Evaluating the Impact of Designing and Testing of High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms

Researcher
Rafia Mukhtar

Supervisor
Dr. Afsah Imtiaz

ASES PUBLISHING

Evaluating the Impact of Designing and Testing of High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms

Researcher

Rafia Mukhtar

Roll No: S21BDOCS3E01129

MS (CS) Specialization in Information Technology

Supervisor

Dr. Afsah Imtiaz

Assistant Professor

Department of Information Technology

Session: 2021-2023

DEPARTMENT OF INFORMATION TECHNOLOGY

The Islamia University of Bahawalpur

ASES PUBLISHING

Evaluating the Impact of Designing and Testing of High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms

Basım Tarihi: 26. 02. 2024

ISBN: 978-625-95806-3-0

A thesis presented to
The Islamia University of Bahawalpur

In partial fulfillment
Of the requirement for the degree of

MS (CS) Specialization in Information Technology

By

Rafia Mukhtar

S21BDOCS3E01129

SPRING 2021-2023

ASES YAYINCILIK

Sertifika No: 63715

DEDICATION

To ALLAH the Almighty &

To My Parents & Family

DECLARATION

I hereby certify that I worked on the projects detailed in this dissertation under the guidance of **Dr. Afsah Imtiaz,** an assistant in the Department of information technology at the Islamia University of Bahawalpur, Pakistan.

Moreover, I hereby attest that neither the main body of this dissertation nor its concurrent submission for any other degree have been submitted anywhere else.

I additionally affirm that the dissertation was written by me and, where appropriate, that I have acknowledged the contribution of others. It contains the findings of my own research or advanced study.

Rafia Mukhtar
S21BDOCS3E01129

SUPERVISOR'S DECLARATION

It is hereby certified that work presented by **Rafia Mukhtar** in the thesis titled "***Evaluating the Impact of Designing and Testing of High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms***" is based on the findings of the investigation that the applicant under my supervision completed. To the best of the author's knowledge, no material other than his own work has been utilized in this thesis, with the exception of instances in which appropriate attribution has been given, for a higher degree at this university or any other institution of learning. She is eligible to present this thesis in partial fulfillment of the prerequisites for the degree of Master of Information Security at the Department of Information Technology, The Islamia University of Bahawalpur since she has met all of them.

_____

Dr. Afsah Imtiaz
Assistant Professor
Department of Information Technology
The Islamia University of
Bahawalpur Pakistan

FINAL APPROVAL

It is certified that we have gone through this dissertation titles **"*Evaluating the Impact of Designing and Testing of High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms*"** that is submitted by **Rafia Mukhtar**, MS CS(IT) Roll No: S21BDOCS3E01129 Session Spring 2021-2023. The work that serves as the basis for the thesis is therefore determined to be original. We came to the conclusion that this thesis meets the requirements for The Islamia University of Bahawalpur to accept it for the MS/MPhil degree in information technology.

DISSERTATION COMMITTEE

EXTERNAL EXAMINER

_____

INTERNAL EXAMINER

_____

Dr. Afsah Imtiaz
Assistant Professor
Department of Information Technology
The Islamia University of Bahawalpur
Pakistan

CHAIRMAN OF DEPARTMENT

_____

Dr. Dost Muhammad Khan
Chairman
Department of Information Technology
The Islamia University of Bahawalpur
Pakistan

ACKNOWLEDGEMENTS

Firstly, I am immensely grateful to Almighty **Allah** and the **Holy Prophet Muhammad P.B.U.H**. for providing me with the strength and support to accomplish my lifelong aspirations, which once seemed like mere dreams.

I also extend my heartfelt appreciation to my parents, family, and supervisor**, Dr. Afsah Imtiaz**, for their unwavering encouragement and guidance throughout my academic journey. Their valuable support has played an indispensable role in shaping me into the person I am today, and for that, I am forever grateful.

Rafia Mukhtar
S21BDOCS3E01129

ABSTRACT

In this research, the researcher presented pipelining of functional units in superscalar and DSP Processors, using Verilog hardware description language (VHDL). The purpose of this Research was to increase the efficiency and computation performance of superscalar and DSP Processors making it cost effective due to the parallel execution of tasks. Previously work done regarding this has produced serial execution of instructions that used to take much more time in the execution of single instruction and after completion of that instruction, next instruction is to be executed, which simply produces time load in Superscalar and DSP processors. To resolve this matter much hardware (registers, adders, Multiplexers, decoder, encoders) was used. Although it improved computing performance yet it became costly and became out of range of pocket. Through this research, the researcher aimed to produce such an architecture that not only increases computation performance but also is cost effective as well. Parallel processing has been introduced through which we break the process into a series of steps. Each step transforms the previous step through refinement. The investigation was concluded with a summary of the researcher's conclusions, comments, and recommendations.

