T. Reducing V_T exponentially increases the subthreshold current, which to first order, is proportional to V_{DS}, or equivalently, V_{DD}.

3.2 Functional Units

3.2.1 Circuit Models

The proposed ALU consists of functional units that perform logic, bit and arithmetic operations. Depending on the design, these functional units are either implemented with either high-performance complex logic that has high-power consumption circuits or simpler low-performance that has low-power consumption circuits.

The circuit design for the functional units may be described with behavioural, structural or hybrid models in Register Transfer Language (RTL) style using Verilog [38]. Behavioural models describe the functions of circuits using synthesizable Verilog function operators, while structural models describe the logical structure of the circuits using logic functions. In another words, the behavioural model can be seen as a high level description of the circuit model, while the structural model is a low level description.

Behavioural model is useful when designs can be described with synthesizable Verilog operators, instead of expressing the designs using primitive logic operators like AND, OR and XOR. In some cases, different parts of the design can be described with behavioural and structural model. Such circuit modelling approach is termed as the hybrid model.

Structural models are used to describe the logic operation circuits, since their design is simple and consists only of logic gates and registers. Behavioural models are used to describe bit operation circuits using the bit operators in Verilog as they are more complicated. The design of arithmetic circuits range from simple circuits to highly complicated logic networks. As such, the behavioural, structural and hybrid models are all used to describe different types of arithmetic circuits.

The performance and power consumption of the synthesized circuits depends on both the circuit design and the technology of the standard cell library. The circuit design determines the complexity of the circuit and in turn the number of logic components (from the standard cell library) required to realize the design. Each logic component has its own performance and power consumption. As such, the performance and power consumption of the synthesized circuit is computed based on the logic components used in the circuit.

3.2.2 Circuit Synthesis

The circuits for the functional units are synthesized with the Synopsis Design Analyzer using the C35 0.35um CMOS standard cell library – the technology available at the time of development.

The Synopsis Design Analyzer has different circuit-optimizing options to fulfil different requirements of performance or power consumption. These options are used along with a set of clock constraints to synthesize the circuit models to obtain circuits of different performance and power consumption level. The set of clock constraints ranged from 100ns up to the maximum performance limit of each model.

Clock constraint defines the amount of time in which the circuit is bound to deliver computation results. The inverse of the clock constraint is the operating clock frequency for the circuit. Thus, as clock constraint is shortened, the synthesized circuits run on a faster operating clock frequency. Depending on the level at which it is set, the circuits are synthesized to a performance point that is sufficient to meet the clock constraint.

Table 3.1 shows a summary on the circuit information obtained from synthesizing the behavioural addition circuit model with a range of clock constraints.

Clock Constraints	**100ns**	**50ns**	**25ns**	**10ns**	**5ns**
Area (um^2)	24156	24156	24156	51541.4	63654.6
Dynamic Power (mW)	0.23	0.46	0.92	4.45	11.21
Normalized Power (mW)	0.23	0.23	0.23	0.44	0.56
Data Arrival (ns)	12	12	12	4.87	3.77

Table 3.1 Synthesis process for behavioural model adder

In Table 3.1, Area indicates the required circuit size on the silicon die. Performance is reflected in the Data Arrival measurements – it indicates the time the circuit takes to deliver computations. Dynamic Power indicates the circuit's power consumption.

From the observations on Table 3.1, circuits synthesized with clock constraints at 100ns, 50ns and 25ns have equal circuit areas and performance levels. This implies that the same circuit is synthesized for all three clock constraints, since this circuit performance meets the requirements. On the other hand, power consumption is observed to increase proportionally as the clock constraints get shorter. For example, the power consumption for 50ns is twice the power consumption for 100ns, while for 25ns it is four times the power consumption for 100ns.

From the datasheets of the logic components [39], power consumption is provided as micro Watts per MHz. This implies that power consumption is proportional to operating clock frequency. Thus, it explains the proportional difference in power consumption between circuits synthesized with clock constraints at 25ns and 100ns.

For clock constraints at 10n and 5ns, different circuits are synthesized to meet the requirement. In general, these circuits have larger size as there are more components used to execute parallel functions to speed up performance, but consume more power during operation.

The power consumption recorded in Table 3.1 is obtained from synthesizing circuits at different clock constraints. In order to provide a fair comparison among different circuits, it should be normalized based on a common clock constraint. Normalized power can be computed as follow,

1) divide the 100ns (common clock constraint) by the applied clock constraint to obtain a power factor,

2) then divide the Dynamic Power by the power factor to obtain the normalized power.

In another words, the normalized power indicates the power consumption for operating the circuit with the common clock constraint at 100ns. The reason for using 100ns as the common clock constraint is the ease of computation and also it is the lowest clock constraint in the range used to synthesize the circuit models.

Although the circuits are being compared based on the common clock constraint, the performance figure for the circuits are still valid. This can be proven from the performance of circuits synthesized with clock constraint at 100ns, 50ns and 25ns - which the performance of the circuits does not change regardless of the clock constraint applied. Hence, we can compare the synthesized circuits based on normalized power and performance, for selecting appropriate circuits to implement the functional units.

3.2.3 Logic and Bit Operation Circuits

The logic circuits are described with simple structural models. The AND, OR, NOT and XOR circuits are synthesized with a clock constraint of 100ns. For the AND, OR and NOT circuits, the propagation delay is 2ns and power consumption is 100uW. The propagation delay for the XOR circuit is 2.5ns and the power consumption is 150uW. As more transistors are used to implement the functions in the XOR circuit, it has a slightly longer delay and higher power consumption level compared with the other circuits.

Due to their structural complexity, bit operation circuits are described with behavioural models. The register shift, rotate and compare function circuits are synthesized with clock constraint of 100ns. For the register shift and rotate function, the propagation delay is 11ns and power consumption, 165uW. The propagation delay for the register compare function circuit is approximately 8ns, with a power consumption of 66uW. The register compare function has smaller power consumption as it only writes the result to a one-bit register, which is usually part of the flag register.

3.2.4 Addition Circuits

The slow and fast adder circuits are obtained by synthesizing the behavioural model adder. With different optimizing options and clock constraints, the behavioural model adder can synthesize anything from a simple Carry Ripple adder circuit to an extremely complicated logic network like the Carry Look Ahead (CLA) adder, which executes very fast additions.

The Carry Ripple adder circuit uses Full Adder cells (shown in Figure 3.7) as circuit building blocks, to form adder circuits (shown in Figure 3.8). For an *n*-bit adder circuit, *n* Full Adder cells are required. Full Adder cells can be expressed with the following equations:

$$Sum_i = A_i \oplus B_i \oplus Carry_{i-1} \quad \text{...(Eq. 2)}$$

$$Carry_{i-1} = A_i B_i + (A_i + B_i)Carry_i \quad \text{.......................................(Eq. 3)}$$

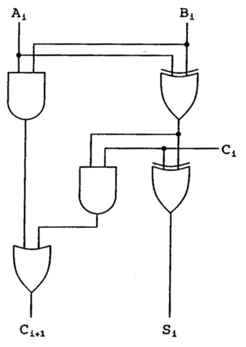


Fig. 3.7 Full Adder cell

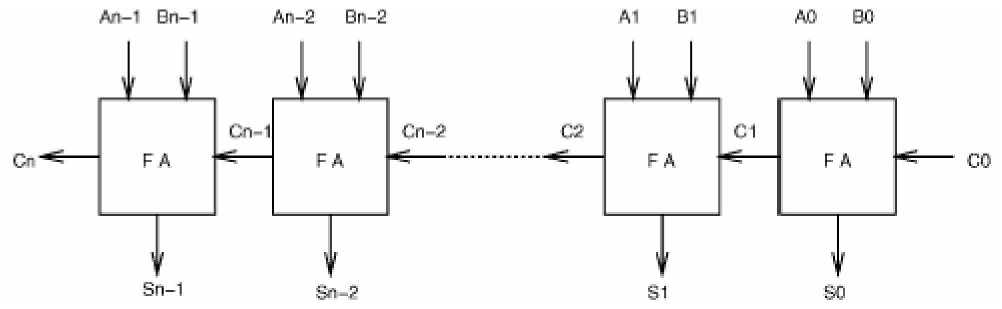


Fig. 3.8 Carry Ripple adder design

The CLA adder uses parallel circuits to generate carry bits for all inputs, instead of propagating the carry signals through the stages of the adder. The carry bits for all inputs are generated based on the following equations:

$Sum,\ S_i = A_i \oplus B_i \oplus Carry_{i-1}$ ……………….………………..…………(Eq. 4)

$Carry\ Propagate,\ P_i = A_i \oplus B_i$ ……………………………………....(Eq. 5)

$Carry\ Generate,\ G_i = A_i B_i$ ……………….………………………..………(Eq. 6)

$General\ Carry\ Equation,\ C_{i+1} = G_i + P_i C_i$ ……………..……………..(Eq. 7)

Expanded Carry Equations,

$C_0 = G_0 + P_0 C_{in}$ ……………….………………………………………....(Eq. 8)

$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$ ……………….……………………………(Eq. 9)

$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$ ………..……………….………(Eq. 10)

$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$ …………………...(Eq. 11)

$$C_i = G_i + P_iG_{i-1} + P_iP_{i-1}G_{1-2} + ... + P_iP_{i-1}P_{i-2}...P_0C_{in} \;\;..................(\text{Eq. } 12)$$

The same equation is used for the sum of CLA adder as well as the sum of the Carry Ripple adder. The expanded carry equations are generally Sum-of-Product expressions, usually implemented in parallel circuits to compute carry bits for all input bits to speed up additions. Figure 3.9 shows an implementation of a 4-bit CLA adder.

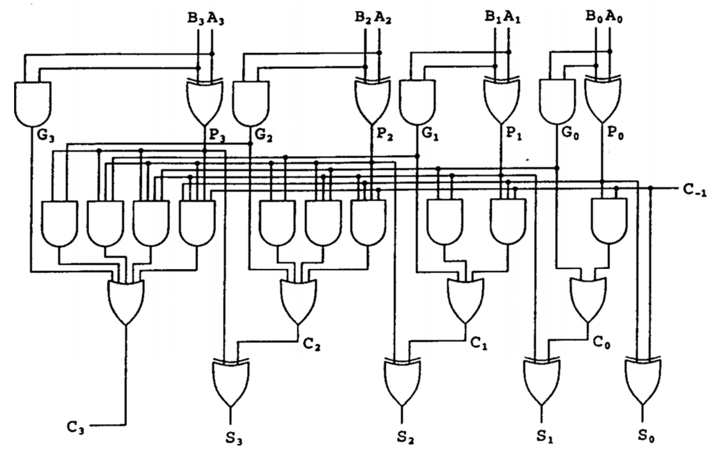


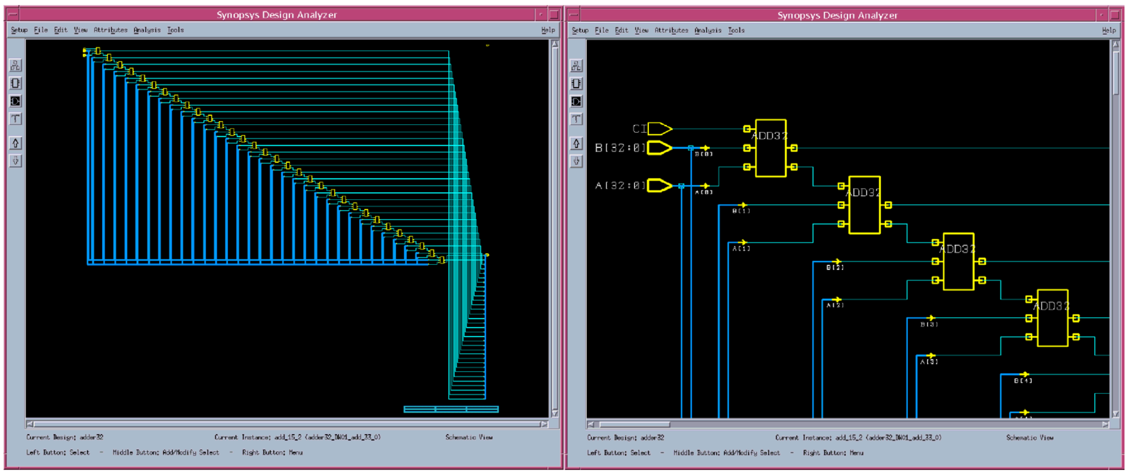
Fig. 3.9 4-bit Carry Look Ahead adder

The 4-bit CLA adder circuit shown in Figure 3.9 comprises four layers of logic components; the performance of the CLA adder depends on the propagation delay of the signals through these four layers of logic components in the CLA circuit structure.

However, as the width of the adder increases, the number of Sum-of-Product terms also increases in the carry equations. This in turn raises the fan-in and fan-out requirement on the logic components during implementation. As such, more layers of logic components are used to meet the higher fan-in and fan-out requirement when implementing the carry equations.

Although the performance of the CLA adder should theoretically remain constant, performance may differs for CLA adders of different widths, because of different number of layers of logic components may be implemented for each circuit.

A Carry Ripple adder is obtained by synthesizing the behavioural model adder with a clock constraint of 100ns. The schematic shown in Figure 3.10 consists of 32 Full Adder cells to implement the Carry Ripple adder.



Current Design: adder32 Current Instance: add_15_2 (adder32_DW01_add_33_0) Schematic View

Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Fig. 3.10 Behavioural model Carry Ripple adder schematic

The behavioural model adder circuit reached its performance limit, when synthesized at a clock constraint of 5ns. At the performance limit, the synthesized circuit (shown in Figure 3.11) is shown to have a CLA adder structure which uses parallel circuits to perform fast additions.

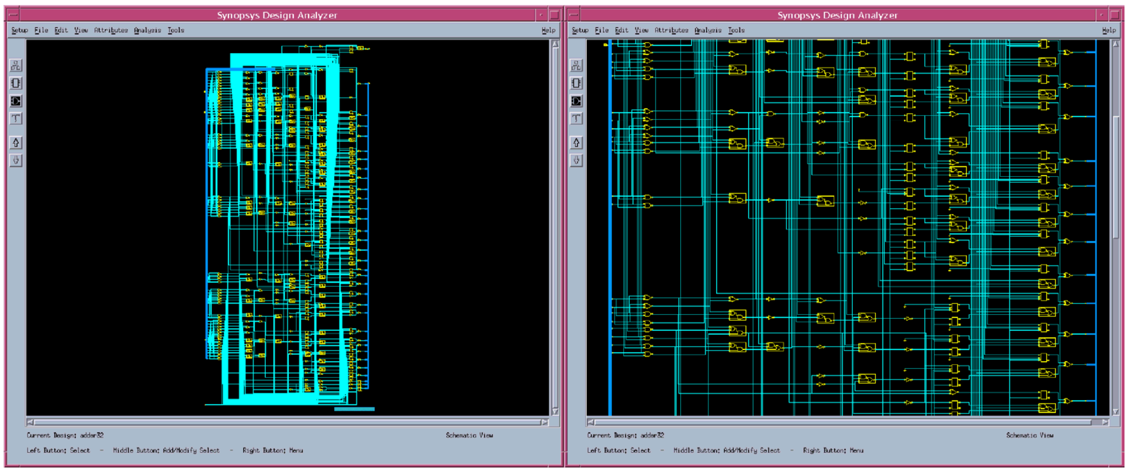

Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Current Design: adder32
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu
Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Current Design: adder32
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Fig. 3.11 Behavioural model CLA adder schematic

Model	Characteristics	100ns	5ns
Behavioural	Area (um^2)	24156	63654.61
	Dynamic Power (mW)	0.23	11.21
	Normalized Power (mW)	0.23	0.56
	Data Arrival (ns)	12	3.77

Table 3.2 Behavioural model adder circuit synthesis

Table 3.2 shows that from synthesizing the behavioural model adder circuit, a Carry Ripple adder is obtained with clock constraint at 100ns and the CLA adder circuit is obtained with clock constraint at 5ns. The Carry Ripple adder circuit has the slowest performance and lowest power consumption, while the CLA adder circuit has the fastest performance and highest power consumption. These two adder circuits are implemented in the ALU as the slow and fast adder.