Key Words: Pipelining, Superscalar, High Performance, DSP Processor, Execution

TABLE OF CONTENTS

LIST OF FIGURES

Figure No. **Page No.**

CHAPTER 1

INTRODUCTION

1.1 Background

Pipelining is utilized to provide processing time improvements and higher processor throughput without needing a significant quantity of additional hardware that is not feasible with current non-pipelined technologies. Like other types of architecture, computer architecture is the skill of identifying a user's demands and then creating a structure to best suit those goals while staying within a certain budget and set of available technologies. (Jeffrey Fred P. Brooks, 1962)

Cache memory technology and various types of parallelism at the instruction level are only a few of the speedup strategies used by modern CPUs. Such parallelism may be found in the Data Processing Unit's (DPU) internal structure or in the overlap of tasks performed by the DPU and Program Control Unit (PCU). These characteristics increase the complexity of the CPU. The straightforward CPU makes clear the significant possibilities for parallel processing at the instruction level. The primary PCU and DPU tasks are carried out at various clock cycles. If they don't share a resource, like the system bus, then these tasks could be finished simultaneously. In other words, the PCU may fetch the next instruction while the DPU is handling the current command. This overlap of receiving and processing instructions is an illustration of command pipelining, a key component of RISC processor speedup. Fig. 1.1 provides a visual representation of the two-stage pipelining type. A fetch stage, which is primarily handled by the PCU, and an execution stage, which is primarily carried out by the DPU, can be thought of as two processing stages that each instruction goes through in succession. This allows for the processing of two instructions at once during each CPU clock cycle, with the first instruction finishing its fetch phase and

the second instruction finishing its execute phase. Therefore, two-stage pipelines may quadruple the CPU's performance by switching from one instruction every two clock cycles to one instruction per clock cycle.

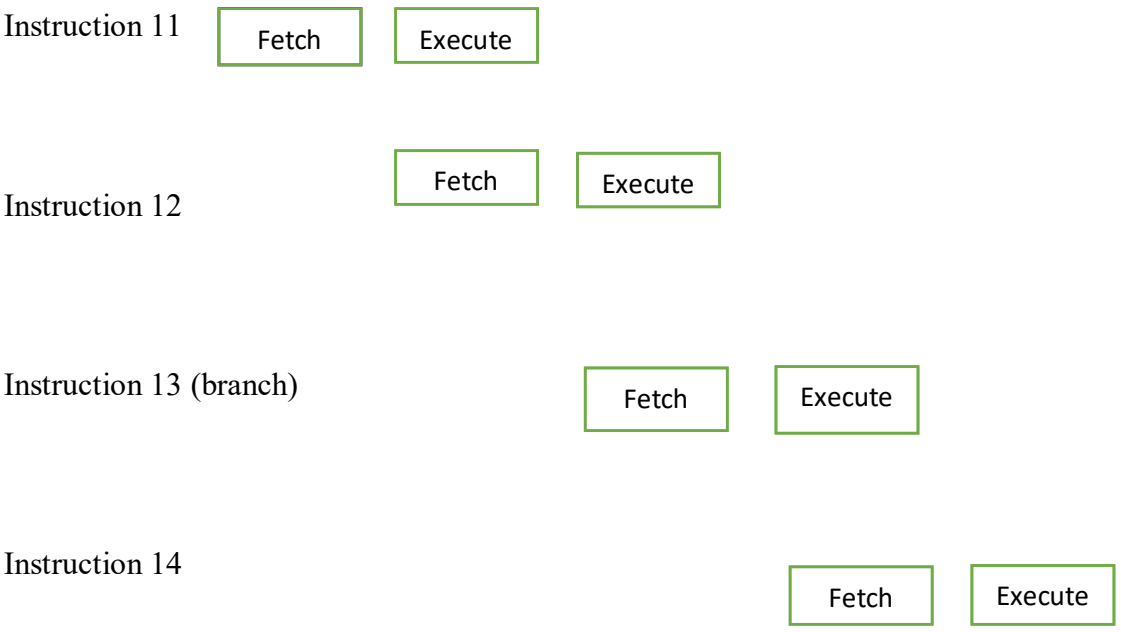


Figure 1.1: Overlapping instruction in a two-stage instruction pipeline

1.2. Problem Statement

Pipelining is frequently equated to an assembly line in production, when many components of an item are put together simultaneously. Hence the present study is an intention to evaluate the impact of designing and testing high performance pipeline functional units by using Verilog hardware description language about superscalar and DSP processors and their usage in complex DSP algorithms.

1.3. Research Questions

The following questions were addressed by the present study:

RQ1. How to increase efficiency and .computational performance of superscalar and DSP Processors making it cost effective due to the parallel execution of tasks?

RQ2. How to determine the need of a user of structure and designing to meet those needs as effectively as possible?