3.2.5 Subtraction Circuits

Subtraction circuits are essentially addition circuits, with one of the operands complemented using inverters that force a logic high signal into the carry bit to implement the 2's complement for the complemented operand [40].

Figure 3.12 shows a block diagram for the subtraction circuit, where addition circuits with slight modifications were used. This therefore shows that the performance of subtraction circuits is close to that of addition circuits.

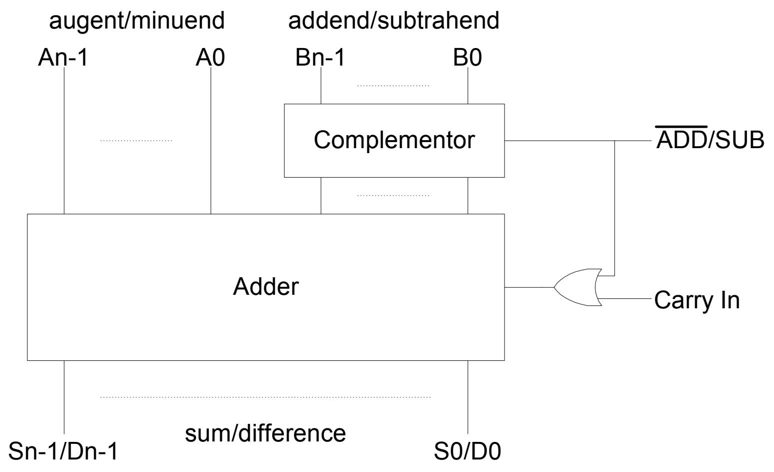


Fig. 3.12 Subtraction circuit implementation

The same process for synthesizing the addition circuits was repeated for subtraction circuits. Table 3.3 shows the circuit information for synthesizing a behavioural subtraction circuit model.

Model	Characteristics	100ns	5ns
Behavioural	Area (um^2)	25940.71	65312.05
	Dynamic Power (mW)	0.24	11.32
	Normalized Power (mW)	0.24	0.57
	Data Arrival (ns)	12.51	4.13

Table 3.3 Behavioural model subtractor circuit synthesis

Comparing Table 3.3 (subtraction circuits) with Table 3.2 (addition circuit), due to the additional inverters, the performance, area and power consumption of the subtraction circuit is only slightly slower or more (respectively) than the addition circuits.

3.2.6 Multiplication Circuits

The behavioural model multiplier synthesizes circuits with fast performance and high power consumption, even with a slow clock constraint at 100ns. As such, another multiplication circuit model (based on a simple multiplication algorithm) has been developed to synthesize slow-performance circuits that consume low power.

The schematic for the behavioural model multiplier shown in Figure 3.13 consists primarily of a block diagram with representations of the multiplication circuits and registers used to store the processed data.

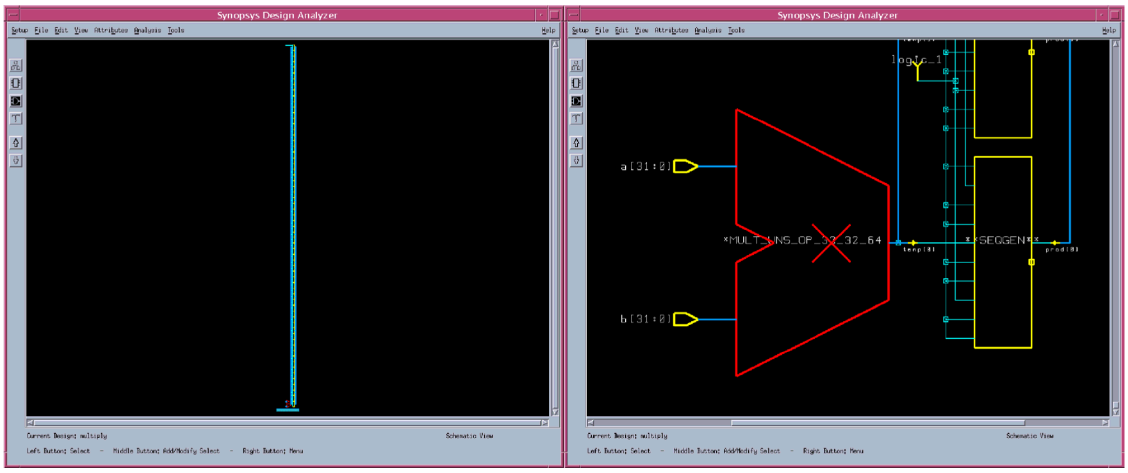


Fig. 3.13 Behavioural model multiplier schematic

Model	Characteristics	100ns	10ns	5ns
Behavioural	Area (um^2)	424413.25	648100.94	651773.88
	Dynamic Power (mW)	5.11	70.29	135.27
	Normalized (mW)	5.11	7.03	6.76
	Data Arrival (ns)	10.76	8.17	8.00 (Fail)

Table 3.4 Behavioural model multiplication circuit synthesis

Table 3.4 shows the performance and power consumption of the synthesized behavioural model multiplier circuits. With clock constraint at 10ns, the synthesized circuit nearly reached the performance limit. With clock constraint at 5ns, the synthesized circuit could not deliver the required performance. Hence, the circuit synthesized with clock constraint at 10ns is selected to implement as functional unit in the proposed ALU as the fast multiplier circuit.

A hybrid model multiplier that uses parallel shifted additions to compute multiplications is used to synthesize slow multiplier circuits. This hybrid model multiplier implements a shifted parallel addition algorithm, modified from the simple paper and pencil multiplication algorithm [40].

The paper and pencil multiplication algorithm performs multiplications by summing up additions of the multiplicand, aligned sequentially with respect to the multiplier. Figure 3.14 shows the 8-bit multiplication process of a 16-bit product. This multiplication process requires 8 steps of additions. In general, *n* steps of additions are required for *n*-bit multiplication.

```
          11101010     = 234
        X 11000100     = 196
          00000000     ----- (1)
         00000000      ----- (2)
        11101010       ----- (3)
       00000000        ----- (4)
      00000000         ----- (5)
     00000000          ----- (6)
    11101010           ----- (7)
 + 11101010            ----- (8)
  1011001100101000     = 45864
```

Fig. 3.14 Simple paper and pencil multiplication algorithm

The algorithm above has been modified to perform the additions in parallel alignment. With parallel additions, the number of steps it takes to compute a 16-bit multiplication via addition is reduced to 3. In general, $log_2\ (n)$ of addition steps are required for *n*-bit multiplication. Figure 3.15 shows the modified 8-bit multiplication process.

```
                    11101010      = 234
                  X 11000100      = 196

 11101010-(7)  00000000-(5)  11101010-(3)  00000000-(1)
+11101010 -(8)+00000000 -(6)+00000000 -(4)+00000000 -(2)
1010111110-(9)0000000000-(A)0011101010-(B)0000000000-(C)

              0000000000 -(A)      0000000000 -(C)
            +1010111110   -(9)   +0011101010   -(B)
             101011111000 -(D)    001110101000 -(E)

                    001110101000 -(E)
                  +101011111000      -(D)
                   1011001100101000  = 45864
```

Fig. 3.15 Modified multiplication algorithm

The schematic for the shifted parallel additions hybrid model is shown in Figure 3.16 comprises layers of logic components and addition circuit blocks.

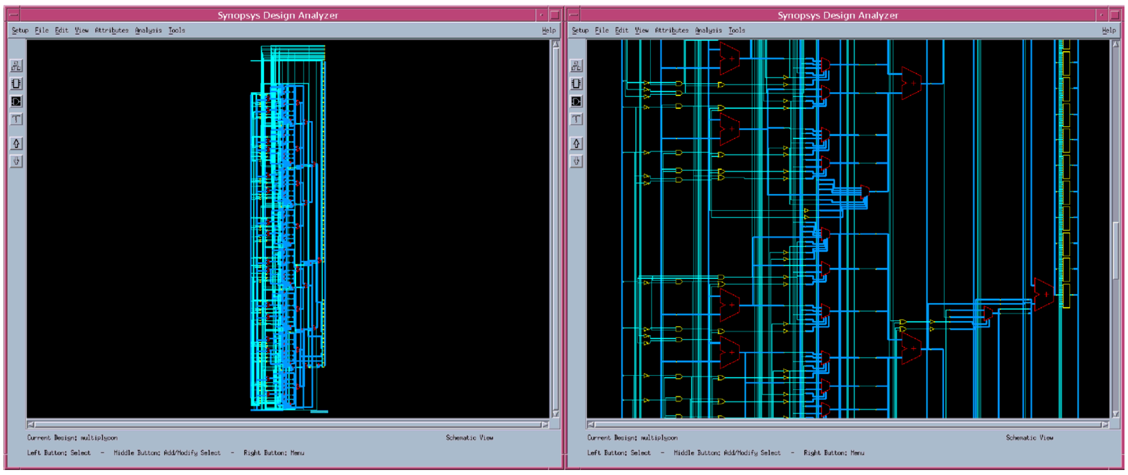

Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu
Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Fig. 3.16 Modified multiplication circuit schematic

Model	Characteristics	100ns	25ns	10ns
Parallel Shifted Additions Hybrid	Area (um^2)	343943.19	770634.06	1623810.75
	Dynamic Power (mW)	3.94	33.36	179.30
	Normalized (mW)	3.94	8.34	17.93
	Data Arrival (ns)	27.11	15.10	9.75

Table 3.5 Multiplication circuits synthesis

Table 3.5 shows the performance and power consumption of the behavioural model multiplication circuits, where they have reached their performance limit with clock constraint at 10ns. As this hybrid model multiplier is designed for implementing slow functional units, the circuit synthesized with clock constraint at 100ns is selected since it consumes the least power.

3.2.7 Division Circuits

The division operator used in the behavioural model division circuit produces only the quotient without the remainder; the remainder has to be computed separately using the modulus operator. As such, another division circuit model (based on the non-performing division algorithm [40]) has been developed for the proposed ALU to compute both division and remainder within one operation. Hence, we have two types of division circuits to cater for different division requirement – with or without remainder computation.

The schematic for the behavioural model division circuit shown in Figure 3.17 consists of a block diagram for representation of the division circuits and the registers used to store the processed data.

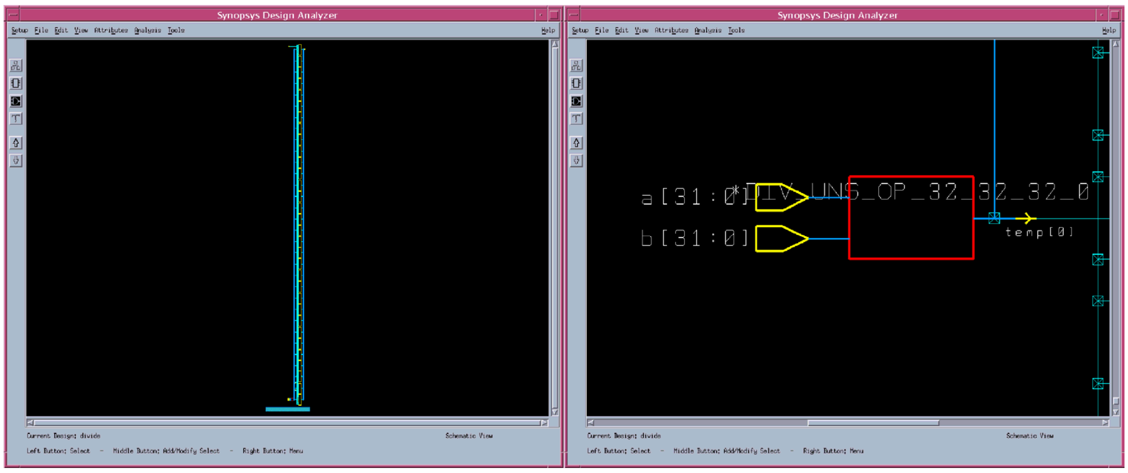


Fig. 3.17 Behavioural model division circuit schematic

Model	Characteristics	100ns	25ns
Behavioural	Area (um^2)	907729.81	1435558.38
	Dynamic Power (mW)	7.94	48.70
	Normalized (mW)	7.94	12.18
	Data Arrival (ns)	51.87	30.26

Table 3.6 Behavioural model division circuit synthesis

Table 3.6 shows the performance and power consumption of the behavioural model division circuits. The circuits reached their performance limit with clock constraint at 35ns.

The non-performing hybrid model is developed to compute both quotient and remainder within a single operation based on the non-performing division algorithm [40]. The reason for selecting this algorithm is its implementation is not complicated. Unlike other algorithms that proposed using different number system [41] or use fast look-up-tables to cache pre-processed data [42], this algorithm only needs circuits for subtraction, comparison, OR logic and left shifting operation when implemented.

The flowchart for the non-performing algorithm is shown in Figure 3.18.