1.4. Research Objectives

The following goals were considered when this study was being conducted:

i. To increase efficiency and .computational performance of superscalar and DSP Processors making it cost effective due to the parallel execution of tasks.

ii. To produce such an architecture that not only increases computational performance but also its cost effective to obtain improvements in processing time and boosting the CPU without adding a lot of additional gear.

iii. determine the need of a user of structure and designing to meet those needs as effectively as possible.

1.5 Research Scope

In this research work, the researcher has just worked on Evaluating the Impact of Designing and Testing High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms. We just discussed a simple e-learning system and its quality. In future more work can be done on it and this system can be developed. By developing this system performance can be improved in future.

1.6. Research Significance

The following are some reasons why the research will be useful:

1. This would only be a meager contribution to the body of current knowledge. Due to the concurrent execution of activities, this research will help boost the productivity and

computing performance of superscalar and DSP processors, enabling them to be more cost-effective.

2. It would be useful for this research to investigate how best to create a structure in order to satisfy a user's demands.

1.7. Thesis Organization

This research thesis comprises of six chapters including this chapter. The summaryof the remaining chapters is provided as follows:

- **Chapter Two: Literature Review**

The second chapter is concerned with reviewing the literatures in order to gain knowledge about the existing research of Designing and Testing High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms.

- **Chapter Three: Research Methodology**

Third chapter describe the research methodology to achieve the objectives of the study. The research methodology encompasses the design of pipelining for Superscalar and DSP processors. Unlike other approaches, it attacks the processor design problem at a coarser level of granularity, at the execution unit level. This is a reasonable approach because large tasks are subdivided into smaller subtasks of equal time duration and it is possible to do all of them simultaneously.

- **Chapter Four: Results of Proposed Study**

This chapter discusses the outcomes achieved from the study conductedabout Designing and Testing High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex

DSP Algorithms.

- **Chapter Five: Model/Framework Evaluation**

  This chapter presents the improved proposed model of Designing and Testing High Performance Pipeline Functional Units by Using Verilog Hardware Description Language about Superscalar and DSP Processors and their Usage in Complex DSP Algorithms.

- **Chapter Six: Conclusions and Future Work**

  This chapter concludes the finding by achieving the objectives of the study. This also highlightedthe contributions and limitations of the study that is followed by the future directions in related field.

CHAPTER 2

LITERATURE REVIEW

2.1. Definition of Pipeline

A pipeline is a collection of phases, each of which requires some labor. The task is not complete until each step has been completed. The separation of a major work into smaller, overlapping tasks is the key of pipelining. Computers use the term "pipeline" to describe the constant, somewhat overlapping flow of instructions to the processors or the mathematical computations the CPU does to execute an instruction. A computer processor would have to read the first data from memory, execute it, then read the second instruction from memory, and so on, in the absence of a pipeline. The arithmetic part of the processor is not in use while the command is being retrieved. It must wait until it is given the next instruction. The pipelining capability of the computer architecture allows for the acquisition of the gets hard while the system is performing an arithmetic operation. The next instruction is held in a buffer near the CPU until each instructions operation is complete. Instruction fetching makes advantage of continuous staging. As a result, more instructions may be carried out in a given amount of time.

While ultimately certain parts would need to be finished before others, pipelining is sometimes compared to an industrial assembly line where several components of a product are being built at once. Whether or whether there is a certain sequential reliance, the whole process may benefit from those operations that may operate concurrently. Pipelining is now essential to creating a fast processor. A pipeline is similar to an assembly line in that each stage completes a portion of the overall task in both. Installing seat coverings is one of the smaller activities carried out by workers on an automobile production line. The assembly line's strength stems from producing several automobiles each day. In the time

it takes to complete one of the several processes on a balanced assembly line, a new automobile comes off the line. The assembly line increases the number of vehicles being manufactured concurrently, which improves the pace at which the cars are started and finished but does not shorten the time it takes to build a single car.

An instructional pipeline and an algebraic pipeline are two categories of computer processor pipelining. The phases that an instruction goes through while being processed, such as when it is retrieved, sometimes buffered, and then executed, are represented by the instruction pipeline. The arithmetic pipeline represents the parts of an arithmetic that may be divided up and overlapping while being carried out. In addition, pipelines and memory controllers are used in computers to transfer data among different memory staging locations.

A pipeline processor is made up of a series of m data processing circuits, sometimes known as stages or segments, that work together to carry out an action on a stream of data operands. Each step includes some processing, but a complete result can only be reached once an operand set has gone through the full pipeline. A stage si in fig. 1.2 has a combinational data route circuit Ci and a multiword input register or latch Ri. The Ri's act as buffers to stop nearby stages from interfering with one another and keep partly processed results as they pass through the pipeline. The Ri s undergoes synchronous state change in response to a shared clock signal. Except for R1, which obtains its data from an outside source, each Ri is fresh set of input data Di-1 from the stage Si-1 before it. The computations made by Ci-1 during the previous tick of the clock are represented by Di-1. After loading Di-1 into Ri, Ci uses Di-1 to calculate a new data set Di. As a result, each stage computes a fresh set of results for each clock period and transfers its prior results to the stage before it.