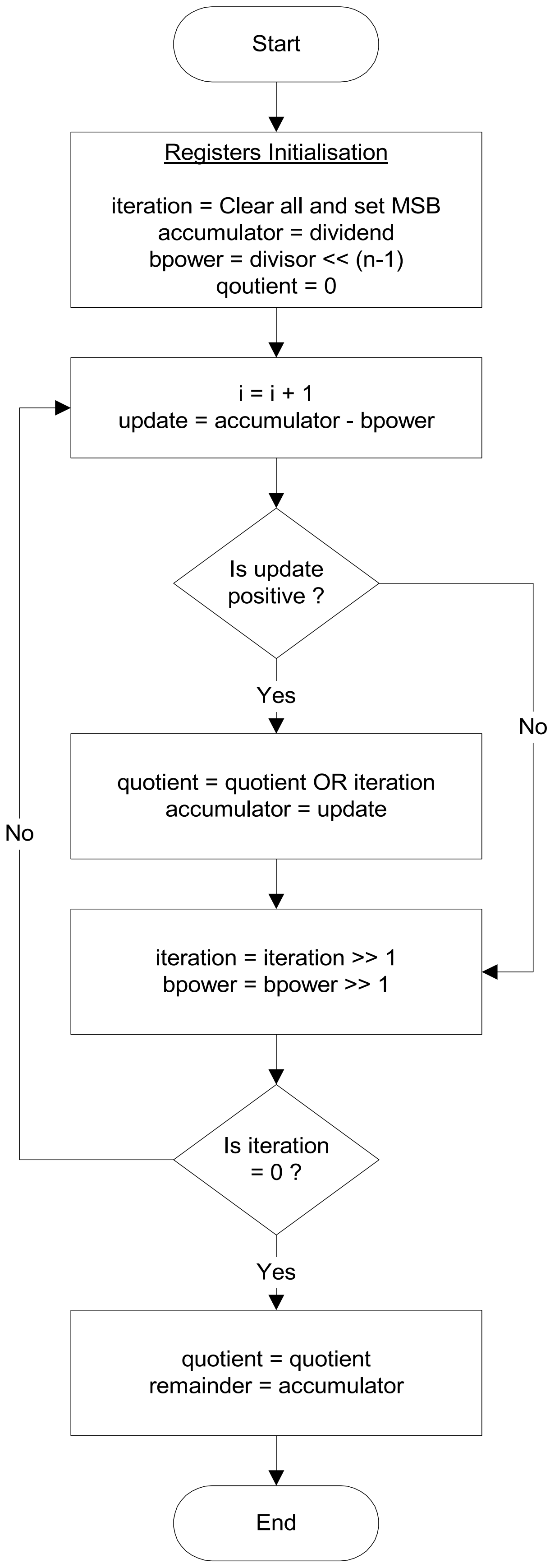


Fig. 3.18 Non-performing division algorithm

Figure 3.19 shows the non-performing division process for 5-bit data. In general, *n* steps of computations are required for *n*-bit division.

27 ÷ 11 = 2 remainder 5
or
11011 ÷ 01011 = 00010 remainder 00101

Iteration	Accum.	Bpower	Update	Quotient	Accum.(Updated)
10000	11011	0010110000	1101101011	00000	11011
01000	11011	0001011000	1111000011	00000	11011
00100	11011	0000101100	1111101111	00000	11011
00010	11011	0000010110	0000000101	00010	00101
00001	00101	0000001011	1111111010	00010	00101

Fig. 3.19 5-bit non-performing division process.

Figure 3.20 shows the schematic for the non-performing division hybrid model comprises of layers of logic components and subtraction circuits.

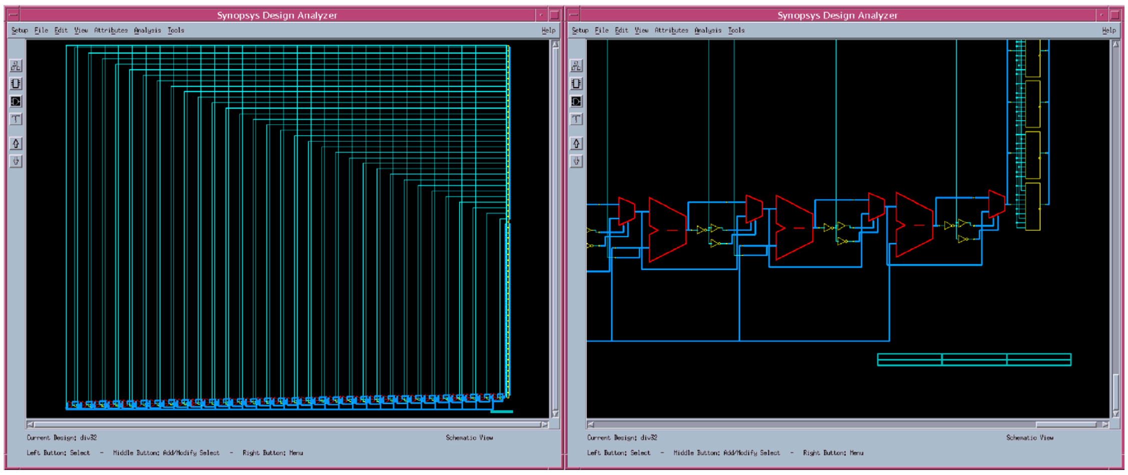

Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Current Design: div32
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu
Synopsys Design Analyzer
Setup File Edit View Attributes Analysis Tools
Help
Current Design: div32
Schematic View
Left Button: Select - Middle Button: Add/Modify Select - Right Button: Menu

Fig. 3.20 Non-performing division circuit schematic

Table 3.7 shows the performance and power consumption of the hybrid model division circuits.

Model	Characteristics	100ns	50ns
Non-Performing Division Hybrid	Area (um^2)	1338392.13	1531896.38
	Dynamic Power (mW)	10.49	23.94
	Normalized (mW)	10.49	11.97
	Data Arrival (ns)	54.68	49.70

Table 3.7 Division circuit synthesis performance

The hybrid circuits reached performance limit with clock constraint at 50ns. In a comparison of performance, the behavioural model synthesized circuits faster than the hybrid model, but cannot compute both quotient and remainder in one operation.

Thus, fast functional unit for division is implemented with the fastest circuit (higher power consumption) obtained from synthesizing the behavioural model. Slow functional unit is implemented with slowest circuits (lower power consumption) obtained from synthesizing the hybrid models.

With such implementation, it is necessary to differentiate PIns for division with and without remainder computation, so that the programmer can take advantage of the different circuits.

3.3 Analysis

3.3.1 Power Saving

As mentioned, the synthesized circuits are selected for implementation in the functional units, based on their performance and power consumption levels – shown in Table 3.8. The last column tabulates the difference in power consumption between the slow circuits and the fast circuits. These figures are also indicative of the amount of power that can be saved when the respective functional units are implemented in the proposed ALU.

Functions		**Slow Circuit**	**Fast Circuit**	**Difference**
Addition	Power (mW)	0.23	0.56	0.33
	Data Arrival (ns)	12.00	3.77	8.23
Subtraction	Power (mW)	0.24	0.57	0.33
	Data Arrival (ns)	12.51	4.13	8.38
Multiplication	Power (mW)	3.94	7.03	3.09
	Data Arrival (ns)	27.11	8.17	18.94
Division	Power (mW)	10.49	12.18	1.69
	Data Arrival (ns)	54.68	30.26	24.42

Table 3.8 Functional unit implementation

3.3.2 Optimal clock period

Assuming there are no time constraints from the execution of other pipeline stages like instruction fetch, decode, memory fetch and register write back, the number of stages assigned for execution will be based on performance of the functional units and the clock cycle time of each stage.

Given a particular performance and period for one clock cycle, the number of clock cycles a functional unit requires is calculated as the number of the clock cycles with the total time period that is sufficient to cover the functional unit's performance. For example, in a given period of 5ns for one clock cycle, 2 clock cycles is needed for a functional unit with a 6ns performance and 3 clock cycles for a functional unit with a 14ns performance.

The time period to assign for each clock cycle is computed based on a slack analysis. Slack is defined as the time difference between the time period and the functional unit's performance, which the functional units will be sitting idle after finished execution. Thus, using the same example of a given time period of 5ns, the slack for a functional unit of 6ns is 4ns and 1ns for a functional unit of 14ns.

The fastest performing functional unit amongst the selection is taken as the benchmark to determine the clock cycle for each stage. As seen in Table 3.8, the fastest performance recorded is 3.77ns. A preliminary time period of 4ns is therefore assigned as the clock cycle.

To compute an optimal time period for the clock cycle, a slack analysis is performed over a small range of time periods, starting from the preliminary value. The optimal time period is selected based on the time period with the smallest average slack. Table 3.9 shows the slack computations for time periods of 4ns, 5ns and 6ns and Table 3.10 shows the average slacks across the functional units.

	Time Period (ns)	**4**		**5**		**6**	
Addition	Clock Cycles (units)	3	1	3	1	2	1
	Total Time Period (ns)	12	4	15	5	12	6
	Data Arrival (ns)	12	3.77	12	3.77	12	3.77
	Slack (ns)	0	0.23	3	1.23	0	2.23
	Slack (normalized)	0	0.06	0.6	0.25	0	0.37
Subtraction	Clock Cycles (units)	4	2	3	1	3	1
	Total Time Period (ns)	16	8	15	5	18	6
	Data Arrival (ns)	12.51	4.13	12.51	4.13	12.51	4.13
	Slack (ns)	3.49	3.87	2.49	0.87	5.49	1.87
	Slack (normalized)	0.87	0.97	0.50	0.17	0.92	0.31
Multiplication	Clock Cycles (units)	7	3	6	2	5	2
	Total Time Period (ns)	28	12	30	10	30	12
	Data Arrival (ns)	27.11	8.17	27.11	8.17	27.11	8.17
	Slack (ns)	0.89	3.83	2.89	1.83	2.89	3.89
	Slack (normalized)	0.22	0.96	0.58	0.37	0.48	0.65
Division	Clock Cycles (units)	14	8	11	7	10	6
	Total Time Period (ns)	56	32	55	35	60	36
	Data Arrival (ns)	54.68	30.26	54.68	30.26	54.68	30.26
	Slack (ns)	1.32	1.74	0.32	4.74	5.32	5.74
	Slack (normalized)	0.33	0.44	0.06	0.948	0.89	0.96

Table 3.9 Slack computations

Time Period (ns)	**4**	**5**	**6**
Average for Addition and Subtraction	0.40	0.32	0.35
Average for Addition, Subtraction and Multiplication	0.47	0.37	0.42
Average for Addition, Subtraction, Multiplication and Division	0.45	0.41	0.55

Table 3.10 Average normalized slacks

In Table 3.10, we see that the clock period of 5ns consistently shows the smallest average slacks, and is thus selected as the time period to be implemented in the functional unit of the proposed ALU.

3.3.3 Area Penalty

Comparing with other ALU designs that use only one set of functional unit, our ALU requires more area to accommodate circuits for two sets of functional units. In another words, our ALU design trade space for power. Table 3.11 shows the area used by the slow and fast circuits. Table 3.12 shows a comparison base on the area used by slow and fast circuits.

Function	**Area (um^2)**		**Total**
	Slow Circuits	**Fast Circuits**	
Addition	24156	63654.61	87810.6
Subtraction	25940.71	65312.05	91252.8
Multiplication	343943.19	648100.94	992044
Division	1338392.1	1435558.4	2773951
Total	1732432	2212626	**3945058**

Table 3.11 Area of ALU

ALU Functional circuits	**Ratio**	
	Slow Circuits	**Fast Circuits**
Slow only	1	0.78
Fast only	1.28	1
Slow & Fast	2.28	1.78

Table 3.12 Ratio of circuit area

Base on Table 3.12, if we compare our ALU against a design which uses only our slow circuits, our ALU suffers an area penalty of additional 128% of area required by the slow circuits. The same penalty applies to comparing against a design which uses only our fast circuits, our ALU suffers an area penalty of additional 78% of area required by the fast circuits.

3.4 Chapter Summary

In this chapter, we discussed the function of CMOS logics in the circuits and how it conforms to the low-power consumption design of the proposed ALU. We also examined the hardware design and how it fits into the make of the proposed ALU. Last but not least, we analyzed in detail the implementation of the proposed ALU, demonstrating the selection process of the various circuits for the functional units.

The circuits designed for the proposed ALU are specifically made to fit either the fast or slow functional unit. They are described with behavioural, structural or hybrid models, then synthesized and implemented with CMOS logic. Circuits that are selected to be implemented in the functional units are chosen based on their performance and power consumption levels. By utilizing the slow function units appropriately, we are able to curtail excessive power consumption without affecting performance.

Base on the circuit performance, the time period for the clock cycle is computed for the ALU. A slack analysis was carried out on a small range of proposed time periods. The optimal time period is selected based on the smallest average slack.

CHAPTER 4

THE SOFTWARE INSTRUCTION SCHEDULER

In this chapter, we will describe the workings of the instruction scheduling algorithms – explaining in detail how it uses the functional units to enhance performance while reducing power consumption of the ALU during runtime. Broken down into several functional stages, every stage of the algorithm will be explicitly described. The developed software scheduler will then be tested, after which we will analyse the results.

4.1 Instruction Scheduling

4.1.1 Background

In many programs, it is common to execute several PIns consecutively to achieve certain tasks or computations. Due to instruction set constraints [43], these PIns may repeatedly use a few registers, such as accumulators, to store and execute interim data during execution. Such constraints cause some PIns to become dependent on the preceding PIns, as they rely on the executed results stored by the preceding PIns for their own computations.

As such, the dependent PIns are stalled during operations, and can only execute after the preceding PIns have completed execution. If executed in improper order, the dependent PIns will cause data hazards such as Read-after-Write, Write-after-Read and Write-after-Write [28], that will give rise to errors in the tasks or computations. However, when the dependent PIns are stalled, the performance of the ALU pipeline suffers as it sits idle while the dependent PIns are waiting to proceed.

Therefore, to prevent performance of the ALU from being stymied, the PIn order is rearranged so that PIns are executed continuously. This eliminates the idle time between preceding and stalled dependent PIns, thus effectively preventing stalls in the pipeline so that the performance does not suffer.

4.1.2 Scheduling Algorithms

Instruction scheduling is commonly used to rearrange PIns order, to resolve hazards caused by dependencies and enhance performance of the ALU pipeline. It can be performed with different algorithms, each using different methods to analyze dependencies among the PIns and rearrange them accordingly.

Many algorithms developed for instruction scheduling rearrange PIn order via dependency analysis [43, 44, 45, 46, 47]. Essentially, these algorithms first identify dependencies among PIns. It then arrange for independent PIns to fill in and increase the distance between preceding and succeeding dependent PIns, to resolve the dependencies between PIns. Different scheduling algorithms identify independent PIns in different forms. In general, independent PIns are PIns that can be safely relocated within a region of the program without causing hazards or computational errors.

During the scheduling operation, these independent PIns are identified and relocated between preceding and succeeding dependent PIns. This allows the ALU to execute them while the dependent PIns are waiting on the preceding PIns to complete execution. As a result, the ALU can execute PIns continuously, instead of idling while the dependent PIns are waiting to proceed.

Algorithms used to schedule processes in real time systems [48, 49], can also be used to schedule PIns. These algorithms analyze PIns and assign metrics like earliest and latest execution time to PIns for constructing time graphs. After which, the PIns are rearranged according to the time graphs. The same approach applies for algorithms used in static resource distribution models [50, 51] as well. These algorithms assign PIns with priority and treat registers as resources. Subsequently, the PIns are rearranged according to priority and availability of registers.

4.1.3 Performance Optimality

As mentioned, the objective of instruction scheduling is to enhance ALU performance, by identifying independent PIns and rearranging them to eliminate dependencies that give rise to stalls. However, the degree of optimality of the enhanced performance depends on the number of independent PIns the algorithms can identify within the scheduling window.

Scheduling window refers to the number of PIns that the scheduler can work with during operation. It is determined by static program analysis before the scheduling operation starts. As the algorithms perform the analysis, the size of the scheduling window may vary across different regions of the program, depending on the program flow structure.

Logically, a larger window size should result in scheduled PIns with better performance [52], since more PIns are available for rearrangement. However, this is not always true if there are insufficient independent PIns available in the scheduling window for rearrangement to resolve data hazards. Figures 4.1 and 4.2 illustrate the correlations between performance optimality and different scheduling window sizes, with respect to a fixed number of independent PIns.

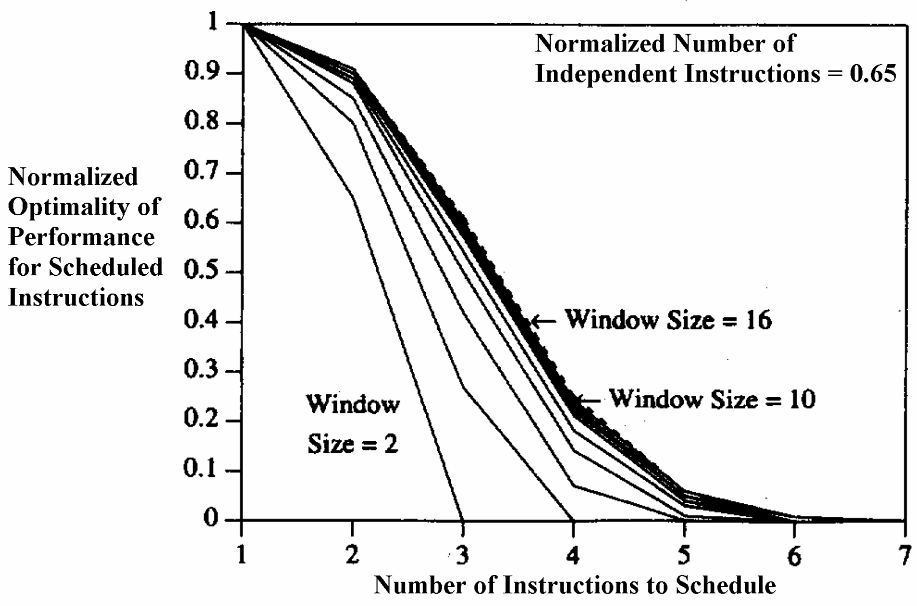


Fig. 4.1 Performance optimality with normalized number of independent instruction of 0.65

Extracted from [52], Figure 3 of "Instruction Window Size Trade-Offs and Characterization of Program Parallelism" by Pradeep K. Dubey et al

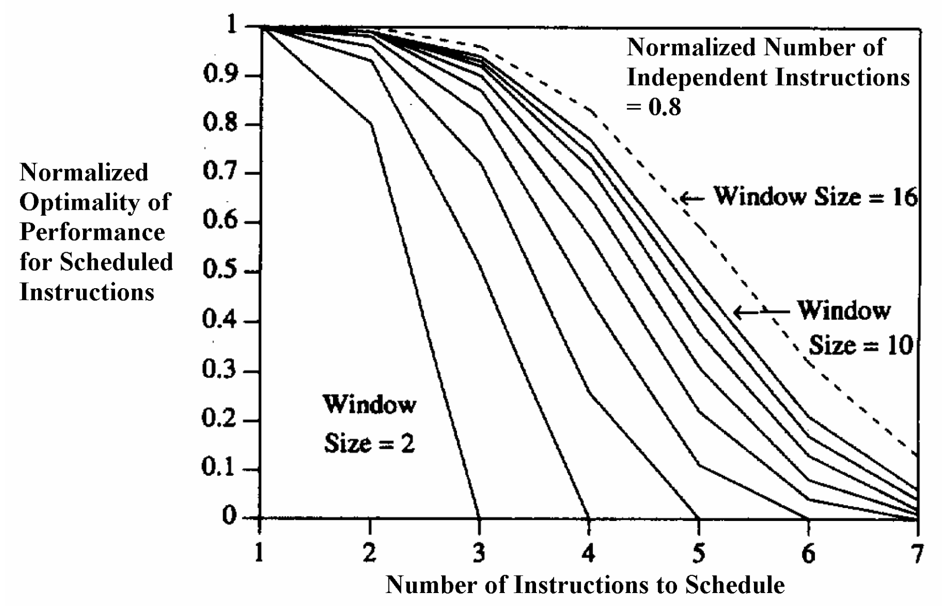


Fig. 4.2 Performance optimality with normalized number of independent instruction of 0.8

Extracted from [52], Figure 4 of "Instruction Window Size Trade-Offs and Characterization of Program Parallelism" by Pradeep K. Dubey et al

From the above figures, we can conclude that both the window size and the number of independent PIns available within the scheduling window affect performance optimality of scheduled PIns. It is also observed that from comparing the two figures, performance optimality can be limited by the number of independent instructions, even with an increase in the scheduling window size.

4.2 Software Instruction Scheduler

4.2.1 Introduction

Instruction scheduling may be implemented in both hardware and software. In hardware implementation, instruction scheduling takes place in complicated circuits during runtime operation. In this case, circuit size and its power consumption greatly constraint the complexity of the scheduling algorithm, as well as the size of the observation window.

As for software implementation, because instruction scheduling is conducted during offline compilation time, algorithm complexity and size of the observation window cease to be issues. Typically, the algorithms worked on the PIns before they are assembled into MIns, thus power is not incurred during runtime.

For the proposed ALU, software implementation of instruction scheduling is chosen for the mentioned advantages in the previous paragraph. Essentially, we selected this method of instruction scheduling as because it does not incur any power consumption during runtime – a primary consideration in our proposed low-power consumption ALU.

The software scheduler is designed to rearrange PIns via dependency analysis, following which, it generates directives for the assembler to map PIns to MIns that executes with the appropriate functional units. The software scheduler undergoes this process to achieve its objective of enhancing performance while reducing power consumption.

4.2.2 Scheduling Process

The proposed software scheduling process is performed in two phases; each phase is further divided into several stages.

The first phase is an *Initialization Phase* that processes the PIns into a format recognized by the scheduling algorithms used in the second phase, also known as the *Scheduling Phase*. In this phase, the PIns are analyzed and rearranged to enhance performance and reduce power consumption.

4.2.2.1 Initialization Phase

The *Initialization Phase* consists of three stages to prepare the PIn sources for the *Scheduling Phase*.

Initialization Phase Stage 1

The PIn sources usually contain many comments and references that are used to mark the PIns for easy inspection. Such comments and references not needed in the subsequent stages. At this stage, the software scheduler reads every line in the PIn sources and deletes those that are not PIns.

While reading every line in the PIn sources, for every unique PIn encountered, a new counter is created with an initial value of one. If the counter already exists for a particular PIn, the value on the counter will be accordingly incremented. At the end of this stage, a cleaned up version of the PIn sources is created, with the statistics of the PIn frequency stored in an external file.

Initialization Phase Stage 2

The algorithms in the *Scheduling Phase* analyze and rearrange the PIns to enhance performance and reduce power consumption during runtime. While this is done, the control flow of the program must be properly maintained, to ensure that the program works correctly after scheduling.

For example, it is common to execute arithmetic instructions and use the computed results in the evaluation of a conditional branch instruction. However, during rearrangement of instructions, it is important to ensure that the scheduling algorithms do not position the arithmetic instructions after the branch evaluation instruction, as this will result in errors in the program.

In Stage 2, control based PIns are identified and used to divide the program into segments. To ensure the control flow of the program remains intact, the algorithms work strictly on PIns within segments, never rearranging any PIns outside segments. Therefore, as a conservative approach, this prevents the control flow of the program from being affected by PIn rearrangement during execution. The program segments are subsequently assembled after the scheduling algorithms have worked on every one.

Initialization Phase Stage 3

During the *Scheduling Phase*, the algorithms identify instruction dependencies by matching the operands of different PIns. Should the operands of different PIns match, instruction dependencies may occur, depending on the execution order and the distance between the matched PIns.

To simplify the matching work, Generic Instructions (GIns) are used to provide an abstraction for the PIns. The GIns contain sufficient extracted information from the PIns the scheduling algorithms need to work on in the *Scheduling Phase*. The translation of PIns to GIns is performed in Stage 3.

At this stage, PIns are translated into GIns using the three generic mnemonics, shown in Table 4.1.

The GIn format consists of one of the three generic mnemonics in Table 4.1, together with one integer representing the destination operand and two integers to represent source operands.

GIn Mnemonic	**PIn Type**
F1F	Floating point Pins
IkF	PIns that required *k* clock cycle for execution ($k \geq 1$)
InXm	PIns that can be performed with slow or fast functional unit, *n* indicates the clock cycle required for execution with fast functional units and *m* indicates the clock cycle required for execution with slow functional units ($n \geq 1$, $m > 1$)

Table 4.1 GIn mnemonic descriptions

As seen in Table 4.1, *"F1F"* represents floating point instructions in PIns. *"IkF"* represents PIns that are supported with only one type of functional unit that requires *k* clock cycles for execution, where the value of *k* is one or greater. *"InXm"* represents PIns that can be performed with a fast or slow functional unit. The value *n* indicates the number of clock cycles required for execution with the fast functional unit, while *m* is indicative of the number of clock cycles for execution with the slow functional unit. Based on the hardware design of the proposed ALU in Chapter 3, *n* can be one or greater and *m* is greater than one. Unique integer numbers are used to represent different operands such as registers and data in PIns, for the source and destination operands in GIns.

As an abstraction of PIns, the GIns provide sufficient information for the scheduling algorithms to work with in place of the PIns. The benefit of the GIn format is that it makes for easy manipulation in the following phase, which is the *Scheduling Phase*.

4.2.2.2 Scheduling Phase

The *Scheduling Phase* is also divided into three stages. One stage is for analysis and the remaining two stages are for scheduling the GIn segments. In this phase, the algorithms work on the GIn segments obtained from Stage 3 of the *Initialization Phase*. At the end of the *Scheduling Phase*, the scheduled GIn segments are assembled and translated back into PIns. Directives are generated for the assembler to map PIns into MIns that are executed with the appropriate functional units.

Scheduling Phase Stage 1

Three sets of preliminary information are required to schedule the PIns correctly in the next two stages. In the first set of information, the dependent GIns have to be identified. The second set of information requires the identities of GIns that are already in proper order, prior scheduling. Last but not least is the moveable space of the GIns.

In Stage 1, the GIns are analyzed to obtain the required information. To identify dependent GIns, an instruction dependency check [43] is performed on every GIn in the working segment. Dependencies are then recorded in a Stall List.

During checking, the destination operand in the checking GIn will be matched against operands in the following GIn in the segment. If the destination operand of the checking GIn matches any of the operands in the following GIn, the following GIn will be marked as a dependent GIn on the checking GIn. A stall entry for the next following GIn will also be marked in the Stall List.

To ensure the scheduling process is stable, it is necessary to identify GIns that are already in proper order prior to scheduling and record them in a Fix List. This is to prevent the scheduling algorithms from rearranging GIns that already in proper order, which may cause extra cases of instruction dependencies under normal situations. Such GIns have the following description:

If the GIn at location *i+2* happens to be dependent on the GIn at location *i*, but the GIn at location *i+1* is independent, these three GIns will be considered to be in proper order, marked as fixed locations and recorded in the Fix List. The scheduling algorithms are then informed to avoid rearranging these GIns.

To obtain the movable space of the GIns, the GIns are analyzed to find every possible location where they can be relocated safely within the working segment. To compute the GIn movable space, the GIn is consistently relocated back and forth. This relocation stops when dependency occurs – the space between the two extreme locations is marked and recorded in the Space Chart as the movable space for the GIns.

Scheduling Phase Stage 2

The scheduling process begins after the final stage of information collection has been completed. At this stage, the *Scheduling Phase Interim Algorithm* works on the GIn segments on an interim schedule, enhancing performance under the assumption that GIns require only one clock cycle to perform, regardless of the GIn type.

Under this assumption, instruction dependency between two consecutive instructions is detected when the succeeding instruction writes to the same operand as the precedent instruction. This dependency is easily resolved by relocating an independent instruction in between these two dependent instructions.

This goes to show that a highly optimized performance under an interim schedule can be obtained under the one clock cycle assumption. This interim schedule will be reworked in the *Scheduling Phase Final Algorithm*, using the correct number of clock cycles for execution in *Scheduling Phase Stage 3*.

The *Scheduling Phase Interim Algorithm* works with the information recorded in the Space Chart, Stall List and Fix List. When the algorithm encounters a stall entry in the Stall List it is reading, it means that hazard has occurred as a result of dependency and has to be resolved.

As mentioned, to resolve the dependency under the one clock cycle assumption at this phase, an independent GIn has to be relocated and inserted between the two depending GIns. For example, if a GIn at location *i+1* is found to be dependent on a GIn at location *i*, an independent GIn will have to be relocated and inserted between *i* and *i+1* to resolve the dependency. If an independent GIn exists, it will be reordered so that the three GIns will be considered scheduled. They are then marked with fixed locations and recorded in the Fix List.

To locate an independent GIn, the algorithm looks up the location of the stall entry on the Space Chart and Fix List. The objective is to find an independent GIn with enough movable space for relocation to resolve the stall. After such an independent GIn is found and relocated, the Space Chart, Fix List and Stall List will be recomputed. If no independent GIns can be found, wait states like NOP or other forms of delay mentioned in Chapter 2.1.2.1 will be used to resolve dependencies.

Figure 4.3 depicts a detailed flowchart for *Scheduling Phase Interim Algorithm*.