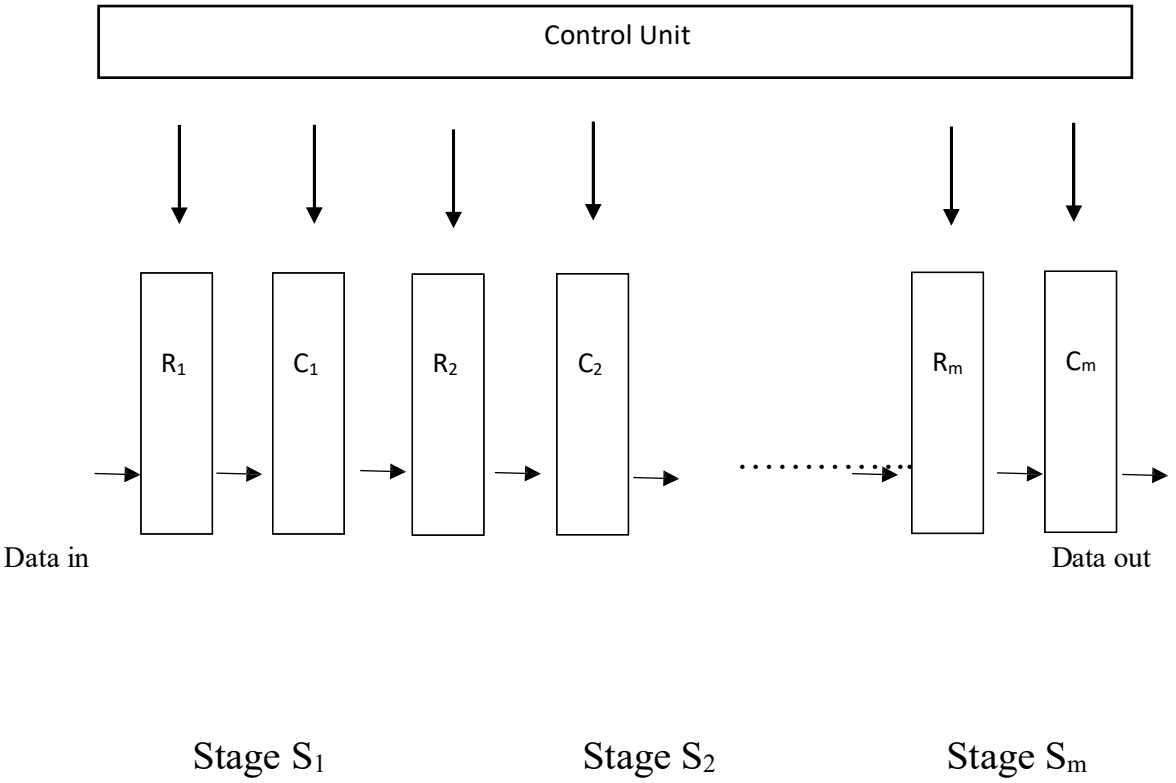


Figure 2.1: Structure of a pipeline processor

A pipeline first seems to be an expensive and cumbersome method of carrying out the desired task. An m-stage pipeline's benefit is that it can handle up to m different sets of data operands at once. As a result of these data sets moving through the pipeline in stages, when it is full, m multiple processes are running simultaneously, each at a different step. Every clock cycle, a fresh final result comes out of the pipeline.

The directive is broken down as follows:

I: Fetch instruction (FI): Into a buffer, read the subsequent anticipated command.

2. Decode Instruction (DI): To decode an instruction, you must first identify its opcodes and operand specifies.

3. Calculate operands (CO): each source operand's specific address. It may be necessary to do calculations for address properties such as displacement, indirect, implicit, or others.