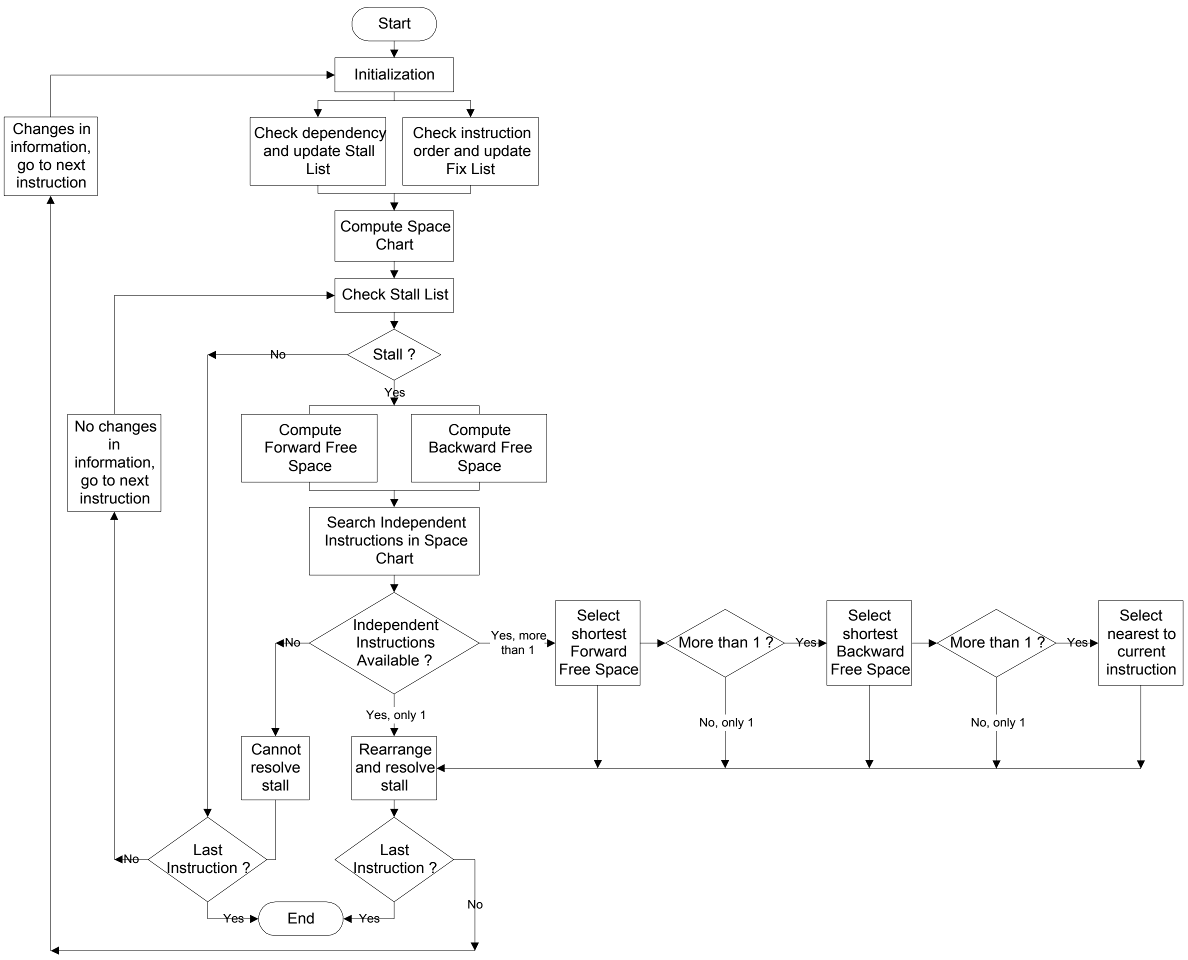

Start
Initialization
Changes in information, go to next instruction
Check dependency and update Stall List
Check instruction order and update Fix List
Compute Space Chart
Check Stall List
Stall ?
No
Yes
No changes in information, go to next instruction
Compute Forward Free Space
Compute Backward Free Space
Search Independent Instructions in Space Chart
Independent Instructions Available ?
No
Yes, more than 1
Select shortest Forward Free Space
More than 1 ?
Yes
Select shortest Backward Free Space
More than 1 ?
Yes
Select nearest to current instruction
Yes, only 1
No, only 1
No, only 1
Cannot resolve stall
Rearrange and resolve stall
Last Instruction ?
No
Last Instruction ?
No
Yes
End
Yes

Fig. 4.3 Scheduling Phase Interim Algorithm Flow Chart

Scheduling Phase Stage 3

The *Scheduling Phase Final Algorithm* in Stage 3 has two objectives. The first objective is to rework the interim schedule with the actual number of clock cycles required by the functional units to perform. The second objective is to identify and rearrange GIns that can be performed with slow functional units to reduce power consumption during runtime without incurring loss in performance. The details of the *Scheduling Phase Final Algorithm* are described as follows:

In this phase, the algorithm identifies GIns with mnemonic *"IkF"* and *"InXm"* (mentioned in Section 4.2.3). GIns with mnemonic "*IkF"* can only be executed with one type of functional unit. The integer *k* represents the number of clock cycles required by the GIn to perform. If the identified GIn is "*IkF"* and *k* is greater than one, it will be registered as XCycle when the GIn is recorded in the Stall List. The algorithm will then use these GIns to resolve dependencies.

GIns with mnemonic *"InXm"* can be executed with fast or slow functional units. The integer *n* represents the number of clock cycles required by the GIns for execution using fast functional units, while the integer *m* represents the number of clock cycles required to execute using slow functional units. If the identified GIn is *"InXm"*, the integer *n* will be registered as SCycle and *m* will be registered as XCycle. The algorithm will base on XCycle or SCycle to rearrange the GIns to resolve dependencies.

Starting with XCycle, the algorithm finds or creates the situations mentioned in Chapter 1.3 for these GIns to perform with slow functional units. Should dependencies occur because of slow functional units used, the algorithm will repeat the scheduling process and rearrange the GIns based on SCycle, using fast functional units to execute the GIns.

The algorithm uses four conditions (described below) which serve as mechanisms to analyze and rearrange the GIns to resolve dependencies. The algorithm fixes the dependencies based on XCycle or SCycle, using information recorded in the Space Chart, Fix List and Stall List.

If the identified GIn is *"IkF" with* an entry on the Stall List, the algorithm will check the identified GIn against the four conditions to resolve dependencies with XCycle only. If the identified GIn is *"InXm"*, the algorithm will check it against the four conditions and arrange for its execution with slow functional units using XCycle first. Should dependencies occur with XCycle, the algorithm would recheck the identified GIn against the four conditions, after which it will arrange for execution of the GIn with fast functional units using SCycle.

In the following description of the four conditions, we will arbitrarily use the term Cycle to refer to XCycle or SCycle.

Condition 1

The movable space of the identified GIn is read from the Space Chart. Base on the space between the current location and the location where the identified GIn can be rearranged to execute latest in time, the forward movable space is computed.

If 'Cycle is lesser or equal to the forward movable space', *Condition 1* is met. If 'Cycle is greater than the forward movable space', *Condition 1* is not met and *Condition 2* will be checked.

Condition 2

If 'the identified GIn is not marked as fixed location in the Fix List', the following procedure proceeds, else *Condition 2* is skipped and *Condition 3* is proceeded instead.

The maximum movable space is computed base on the space recorded in the Space Chart. It is define as the space between the locations where the identified GIn can be arranged to execute earliest and latest in time.

If 'the maximum movable space is equal or greater than Cycle', the identified GIn will be rearranged to the location where it can be executed earliest in time and *Condition 3* is met. If 'the maximum movable space is smaller than Cycle', *Condition 2* is not met and *Condition 3* will be checked.

Condition 3

At this stage, an arbitrary number of independent GIns *N*, is required to be rearranged after the identified GIn location to extend the forward movable space. *N* is defined as,

$$N = \text{Cycle - forward movable space}$$

Hence, '*N* number of independent GIns will be searched within the Space Chart'. If this is found, these *N* independent GIns will be rearranged to locations after the identified GIn and *Condition 3* is met. If this cannot be found, *Condition 3* is not met and *Condition 4* will be checked.

Condition 4

At this stage, an arbitrary number of independent GIns *M*, is required to be rearranged to the locations after the identified GIn to extend the maximum movable space. *M* is define as,

M = XCycle - Maximum free movable space span

Hence, '*M* number of independent GIns will be searched within the Space Chart'. If it can be found, the identified GIn will be rearranged to the location where it can be executed earliest in time, the *M* independent GIns will be rearranged to locations after the identified GIn and *Condition 4* is met. If it cannot be found, *Condition 4* is not met.

If *Condition 4* is not met, it implies dependencies still exist even after the identified GIn has been attempted to be rearranged. An entry will then be made in the Stall List to mark the dependency of the identified GIn.

To resolve the dependency, wait states mentioned in Chapter 2.1.2.1 will have to be implemented, since rearranging the GIns fail to resolve the problem. Fast functional units will be used to execute identified GIns with the mnemonic *"InXm"*, in order to reduce the amount of wait states required.

If any of the four conditions is met during the process, it implies that scheduling of the identified GIn has been successful. Dependencies will be resolved for identified GIns with the mnemonic *"IkF"*, and any entry in the Stall List will be cleared. For identified GIns with the mnemonic *"InXm"*, the value of Cycle will be recorded in the Cycle List. The assembler can then use the Cycle List as directives to map PIns with appropriate MIns. Similarly, the Stall List can be used as directives for the assembler to insert wait states accordingly. The detailed flowchart for Scheduling Phase Final Algorithm Flow Chart is shown in Fig. 4.4.

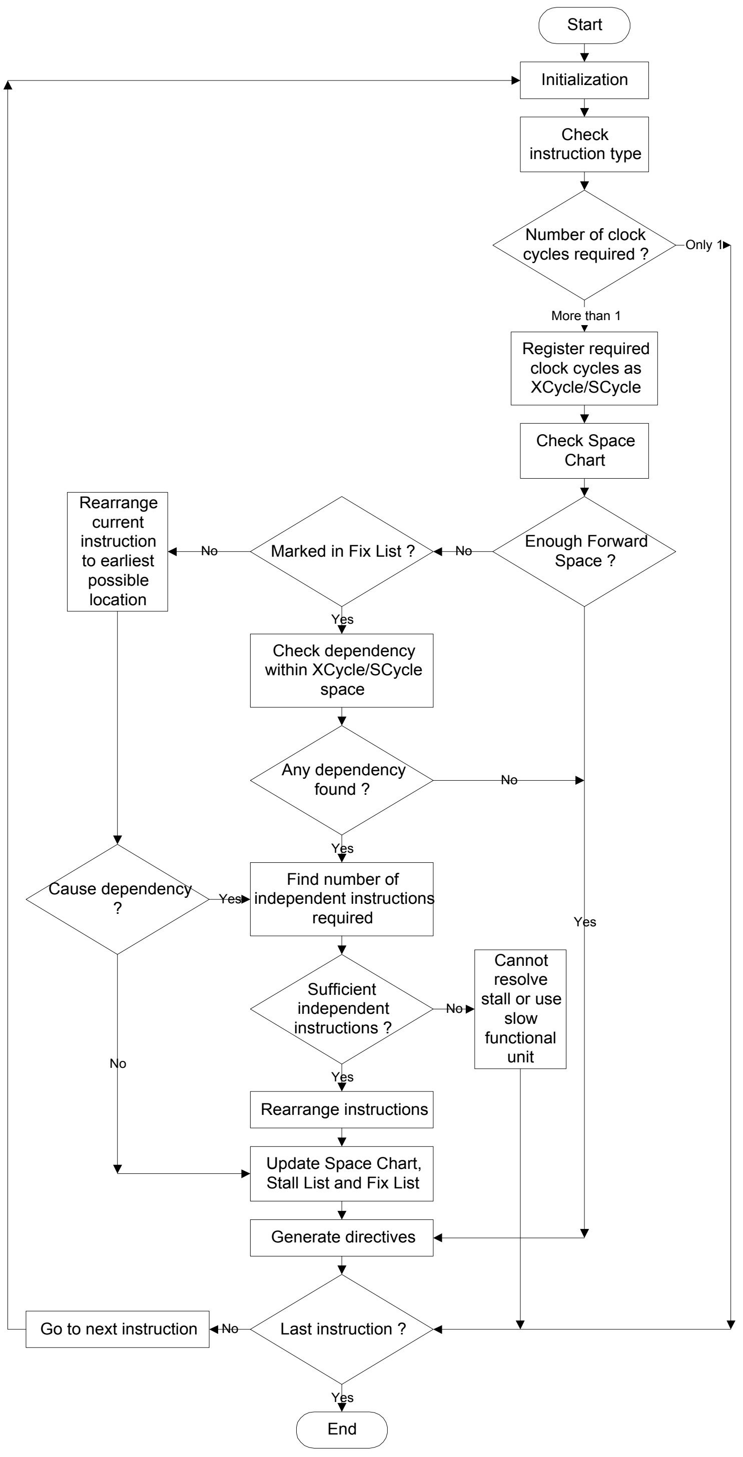

Start
Initialization
Check instruction type
Number of clock cycles required ?
Only 1
More than 1
Register required clock cycles as XCycle/SCycle
Check Space Chart
Enough Forward Space ?
No
Marked in Fix List ?
No
Rearrange current instruction to earliest possible location
Yes
Check dependency within XCycle/SCycle space
Any dependency found ?
No
Yes
Cause dependency ?
Yes
Find number of independent instructions required
Yes
Sufficient independent instructions ?
No
Cannot resolve stall or use slow functional unit
No
Yes
Rearrange instructions
Update Space Chart, Stall List and Fix List
Generate directives
Go to next instruction
No
Last instruction ?
Yes
End

Fig. 4.4 Scheduling Phase Final Algorithm Flow Chart

4.3 Analysis

The software scheduler which we developed was tested on several file compression programs. From the scheduled programs we can see that there are many cases in which the scheduler can rearrange PIn order to resolve stalls and assign slow functional units. On the other hand, there are also several cases where the scheduler could not improve the situations, due to insufficient independent PIns available to resolve dependencies.

From the scheduled programs, we also managed to obtain the statistics on the number and type of PIns, and the frequency of slow functional unit assignment, as a result of rearranging the PIns using the scheduler. Power savings can be estimated base on these statistics.

The following sections describes some best and worst cases found with the scheduler, and analysis on the statistics obtained.

4.3.1 Good and Bad Cases

From these tests we selected three processed segments as examples – two of good cases and one bad case, which the software scheduler encountered in the course of the tests.

Good cases occurred when there were lesser stalls (due to dependencies) with sufficient independent instructions to resolve, while still able to assign slow functional units. Some good cases did not even have any stall. Bad cases on the other hand, were usually caused by insufficient or entire lack of independent instructions to resolve stalls; bad cases could still occur even though there were few stalls – due to insufficiency of independent PIns.

The GIn and PIn segments are shown with the individual cases. The GIn segment records the GIns of before and after scheduling, along with information on stalls and functional unit assignments. Unresolvable stalls are recorded in the Stall column with a non-zero integer. The Cycle column recorded the number of clock cycle that instructions should use during execution. With reference from the GIns, it can be shown the type of functional unit that has been assigned to the corresponding PIns.

Table 4.2 and 4.3 show the one of the good case (Case 1) found. This case does not contain any stalls in its original PIn order, and slow functional units can be assigned for execution without instruction rearrangement.

Case 1

No	Inst	Dest	Src1	Src2	Stall		ReX	Inst	Dest	Src1	Src2	Stall	Cycle
0	I1F	323	7	7	0		0	I1F	323	7	7	0	1
1	I1F	7	8	8	0		1	I1F	7	8	8	0	1
2	I1S3	8	35	35	0		2	I1S3	8	35	35	0	3
3	I1F	324	5	5	0		3	I1F	324	5	5	0	1
4	I1F	325	6	6	0		4	I1F	325	6	6	0	1
5	I1F	5	302	302	0		5	I1F	5	302	302	0	1
6	I1F	327	326	42	0		6	I1F	327	326	42	0	1

Table 4.2 GIn segment for Case 1

Ref	Original	Destination	Source		Ref	Rearranged	Destination	Source
536	Push	Bp			536	Push	Bp	
537	Mov	Bp	Sp		537	Mov	Bp	Sp
538	Sub	Sp	200		538	Sub	Sp	200
539	Push	Si			539	Push	Si	
540	Push	Di			540	Push	Di	
541	Mov	Si	[bp+04]		541	Mov	Si	[bp+04]
542	Cmp	word ptr [2208]	0		542	Cmp	word ptr [2208]	0

Table 4.3 Program segment for Case 1

Table 4.4 and 4.5 show another good case (Case 2) found. This case shows that instruction rearrangement is able to resolve stalls and assign slow functional units for execution.

Case 2

No	Inst	Dest	Src1	Src2	Stall		ReX	Inst	Dest	Src1	Src2	Stall	Cycle
0	I1F	1600	7	7	0		0	I1F	1600	7	7	0	1
1	I1F	7	8	8	0		1	I1F	7	8	8	0	1
2	I1S3	8	215	215	0		3	I1F	2	1597	1597	0	1
3	I1F	2	1597	1597	0		4	I1F	1	1601	1601	0	1
4	I1F	1	1601	1601	0		5	I1F	4	609	609	0	1
5	I1F	4	609	609	0		2	I1S3	8	215	215	0	3
6	I1F	236	4	4	1		6	I1F	236	4	4	0	1
7	I1F	237	1	1	0		7	I1F	237	1	1	0	1
8	I1A3	1598	215	215	0		8	I1A3	1598	215	215	0	3
9	I1F	4	237	237	0		9	I1F	4	237	237	0	1
10	I1F	1	236	236	0		10	I1F	1	236	236	0	1
11	I1F	8	7	7	0		11	I1F	8	7	7	0	1
12	I1F	7	1602	1602	0		12	I1F	7	1602	1602	0	1

Table 4.4 GIn segment for Case 2

Ref	Original	Destination	Source		Ref	Rearranged	Destination	Source
2809	Push	Bp			2809	Push	Bp	
2810	Mov	Bp	Sp		2810	Mov	Bp	sp
2811	Sub	Sp	0004		2812	Mov	Bx	[2628]
2812	Mov	Bx	[2628]		2813	Mov	Ax	[bx+02]
2813	Mov	Ax	[bx]		2814	Mov	Dx	[bx]
2814	Mov	Dx	[bx]		2811	Sub	Sp	0004
2815	Mov	[bp-04]	Dx		2815	Mov	[bp-04]	Dx
2816	Mov	[bp-02]	Ax		2816	Mov	[bp-02]	Ax
2817	Add	Word ptr [2628]	0004		2817	Add	word ptr [2628]	0004
2818	Mov	Dx	[bp-02]		2818	Mov	Dx	[bp-02]
2819	Mov	Ax	[bp-04]		2819	Mov	Ax	[bp-04]
2820	Mov	Sp	Bp		2820	Mov	Sp	bp
2821	Pop	Bp			2821	Pop	Bp	

Table 4.5 Program segment for Case 2

Table 4.6 and 4.7 show a bad case (Case 3) found with not enough independent instructions available to resolve dependencies.

Case 3

No	Inst	Dest	Src1	Src2	Stall		ReX	Inst	Dest	Src1	Src2	Stall	Cycle
0	I1F	3	1618	1618	0		0	I1F	3	1618	1618	0	1
1	I1F	4	1610	1610	0		1	I1F	4	1610	1610	0	1
2	I1F	4	59	59	1		2	I1F	4	59	59	1	1
3	I1A3	4	1	1	2		3	I1A3	4	1	1	2	1
4	I1A3	4	4	4	3		4	I1A3	4	4	4	3	1
5	I1F	1	148	148	0		5	I1F	1	148	148	0	1
6	I1A3	1	4	4	4		6	I1A3	1	4	4	4	1
7	I1F	1612	1	1	5		7	I1F	1612	1	1	5	1

Table 4.6 GIn segment for Case 3

Ref	Original	Destination	Source		Ref	Rearranged	Destination	Source
2859	Pop	Cx			2859	Pop	Cx	
2860	Mov	Dl	[25D2]		2860	Mov	Dl	[25D2]
2861	Mov	Dh	00		2861	Mov	Dh	0
2862	Add	Dx	Ax		2862	Add	Dx	ax
2863	Inc	Dx			2863	Inc	Dx	
2864	Mov	Ax	word ptr [235A]		2864	Mov	Ax	word ptr [235A]
2865	Add	Ax	Dx		2865	Add	Ax	dx
2866	Mov	Word ptr 2622]	Ax		2866	Mov	word ptr [2622]	ax

Table 4.7 Program segment for Case 2

4.3.2 Statistics and Power Savings

Statistics on the arithmetic instructions are obtained by identifying and counting instructions in *Initialization Phase Stage 1*. Power savings with the proposed ALU is estimated base on the number of instructions assigned to using slow functional units and the differences in power consumption between using slow and fast functional units (as mentioned in Section 3.3).

Table 4.8 shows the statistics on the arithmetic instruction found in the tested programs, while Table 4.9 shows the number of instructions assigned using slow functional units in the tested programs. The estimated savings on power consumption with the proposed ALU is summarized in Table 4.10. It is derived from the data in

Table 4.9 and based on the savings on power consumption between fast and slow functional units summarized in Table 3.8.

Programs	**Total**	**Arithmetic**	**Addition**	**Subtraction**	**Multiplication**	**Division**
ARJ	48431	6085	3935	1787	265	98
PKZIP	19800	4848	2857	1800	152	39
PKUNZIP	13944	2561	1990	1235	104	32
DUNZIP32	21875	1757	1701	535	29	27
UNRAR	14001	1363	1057	283	16	7
ACE	35061	2321	2075	1250	171	75

Table 4.8 Statistics on tested programs

Table 4.8 shows that in general, addition instructions dominate the number of arithmetic instructions, followed by subtraction, multiplication and division instructions being the least in all the tested programs.

Program	Addition	Subtraction	Multiplication	Division	Total
ARJ	1451 (36.9%)	828 (46.3%)	78 (29.4%)	14 (39.0%)	2371 (39.0%)
PKZIP	1731 (60.6%)	1103 (61.3%)	65 (42.8%)	12 (60.0%)	2911 (60.0%)
PKUNZIP	1196 (60.1%)	722 (58.5%)	53 (51.0%)	4 (58.8%)	1975 (58.8%)
DUNZIP32	302 (17.8%)	210 (39.3%)	11 (37.9%)	3 (22.9%)	526 (22.9%)
UNRAR	260 (24.6%)	43 (15.2%)	2 (12.5%)	2 (22.5%)	307 (22.5%)
ACE	665 (32.0%)	450 (36.0%)	32 (18.7%)	11 (32.4%)	1158 (32.4%)

Table 4.9 Number of instructions assigned to use slow functional unit

Table 4.9 shows that different programs achieved different amount of slow functional unit assignments with the software scheduler.

Program	Addition	Subtraction	Multiplication	Division	Total
ARJ	478.83	273.24	241.02	23.66	1016.75
PKZIP	571.23	363.99	200.85	20.28	1156.35
PKUNZIP	394.68	238.26	163.77	6.76	803.47
DUNZIP32	99.66	69.3	33.99	5.07	208.02
UNRAR	85.8	14.19	6.18	3.38	109.55
ACE	219.45	148.5	98.88	18.59	485.42

Table 4.10 Estimated power consumption savings (mW)

Table 4.10 shows the power savings achievable from using the proposed ALU to execute the test programs. It is computed based on the number of instructions assigned to use slow functional units (Table 4.9) and the amount of power saved from using slow functional units (Table 3.8).

4.4 Chapter Summary

In this chapter, we discussed how the design and function of the software scheduler enhance performance while reducing power consumption of the proposed ALU. Algorithms are central to the functioning of the software scheduler, as they are primarily responsible for the analysis and rearrangement of instructions.

In essence, the algorithms resolve the dependency between instructions, enabling the ALU to pack a better and more power-efficient performance. After the instructions have been analyzed for dependencies, they are rearranged for continuous execution to avoid the problem of an idling ALU that drags performance down, and using slow functional units for execution whenever possible, to reduce power consumption.

From the scheduled programs, the good and bad cases are identified and illustrated. Statistics on the instruction types in the test programs are obtained. These statistics give us an estimate of savings in power consumption, when the proposed ALU is used to execute the test programs. By illustrating the amount of power that can be saved using the proposed ALU, we are therefore able to prove the viability the design.

CHAPTER 5

CONCLUSIONS

This chapter summarizes the previous chapters and concludes the thesis, recommending future work which can be done in the future to improve the project.

5.1 Conclusion

Considering the widespread use of mobile electronic devices today, the IT and electronic industry would be able to reap many benefits from a high-performance, low-power consuming microprocessor. Not only can such a microprocessor increase the usage periods of electronic devices, chances are it would result in numerous other mobile electronic innovations.

We therefore focussed on developing the ALU – the heart of the microprocessor – in this project, to work towards the goal of creating a low power microprocessor. An ALU with a simple design was thus developed, for the obvious advantage that it would consume less power by virtue of simplicity of its operations, without compromising performance.

There are three major phases of development in this project: architecture design, hardware and software development.

The first thing we did was to design the ALU hardware architecture and define the run time operation. The ALU architecture consists of a set of slow and fast functional units for executing instructions, a Control Unit for synchronizing operations and a Register File which can update several registers within one clock cycle.

The simplicity of the ALU design enables power to be consumed mainly by instruction execution during runtime, with no extraneous consumption. This lean power consumption is mostly made possible by the ALU's methodical inner processes. During runtime, the Control Unit selects the functional units for instruction execution based on the MIns, while the software scheduler rearranges PIns prior to execution, so stalls are resolved and PIns are assigned to slow functional units whenever possible. As such, the ALU's function is simplified so that all it has to do is to execute MIns.

Next on the list was the functional unit hardware, which we developed so that it conformed to the requirement of the ALU design. We primarily did this by implementing slow functional units for execution instead of fast ones, as they consumed less power than fast functional units during execution. Prior to this decision, simulations were conducted on several circuit designs and models, allowing us to acquire an estimate on the power savings from using the slow functional units – a significant figure compared with the power consumption level of fast functional units.

Third and lastly, is the software scheduler. Developed to rearrange instruction order, we designed it to fully exploit the ALU architecture for resolving stalls and reducing power consumption. Likewise the other aspects of the ALU, the two algorithms developed for these tasks were designed to keep within the limits of minimal power consumption in the way they are processed.

The first algorithm rearranges instructions to obtain an interim schedule that focused only on high performance, under the assumption that all instructions required only one clock cycle for execution. After which, the second algorithm works on the interim schedule to correct the assumption while looking for opportunities to assign instructions to be executed with slow functional units.

Two lists of directives are generated after the two algorithms have been processed – one is a functional unit assignment list, while the other is a list of stalls. The stall directives embed delay information in instruction opcodes, which in turn instruct the Control Unit to delay instruction issue. This step of the scheduling process thereby avoids power incurrence when executing instructions that insert wait states.

After the software scheduler was completed, we put it to the test on several file compression programs. Analysis of the test results shows that stalls and power consumption were reduced when the proposed ALU was used to execute instructions rearranged by the software scheduler. This positively confirms the effectiveness of the software scheduler and the potential benefits that can be garnered when implemented in mobile electronic devices.

5.2 Future Work

For Design and Architecture

To further reduce power consumption, it is possible to incorporate the simple ALU architecture with other voltage-reduction techniques, such as those mentioned in Chapter 1. However, there are potential problems to be dealt with before the two can be combined. Challenges that have to be faced include issues in the implementation of techniques such as real time slack analysis and interfacing circuits between different voltage systems.

In order to enhance performance and increase opportunities so slow functional units are used to assign instructions, the ALU design can adopt the multiple-instruction issue architecture like the VLIW. With the multiple-instruction issue architecture, the ALU can execute other instruction streams when a particular stream is stalled, reverting to it when it is ready to proceed. This way, the ALU constantly executes instructions from different streams [25].

While this deals with the problem of stalling, there are two major challenges that arise with this method. Firstly, this will involve a complicated Control Unit design as it has to synchronize all the instruction streams supported, while consuming reasonable additional power. Offline software may come in useful here as it can offload some or all of the synchronizing tasks for the Control Unit. Secondly, it must be ensured that an optimal number of functional units are implemented to avoid structural hazards – a common problem in multiple-instruction issue architecture.

For Functional Units

Being a hot topic in the electronic industry at the moment, designs for functional units are periodically reviewed and updated with the ongoing research on low-power arithmetic circuit designs. Also, as technology improves, the circuit models can be re-synthesized using latest standard cell library to obtain better performance and power consumption levels.

With the present ALU design, either fast or slow functional units can be used. However when functions like multiplication is involved, it may be advantageous to introduce a small range of medium functional units to fill out the performance difference between the slow and fast functional units. This is so that when slow functional units cannot be assigned, a medium-performing one can be assigned instead of a fast functional unit, enabling a measure of power to be saved. Potentially, problems might surface because of the increase in workload for the software scheduler – it would have to go through more options before selecting a suitable functional unit to execute an instruction.

For Software Scheduler

Although the software scheduler has demonstrated the effectiveness of instruction rearrangement to resolve stalls and reduce power consumption, there is still a generally tentative outlook of it. The approach towards the design of algorithms based on intra-segment analysis is still fairly conservative, but the algorithms are already quite complex in its first stage of development. However, the effectiveness of the software scheduler can be enhanced by incorporating more complicated tasks in the algorithms, such as transferring unused independent instructions or carrying out scheduling based on inter and intra segment analysis.

APPENDIX

CMOS CIRCUIT CHARACTERIZATION

In this appendix, we will describe the propagation delay, signal quality and power consumption pattern of the CMOS circuits, which were implemented as basic logic gates and adder blocks in the simulation setup.

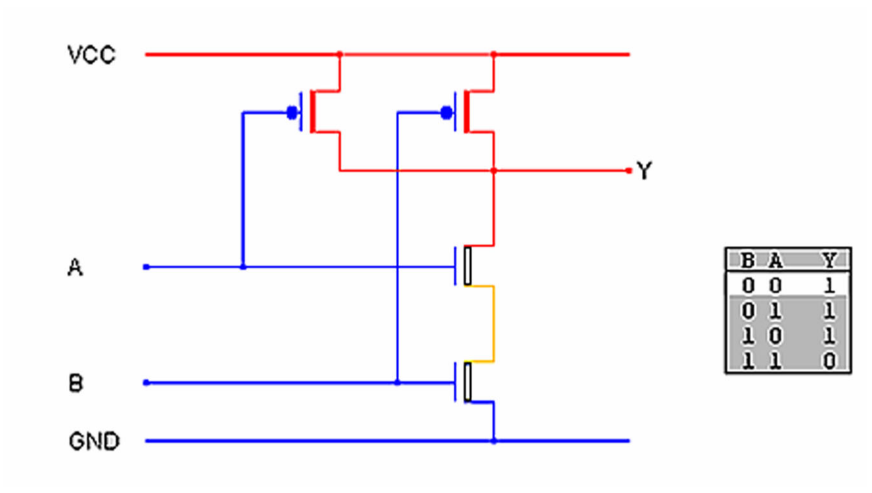
A1 Characterization

In Chapter 3.1, we described the advantages of the CMOS circuits, the foremost being its low power consumption. This characteristic of the CMOS circuit was the primary consideration when we decided to implement it in the proposed ALU's functional units.

CMOS circuits are essentially made up of MOSFET transistors. Circuit designs for basic logic gates and adder are used to characterize the simulated output propagation delays and power consumption. Cadence IC design tools were used to implement the circuits with the CSX 0.35um technology library. The circuits are simulated for output propagation delays and power consumption, which we will analyze in the next few sections.