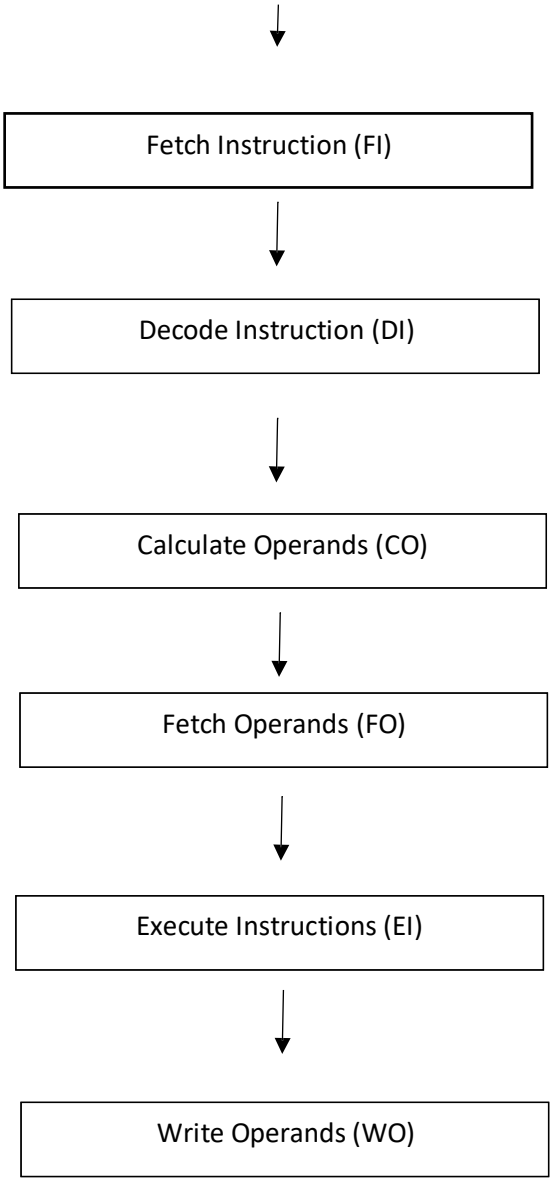


Figure 2.2: Pipelined execution of a single instruction time

4. Fetch Operands (FO): Retrieve each operand. No need to retrieve operands that are in registers.

5. Execute Instruction (EI): Execute the provided operation, and if necessary, store the result in the given destination operand location..

6. Write operand (WO): Save the result to memory.

With this split, the time of the different phases will be more evenly distributed. Every instruction passes through each of the pipeline's six steps.

It won't always be this this. For instance, the WO stage is not required for a load command. Nevertheless, the timing is built up assuming that each instruction needs all six steps in order to simplify the pipeline hardware. It's specifically expected that there aren't any memory conflicts. For instance, a memory access is required at the FI, FO, and WO phases. The majority of memory architectures forbid these accesses from happening at the same time.

Let's suppose for the sake of example that different phases will last about equal amounts of time. Nine instructions may be executed in only 14 time units instead of 54 time units using a six-stage pipeline.

First, we need to figure out what occurs throughout each machine clock cycle. Every clock cycle activates every step of the pipe. This necessitates the ability for any combination of activities to happen simultaneously and the completion of all operations on a pipe stage in a single clock. The most crucial ones for the data route are listed below.

1. The PC must be used for every clock. Instead of ID, this must be done. There must be an extra incrementer for this. The ALU cannot be utilized to increase the PC since it is already in use throughout every cycle.

2. A fresh instruction is sent on each clock, as in IF.

3. A fresh data word must be accessible on each clock cycle; MEM handles this.

4. As both a data access and an instruction take place on every clock, there needs to be a distinct MAR for each (IMAR and DMAR).

2.2. Concept of Superscalar Processors

The word "superscalar," which was first used in 1987, describes a device built to increase the speed at which scalar instructions may be executed. The superscalar approach is the following advancement in the construction of sophisticated general-purpose processors. A CPU with many independent instruction streams is referred to as a superscalar processor. Each pipeline has many phases, allowing it to process several instructions at once. A further degree of parallelism is introduced by numerous pipelines, allowing for the simultaneous processing of many streams of instructions. Instruction-level parallelism, which specifies the degree to which instructions in a program may be executed simultaneously, is a technique used by superscalar processors.

A superscalar processor often receives a high number of instructions all at once before looking for neighboring instructions that may be performed in parallel and are independently of one another. Since one instruction's input relies on another's output, the later instruction cannot finish its execution concurrently with or before the former instruction. The processor may begin and finish instructions in a different sequence than the machine code if such dependencies have been detected.

By adding new registers and modifying register pointers in the original code, the processor may get rid of certain unnecessary dependencies. Common activities like loads, stores, conditional branching, and integers and floating-point arithmetic may be initiated simultaneously and completed individually in a superscalar implementation of the processing architecture. These implementations bring up a variety of intricate design concerns with regard to the instruction flow.

The capacity to independently execute instructions in many pipelines is the core of the superscalar method. By enabling instructions to be executed in an order distinct from the

program sequence, the notion may be used even more effectively. The superscalar technique is shown in broad terms in Figure 2.1.