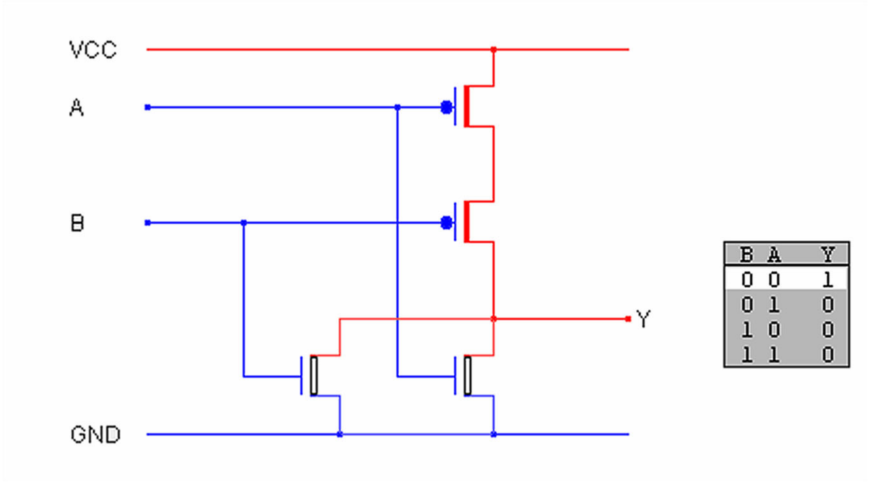
A1.1 Propagation Delay

NAND and NOR are the fundamental logic circuits that serve as basic building blocks for digital circuits. Figures A1.1 shows the 2-input NAND with its logic truth table, while Figure A1.2 shows the 2-input NOR circuit with its logic truth table.



B	A	Y
0	0	1
0	1	1
1	0	1
1	1	0

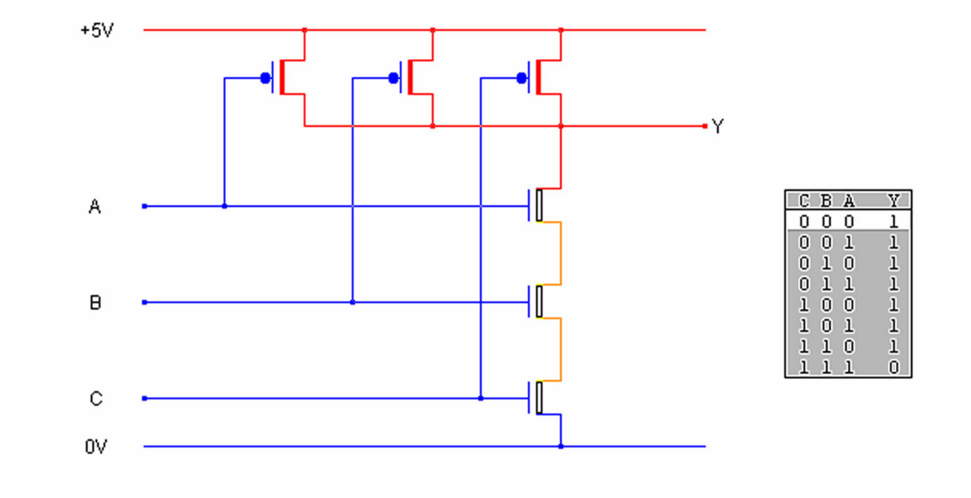
Fig A1.1 2-Input NAND gate and truth table



B	A	Y
0	0	1
0	1	0
1	0	0
1	1	0

Fig A1.2 2-Input NOR gate circuit and truth table

In general, it is possible to create a circuit with more input gates, by adding PMOS or NMOS transistors, in the correct paths, as seen in Figure A1.3 which shows a 3-input NAND circuit.



C	B	A	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Fig A1.3 3-Input NAND gate and truth table

However, while adding inputs to logic gate functions may help ease digital design implementations, it also increases parasitic capacitance along the signal path. As a result, this may affect the signal propagation delay and drive.

The simulations were performed on the logic gates we developed to investigate the effects of increasing inputs on the propagation delay. Firstly, a range of input signals – permanently high, permanently low, low-to-high, and high-to-low transitions – were connected to the logic gates in every possible combination. Propagation delay from the circuit simulation was then measured, whenever the input signals switched.

In the worst case of NAND circuit propagation delay, it was observed that worst rise time occurred when a high-to-low transition input signal was applied to the last input PIn, while the other signals were directed to the ground. Consequently, worst fall time occurred when a low-to-high transition signal was applied to all input PIns.

For the NOR circuits, worst rise times occurred when a low-to-high transition signal was applied to all input PIns. The worst fall time occurred when a low-to-high transition signal was applied to the first PIn while the rest were applied to V_{DD}. The results are summarised in Table A1.1 and Table A1.2:

NAND Input	Power (pW)	T_{rise} (ns)	T_{fall} (ns)	NAND Input	Power (pW)	T_{rise} (ns)	T_{fall} (ns)
2	1.14	0.4315	0.3493	9	3.46	0.6185	1.02255
3	1.38	0.4658	0.4184	10	3.93	0.6416	1.17833
4	1.64	0.4953	0.493	11	4.41	0.6641	1.351
5	1.91	0.5211	0.5675	12	4.89	0.6861	1.54042
6	2.19	0.5461	0.6565	13	5.38	0.7076	1.74533
7	2.58	0.5707	0.7618	14	5.88	0.7287	1.96617
8	3.01	0.595	0.8836	15	6.38	0.7494	2.20282
				16	6.88	0.7697	2.45537

Table A1.1 Worst propagation delay for NAND gate

NOR Input	Power (pW)	T_{rise} (ns)	T_{fall} (ns)	NOR Input	Power (pW)	T_{rise} (ns)	T_{fall} (ns)
2	1.01	0.4923	0.2816	9	3.19	2.4076	0.352
3	1.18	0.609	0.2933	10	3.63	2.90137	0.3606
4	1.38	0.772	0.3037	11	4.08	3.44888	0.3689
5	1.61	0.9889	0.3139	12	4.54	4.04968	0.3769
6	1.95	1.2616	0.324	13	4.99	4.7036	0.3845
7	2.35	1.58856	0.3337	14	5.45	5.41054	0.392
8	2.76	1.92025	0.343	15	5.91	6.17119	0.399
				16	6.36	6.98584	0.4057

Table A1.2 Worst propagation delay for NOR gate

The data in Table A1.1 and A1.2 are used to plot Figures A1.4 and 1.5, from which we can derive 3rd degree polynomial equations. These equations are useful for expressing worst-case propagation delays for the NAND and NOR logic circuits simulations or measurements.

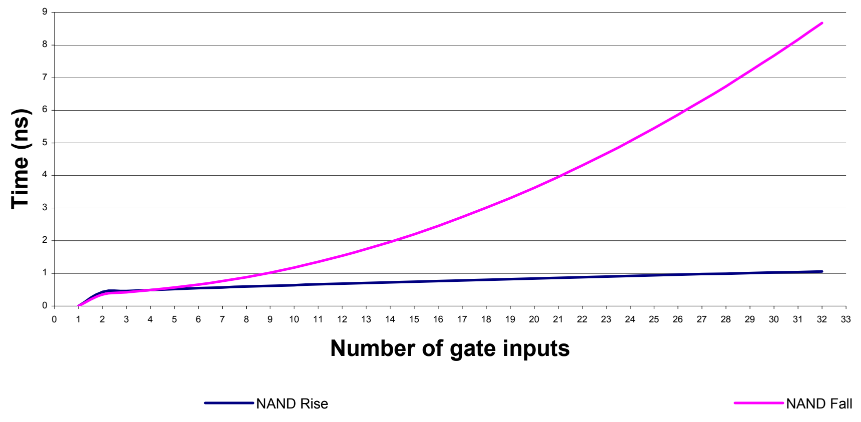


Fig. A1.4 NAND circuit worst timing plot

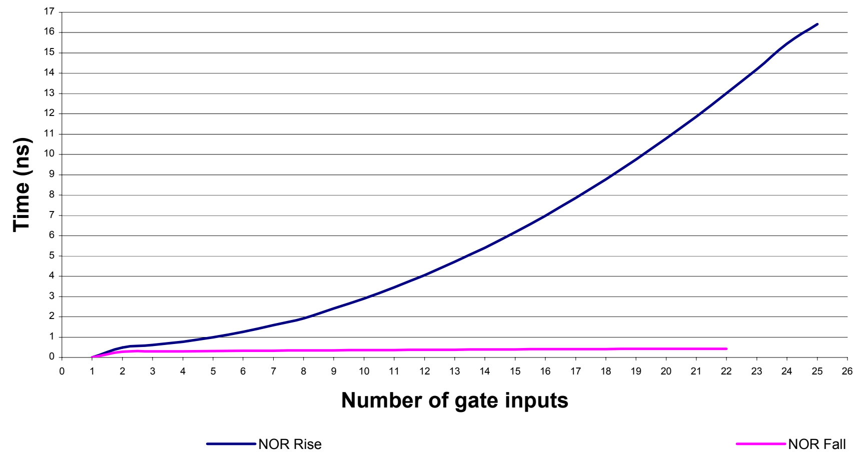


Fig. A1.5 NOR Gate Worst Timing Plot

Worst propagation delay equations for NAND,

$T_{Worst\ Rise} = 0.37942 + 0.02976z - (3.99254 \times 10^{-4})z^2 + (4.24647 \times 10^{-6})z^3$ …(Eq. A1)

$T_{Worst\ Fall} = 0.32441 + 0.00718z + 0.00782z^2 + (3.55846 \times 10^{-6})z^3$ …………...(Eq. A2)

Worst propagation delay equations for NOR,

$T_{Worst\ Rise} = 0.53498 - 0.06747z + 0.03155z^2 - (1.27038 \times 10^{-4})z^3$ ……..……(Eq. A3)

$T_{Worst\ Fall} = 0.25844 + 0.01208z - (1.9823 \times 10^{-4})z^2 + (1.15217 \times 10^{-6})z^3$ …….(Eq. A4)

Where z is the number of inputs.

A1.2 Signal Quality and Static Power Consumption

In this section, we will compare the signal quality of CMOS circuits with Pass Transistor Logic circuits using XOR circuits. The impact due to degraded signal quality on static power consumption of the Pass Transistor Logic circuits is observed.

Based on a survey on XOR circuit designs, a design with CMOS logic using 12 transistors [54] and another with Pass Transistor logic using 4 transistors [55] were selected for investigation. Both the CMOS and Pass Transistor designs were selected on the basis of the fewest transistors used, amid available circuit designs.

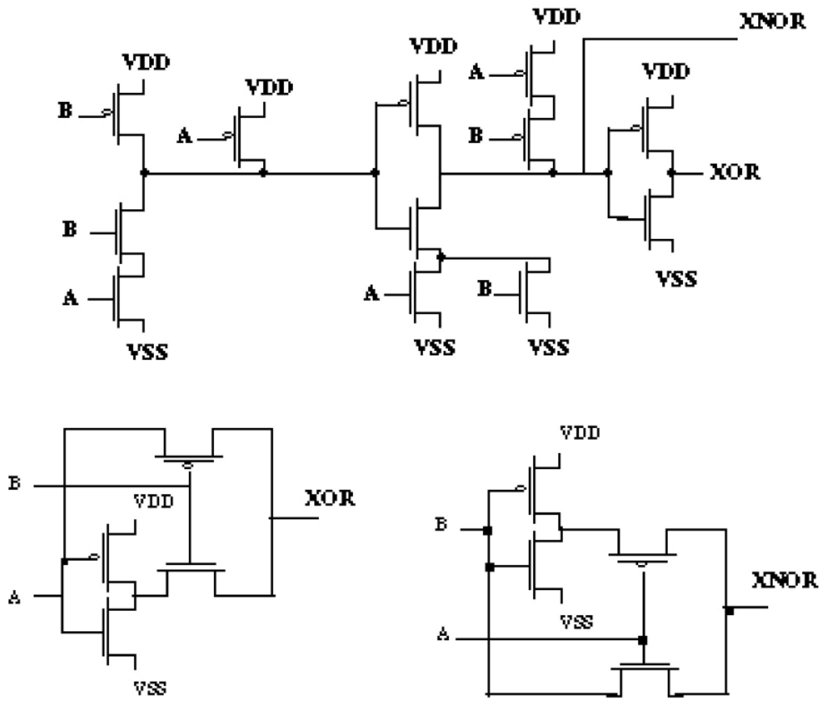


Figure A1.6 XOR Designs: 12 Transistors CMOS circuit (above) and 4 Transistors Pass Transistor Logic circuits (bottom)

Extracted from [54], "Design and analysis of low-power 10-transistor full adders using novel XOR-XNOR gates" and [55] "Design New 4-transistor XOR and XNOR designs" by Heng Tien Bui et al

Figure A1.6 shows the circuit layout for the selected designs. The 12-transistor CMOS design employs a 10-transistor XNOR circuit coupled with an inverter to provide the XOR function, while the first 4 transistors in the design are NAND gate implementations.

Such a design enables a single circuit to provide both NAND and XNOR functions – a useful feature when both NAND and XOR functions are required using the same inputs. On the other hand, Pass Transistor Logic circuits can only provide either the XOR or XNOR function. The performance for these circuits was simulated, with the results of the power consumption listed in Table A1.3.

Circuit	Transistor Count	Power (pW)
XNOR	4	2.61
XOR	4	1.81
XNOR	10	4.94
XNOR + Inverter	12	6.60
XNOR + Inverter	6	8.28
XOR + Inverter	6	5.74

Table A1.3 XOR/XNOR Static Power Consumption

As a stand alone circuit, the 4-transistor design has the lowest static power consumption because of its low transistor count. However, when connected to other circuits – like a simple inverter for instance – it results in significant static power consumption because of sub-threshold conduction that is caused by logic degradation at the output signal.

The last two rows of Table A1.3 show static power consumption caused by sub-threshold conduction, in an inverter circuit connected to the 4-transistor XOR circuit. The 12-transistor CMOS design on the other hand, does not have problems with sub-threshold conduction as CMOS logics generate rail-to-rail output signals (as mentioned in Chapter 3).

Logic degradation is a common problem found in Pass Transistor Logic circuits. This is illustrated in the circuit simulation results of the 4- transistor XOR circuit, shown in Fig A1.6.

When inputs A and B are low, the upper most PMOS is turned on by input B with its drain connected to input A. This low logic is conducted through the PMOS channel but degraded because of reverse bias in the PMOS structure (shown in Figure 3.1). This degraded low logic can turn on any connected PMOS transistors in the sub-threshold region thereby causing static power consumption.