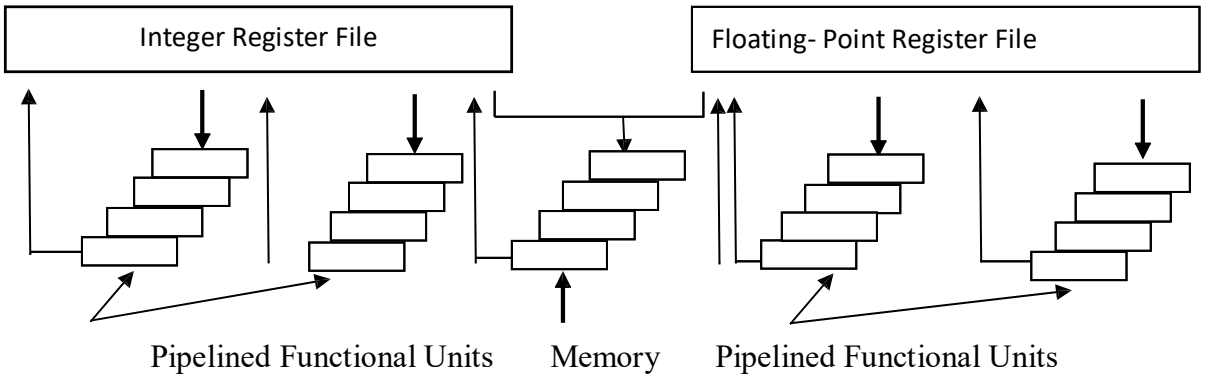


Figure 2.3: General Superscalar Organization

There are several functional units that enable the concurrent execution of numerous instructions, each of which is implemented as a pipeline. This example allows for the simultaneous execution of two integers, two numbers, and a memory action (either load or store).There is some potential for performance increase using superscalar-like computers, according to the work of several researchers.

2.3. High Performance Processor Components

Putting it simply, completing a program more quickly. The period required to receive and execute an instruction is known as its latency. Your options are: I Reduce the latencies of individual instructions or ii) execute more instructions concurrently to speed up the execution of a collection of instructions.

While the latter distinguishes superscalar processor implementations, preventing rising instruction latencies due to an increase in hardware complexity brought on by the requirement for higher parallelism is a fundamental challenge in superscalar design.

To process the instructions simultaneously, it is necessary to identify the relationships between them. It also needs adequate hardware, methods for figuring out if an operation

is ready to go, and methods for passing data from one action to another. The illusion of sequential execution must be preserved even after the outcomes of instructions have been committed and the machine's apparent state has changed. This demonstrates that a superscalar processor implements a variety of features, including: I strategies for snagging two or more tasks at once, frequently by anticipating the outcomes of provisional instructions and fetching beyond them; ii) techniques for identifying the true interconnections involving registers; and iii) processes for sending these values to the areas where they are required during execution.

Resources for the simultaneous execution of multiple instructions in parallel, such as a number of pipelined functional units and memory structures with multiple memory reference support, are also covered in Part II.

iv) Techniques for sending data values through memory that make use of load and store instructions as well as memory interfaces that allow memory hierarchies to perform dynamically and frequently in unexpected ways.The strategies utilized to carry out the instructions must be correctly matched with these interfaces.

v) Methods for accurately recording the status of the process; these methods maintain the appearance of sequential execution to outsiders.

Recent superscalar processors include:

- MIPS R10000
- DEC 21164
- AMD K5

2.4. DSP Devices

Applying mathematical procedures to signals that are represented digitally is known as digital signal processing. Any electrical system that converts digital signals into digital signals in accordance with an algorithm is a digital signal processing system. Digital signals often reflect physical system signals that relate to physical time. DSP systems, and real-time DSP systems in particular, are composed mostly of data-driven behavior that is repeatedly applied and is defined by a mathematical algorithm while being subject to severe temporal constraints.

Timing and inaccuracy are the two primary criteria for DSP applications. DSP processors provide features to increase processing speed and accuracy (see 2.2.1 DSP Algorithms and System Applications).

2.4.1 Speech coding

1. Digital Cellular Telephones
2. Personal Communication systems
3. Digital Cordless Telephones
4. Multimedia Computers
5. Secure communications

2.4.2 Speech Encryption and Decryption

1. Digital Cellular
2. Telephones
3. Personal Communication systems
4. Digital Cordless Telephones
5. Secure Communications

2.4.3 Speech Recognition

1. Advanced user interface
2. Multimedia Workstations
3. Robotics
4. Automotive Applications
5. Digital Cellular Telephones
6. Personal Communication Systems
7. Digital Cordless Telephones

2.4.4 Speech Synthesis

1. Multimedia PCs
2. Advanced User Interfaces
3. Robotics

2.4.5 Image Compression

1. Digital Photography
2. Digital Video
3. Video over Voice
4. Consumer Video etc

2.4.6 Advantages of DSP Processors

- Functions implement able that are expensive or impractical in analog
- Insensitivity to environment
- Insensitivity to component tolerance

- Predictable, repeatable behavior
- Reprogram ability
- Size

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Introduction

The methods used for the research are detailed in this chapter. It provides a detailed explanation of the study's methodology and research strategy.

3.2 Proposed Research Methodology

In this research study, the researcher proposed a methodology for the design of pipelining for Superscalar and DSP processors. Unlike other approaches, it attacks the processor design problem at a coarser level of granularity, at the execution unit level. This is a reasonable approach because large tasks are subdivided into smaller subtasks of equal time duration and it is possible to do all of them simultaneously.