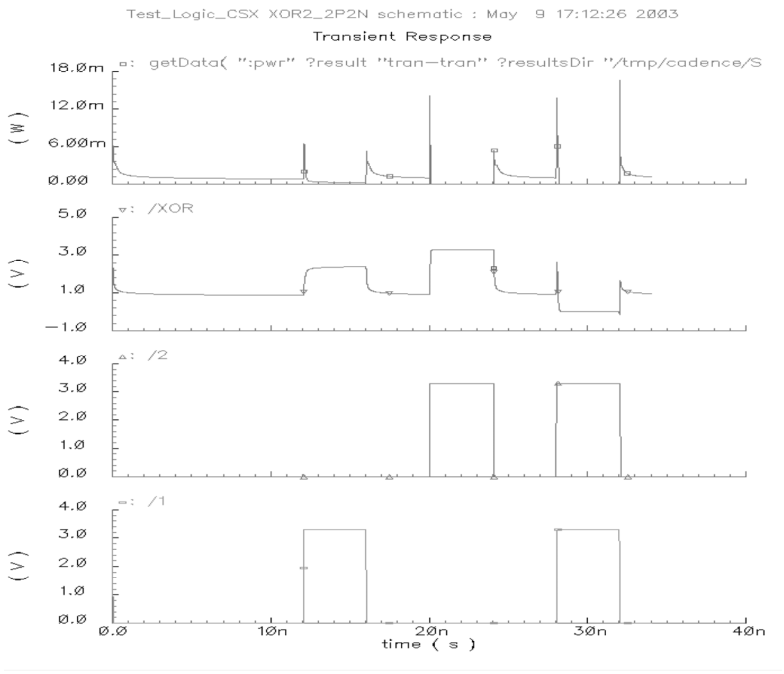


Fig. A1.7 4 Transistors XOR circuit output logic degradation

Figure A1.7 shows the electrical signals waveform obtained from the circuit simulation. From the diagram, we can see that the degraded XOR output signal at low logic is close to 1V. Even though 1V is considered low logic, it is high enough to sustain the PMOS transistor in the sub-threshold region. As in Table A1.3, the PMOS transistor in the inverter circuit conducts significant static power consumption.

As such, the 4-transistor XOR design with Pass Transistor logic is not suitable for use in low-power applications, even though they use very few transistors in the circuits. This holds true, unless the problem of logic degradation can be resolved or if the circuit connected to the XOR circuit output can withstand degraded logic signals without incurring sub-threshold power consumption.

A1.3 Static and Dynamic Power Consumption

By analyzing the simulated power consumption of the four operating blocks of Carry Look Ahead (CLA) circuit blocks carried out under controlled situations, we are able to study the static and dynamic power consumption in (0.35um) CMOS circuits.

Four blocks of the 4-bit CLA adder circuits were cascaded to form a 16-bit rippling CLA adder. A set of test bits, “1010” and “0101” were used as inputs for each block to ensure switching occurred within the circuits. Prior to investigations proper, we conducted two tests on the static and dynamic power consumption of the circuits.

In the first test, the power supply to the circuit blocks was controlled manually. In the second test, the power supply to the circuit blocks was controlled using an OR logic circuit shown in Figure A1.8. This circuit was responsible for switching off the power supply to a CLA block can be when all inputs were connected to low logic.

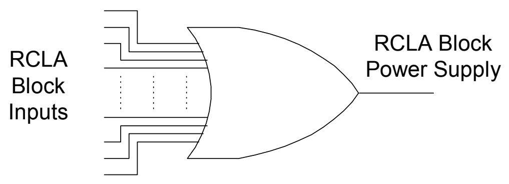


Fig A1.8 Power Control Circuit

The same procedure was applied to both tests. Initially, the inputs to all blocks were connected to the ground. Then block by block, the inputs were connected to the test bits, incrementally. The power consumption levels are recorded in Table A1.4.

Power	Power Supplied (Block)	Input (Block)	Manual Power Switch (Watt)	Circuit Power Switch (Watt)
All On	4	1	5.90057688E-18	2.77309088E-11
Off Unused	1	1	1.76085357E-18	2.77308986E-11
All On	4	2	6.28151072E-18	5.54693734E-11
Off Unused	2	2	3.52170070E-18	5.54693740E-11
All On	4	3	6.66245482E-18	8.32077997E-11
Off Unused	3	3	5.28255047E-18	8.32077985E-11
All On	4	4	7.04341062E-18	1.10946169E-10

Table A1.4 16-bit RCLA power analysis

From the power consumptions data in Table A1.9, we can conclude base on the CSX 0.35um technology library, the amount of static power consumed is negligible compared to switching power. In addition, there is a significant difference in power consumption between manual and circuit power switching.

Compared with manual switching, the power supplied to the CLA blocks in circuit power switching is controlled by OR logic circuit response to the input bits for the CLA blocks. This causes the OR logic circuit to consume dynamic switching power, which can be at least 7 orders of magnitude larger than the static power consumption of the adder circuit.

BIBLIOGRAPHY

[1] Thomas D. Burd, “Energy-Efficient Processor System Design”, Ph.D Thesis, University of California, Berkeley, 2001

[2] Thomas D. Burd and Robert W. Brodersen, “Design Issues for Dynamic Voltage Scaling”, ISLPED 2000, Rapallo, Italy, Pgs 1 – 6

[3] Thomas D. Burd and Robert W. Brodersen, “Voltage Scheduling in the lpARM Microprocessor System”, ISLPED 2000, Rapallo, Italy

[4] Woonseok Kim, Jihong Kim and Sang Lyul Min, “A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis”, Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition

[5] Pouwelse, J., Langendoen, K., and Sips, H., “Energy priority scheduling for variable voltage processors”, ISLPED 2001, Huntington Beach, CA, USA

[6] Chaeseok Im , Huiseok Kim and Soonhoi Ha, “Dynamic voltage scheduling technique for low-power multimedia applications using buffers”, ISLPED 2001, Huntington Beach, CA, USA

[7] DongKun Shin and JiHong Kim and SeongSoo Lee, “Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis”, Design Automation Conference 2001, Pgs 438 - 443

[8] C. Lee, J. Lee, T. Hwang, and S. Tsai., “Compiler Optimization on Instruction Scheduling for Low Power”, 13th International Symposium on System Synthesis, ACM, Septermber 2000

[9] Chung-Hsing Hsu, Ulrich Kremer and Michael Hsiao, “Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessor”, ISLPED’01, California USA, Aug 6 - 7 2001, Pgs 275 - 278

[10] Mansour M.M, Hajj I and Shanbhag N, “Instruction Scheduling for Low Power on Dynamically variable Voltage Processors”, 7th IEEE International Conference on Electronics, Circuits and Systems, Vol. 1, 17 – 20 Dec 2000, Pgs 613 – 618

[11] E. Musoll and J. Cortadella, “Optimizing CMOS Circuits for Low Power using Transistor Reordering”, 1996, Pgs 219 – 223

[12] A.M. Sham and M.A. Bayoumi, “A New Full Adder Cell for Low-Power Applications”, Great Lakes Symposium on VLSI '98, Lafayette, Louisiana, Pgs 45 - 49

[13] D. Radhakrishnan, “Low-voltage low-power CMOS full adder”, IEE Proc. Circuits Devices System. Vol 148, No. 1, February 2001, Pgs 19 – 24

[14] Yuke Wang; Parhi, K.K., “New low power adders based on new representations of carry signals”, Conference Record of the 34th Asilomar Conference on Signals, Systems and Computers, Vol. 2 , 2000, Pgs 1707 - 1712

[15] Youngjoon Kim; Lee-Sup Kim, “A low power carry select adder with reduced area”, The 2001 IEEE International Symposium on Circuits and Systems, ISCAS 2001, Vol. 4 , 6 - 9 May 2001, Pgs 218 - 221

[16] Issam S. Abu-Khater, Abdellatif Bellaouar and M.I. Elmasry, “Circuit Techniques for CMOS Low-Power High Performance Multipliers”, IEEE Journal of Solid State Circuits, Vol. 31, Oct 1996, Pgs 1535 – 1546

[17] Yuke Wang, YingTao Jiang and Edwin Sha, “On Area-Efficient Low Power Array Multipliers”, Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on, Vol. 3, 2 - 5 Sep 2001, Pgs 1429 – 1432

[18] Issam S. Abu-Khater, Abdellatif Bellaouar and M. I. Elmasry, “Circuit Techniques for CMOS Low-Power High-performance Multipliers”, IEEE Journal of Solid-State Circuits, Vol. 31, 10 Oct 1996, Pgs 1535 – 1546

[19] Wei-Chung Cheng, Jian-Lin Liang and Massoud Pedram, “Software-Only Bus Encoding Techniques for an Embedded System”, Proceedings of the 15th International Conference on VLSI Design, 2002

[20] L. Kurian-John, V. Reddy, P. Hulina and L. Coraror, “A Comparative Evaluation of Software Techniques to hide Memory Latency”, Proceedings of the 28th Hawaii International Conference of System Science, Jan 1995, Pgs 229-238

[21] Parik A, Kandemir M, Vijaykrishnan N and Irwin M.J, “Instruction Scheduling Base on Energy and Performance Constraints”, Proceedings IEEE Computer Society Workshop VLSI, 27-28 April 2000, Pgs 37 – 42

[22] Xu W, Parik A, Kandemir M, and Irwin M.J, “Fine-grain Instruction Scheduling for Low Power”, IEEE Workshop on Signal Processing Systems, 16-18 Oct 2002, Pgs 258 – 263

[23] Cheol-Ho Jeong, Woo-Chan Park, Sang-Woo Kim, Tack-Don Han, and Moon-Key Lee, “In-Order Issue Out-of-Order Execution Floating-Point Coprocessor for CalmRISC32”, IEEE 15th International Symposium on Computer Arithmetic, June 2001, Pgs 195 – 200

[24] S. Abraham and K. Padmanabhan, “Instruction reorganization for variable-length pipelined microprocessor”, Proceedings of the International Conference on Computer Design, New York, October 1988

[25] Hily. S and Seznec. A, “Out-of-Order execution may not be the cost-effective on processors featuring simultaneous multithreading”, 5th International Symposium on High-Performance Computer Architecture, 9 – 13 Jan 1999, Pgs 64 – 67

[26] Jessica H. Tseng and Krste Asanovic, "Banked Multi Ported Register Files for High-frequency Superscalar Microprocessors", Proceedings of the 30th International Symposium on Computer Architecture, San Diego, California, 2003, Pgs 62-71

[27] J. L. Cruz, A. Gonzalez, M. Valero and N. P. Topham, "Multiple Banked Register File Architecture", Proceedings of the 27th International Symposium on Computer Architecture, San Diego, California, 2000

[28] Kai Hwang, "Advanced Computer Architecture: Parallelism, Scalability and Programmability", McGraw Hill, Inc

[29] Thomas D. Burd, Trevor A. Pering, Anthony J. Stratakos, and Robert W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System", IEEE Journal of Solid-State Circuits, Vol. 35, Nov 2000, Pgs 1571 – 1580

[30] Vivek Tiwari, "Instruction Level Power Analysis and Optimization of Software", Journal of VLSI Signal Processing Systems, Vol. 13, Aug 1996

[31] J.P. Grossman, "Cheap Out-of-Order Execution using Delayed Issue", IEEE International Conference on Computer Design: VLSI in Computers & Processors Austin, Texas, September 17 - 20, 2000

[32] Thomas D. Burd, "Energy-Efficient Processor System Design", Ph.D Thesis, University of California, Berkeley, 2001

[33] R. Zimmermann and W. Fichtner, “Low-Power Logic Styles: CMOS Versus Pass-Transistor Logic”, IEEE Journal of Solid State Circuits Vol. 32, Pgs 1079 – 1089

[34] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw and T. Mudge, “Drowsy caches: simple techniques for reducing leakage power”, Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on , 25-29 May 2002, Pg 148 – 157

[35] J. Rabaey, Digital Integrated Circuits, A Design Perspective, Prentice Hall, Upper Saddle River, NJ, 1996.

[36] R. Muller, T. Kamins, Device Electronics for Integrated Circuits, Wiley, New York, 1986.

[37] S. Sze, Physics of Semiconductor Devices, Wiley, New York, 1981.

[38] James M. Lee, Verilog QuickStart: A Practical Guide To Simulation and Synthesis in Verilog

[39] Digital Standard Cell Datasheets for AMS C35 Standard Cell Library, http://asic.austriamicrosystems.com/databooks/index_c35.html

[40] Amos R. Omondi, “Computer Arithmetic Systems – Algorithms, Architecture and Implementation”, Prentice Hall

[41] Stuart F. Oberman and Michael J. Flynn, “Division Algorithms and Implementations”, IEEE Transactions on Computers, Vol. 46, August 1997, Pgs 833 – 854

[42] Hung. P, Fahmy. H, Mencer. O and Flynn. M.J, “Fast division algorithm with a small lookup table”, Conference Record of the 33rd Asilomar Conference on Signals, Systems and Computers, Vol 2, 24 – 27 Oct. 1999, Pgs 1465 - 1468

[43] Ing-Jer Huang and Alvin Despain , “An Extended Classification of Inter-instruction Dependency and Its Application in Automatic Synthesis of Pipelined Processors”, Proceeding of 26th International Symposium on Microarchitecture, Dec 1993

[44] Augustus K. Uht, “Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream”, IEEE Transactions on Computers Vol. 41, July 1992

[45] Sunghyun Jee, Kannappan Palaniappan, “Dynamically Scheduling VLIW Instructions with Dependency Information”, Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures, 2002

[46] M.F. Chang, Y.K. Chan, “Parallel Execution of Multiple Sequential Instruction Streams”, Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, USA. IEEE Computer Society Press, 1- 4 December, 1993

[47] Gurindar S. Sohi, "Instruction Issue Logic for High-Performance Interruptible, Multiple Functional Unit", pipelined computers, IEEE Transactions on Computers Vol. 39, Mar 1990

[48] Tai M. Chung, Hank G. Dietz, "Static Scheduling of Hard Real-time Code with Instruction-Level Timing Accuracy", Third International Workshop on Real-Time Computing Systems Application , Seoul, Korea, 1996

[49] S. Abraham and K. Padmanabhan, "Instruction reorganization for variable-length pipelined microprocessor", Proceedings of the International Conference or, Computer Design, New York, October 1988

[50] Q. Zhao, T. Basten, B. Mesman, "Static resource Models of Instruction Sets", ISSS 01, Oct 1-3 2001, Montreal Quebec, Canada.

[51] Q. Zhao, B. Mesman and T. Basten, "Practical Instruction Set Design and Compiler Retargetability Using Static Resource Models", Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition

[52] Pradeep K. Dubey, George B. Adams III and Micheal J. Flynn, "Instruction Window Size Trade-Offs and Characterization of Program Parallelism", IEEE Transaction on Computers Vol. 43, April 1994, Pgs 431 – 442

[53] Allen Leung, Krishna V. Palem and Cristian Ungureanu, “Run-time versus Compile-time Instruction Scheduling in Superscalar (RISC) Processors: Performance and Tradeoffs”, Journal of Parallel and Distributed Computing, Vol. 45, 1997, Pgs 13 – 28

[54] Heng Tien Bui, Yuke Wang and YingTao Jiang, “Design and analysis of low-power 10-transistor full adders using novel XOR-XNOR gates”, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 49, Jan 2002, Pgs 25 – 30

[55] Heng Tien Bui, Al-Sheraidah A. K. and Yuke Wang, “New 4-transistor XOR and XNOR designs”, Proceedings of the 2nd IEEE Asia Pacific Conference on ASIC, Aug 2000, Pgs 25 - 28