The purpose of this research is to increase the efficiency and computational performance of superscalar and DSP processors making it cost-effective due to the parallel execution of tasks.

3.2.1 Architecture of Adders

Many information processing tasks are carried out by digital computers. The different arithmetic operations are among the fundamental operations that are used. The addition of binary bits is unquestionably the most fundamental mathematical operation.

The design of adders, such as the Bit Serial Adder, is covered in this section.

Carry Look Ahead Adder, Pipelined Ripple Carry Adder, and Ripple Carry Adder.

- Carry propagation with Look Ahead Adder.
- Floating Point Adder with Pipeline.
- Keep Save Adder on you.

3.3. Bit Serial Adder

Cascaded circuits with numerous flip-flops make comprise a sequential circuit. The cognitive or information processing portion of the circuit is provided by the combinational 1ogic. The inner data Y of the gadget is determined by the information that the flip-flops hold about the circuit's previous activity. Both X and Y are variables of Z if X and Y are the primary input and Z is the primary output. A precisely controlled clock signal often regulates the intervals at which flip-flops change states in a sequential circuit intervals at which switch change states in a linear chain are often regulated by a precisely controlled clock signal. State Y and primary output Z of the circuit have the capacity to vary with each clock tick. The behavior of a linear chain may be described using a state table that lists the potential principal output values and internal states. A bit serial adder in Fig. 4.1 is created to add two unregistered binary values of any length, X1 and X2, in order to get their sum Z=X1 + X2. The numbers are given out in serial form, or bit by bit, and the outcome is likewise generated in serial form. The serial adder only has one flip-flop in its memory since it only has two internal states, and that flip-flop stores the value variable Y. There are only two strategies that may be used to allocate 0s and 1s to Y. Assigning the natural states, where So has Y=0 and S1 has Y=1, is what we do. The equation Y (i+1) = D may be used to explain the behavior of the flip-characteristic flop (i). The serial adder has to be reset to the S0 state before inputting the last two digits to be added. Giving the switch flop's clear (CLR) input a reset pulse is the simplest method to do this.

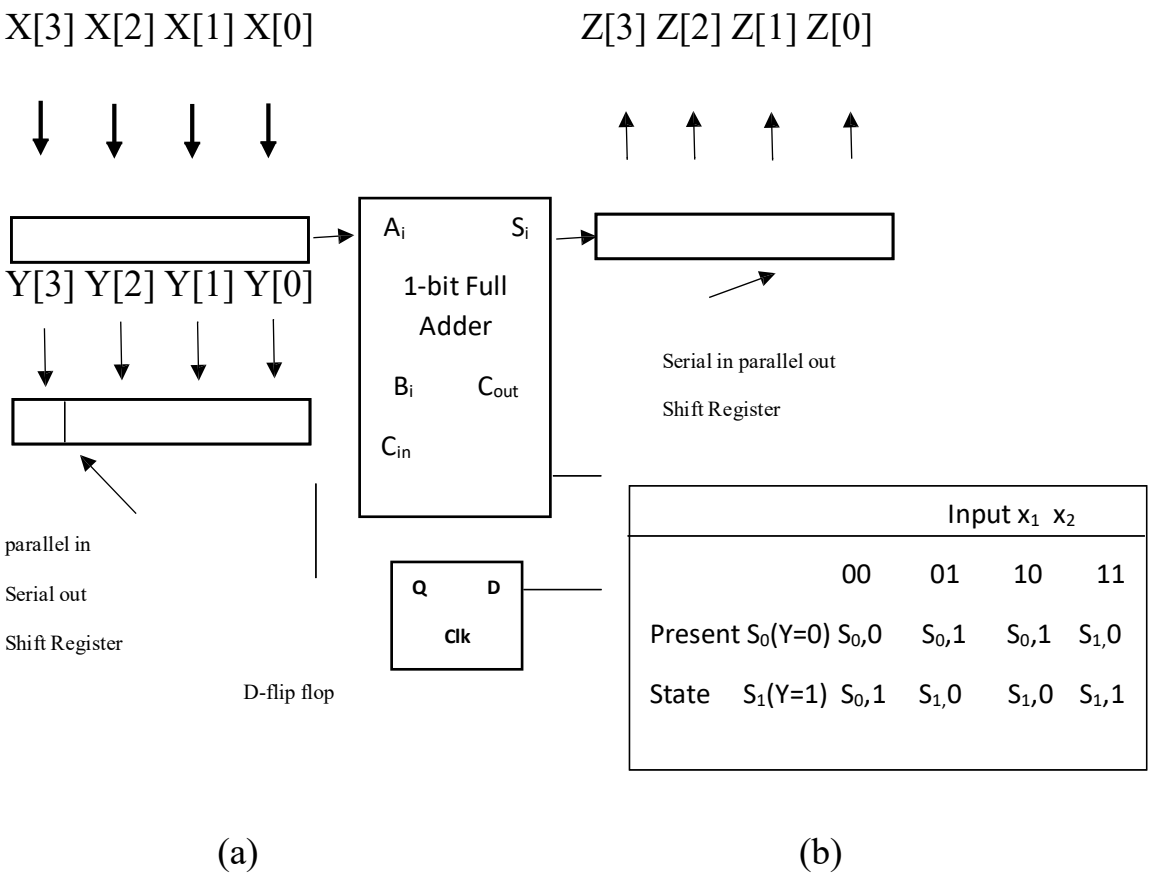


Figure 3.1: (a) Logic circuit for a bit serial adder; (b) State table

Example:

A = 0011, B = 0111

A	B	C_{in}	C_{out}	Sum
1	1	0	1	0
1	1	1	1	1
0	1	1	1	0
0	0	1	0	1

Each shift register is of four bits containing four d-flip flops. The data enters as parallel into the registers and shifts one position right at each clock cycle that is the output of one d-flip flop enters as input into the next d-flip flop.

3.3.1 Advantages and disadvantages of serial adder

Advantages

- The least costly hardware-cost circuit.

- Requires only one full adder

Disadvantages

- Requires n clock cycles to compute the complete sum of two n-bit numbers.
- This adder is slow but can work well if the clock speed is high.

3.4. Ripple Carry Adder

The simplest and shortest adders are ripple carry adders, but they are ineffective because the carrier must bounce from the smallest to the greatest bit. The easiest method is to combine two amounts. By using a simple array of full adders, carrying out one full adder is equivalent to bringing the next full adder. A "carry ripple adder" is a device that waves the transfer function from the least significant full adder to the highest relevant full adder.

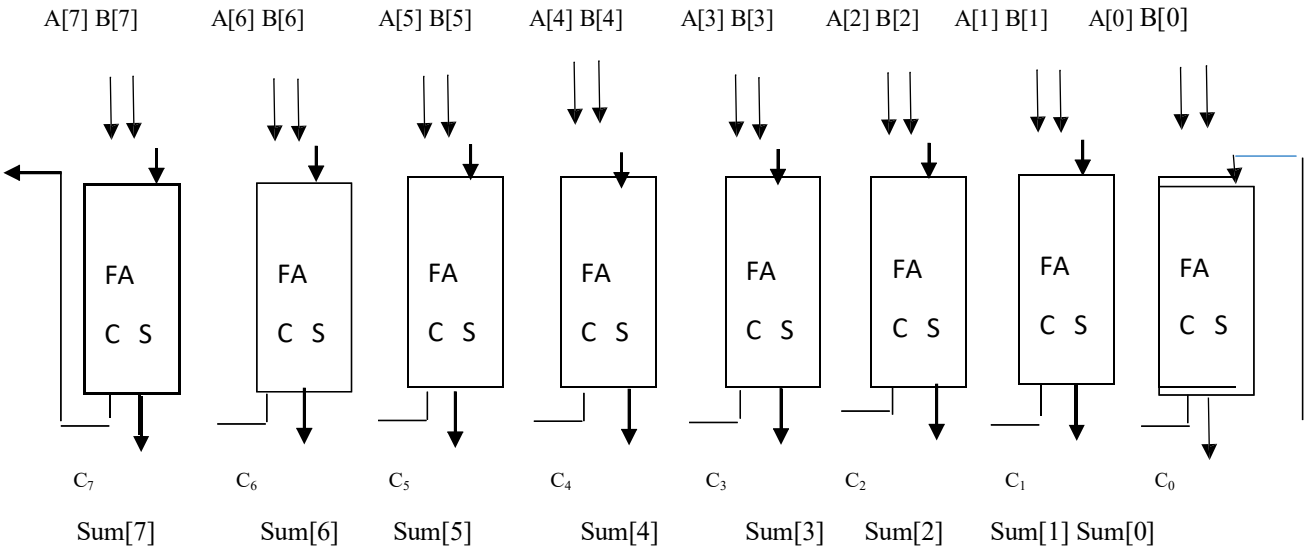


Figure 3.2: Ripple Carry Adder

The speed of this ripple carry adder is:

$$^{r}CPA_{cr} = (n\text{-}1)\ T_{carry} + T_{sum}$$

where the carry and total speeds of a complete adder are Tcarry and Tsum, respectively. Keep in mind that this adder's speed increases linearly with word length. It is thus of order O. (N).

A simple illustration of an 8-bit ripple carry adder is shown in the figure above. The two binary discords A and B are being added and stored in Sum. This design has a fairly straightforward look and structure. The ripple results from connecting the carry in from one adder to the carry out from the next adder.

3.5. Advantages and Disadvantages of Ripple Carry Adder

Advantages

- • Adds all the bits of two n-bit values in one clock cycle.
- This adder is fast.

Disadvantages

- • N-full adders are necessary for an n-bit ripple carry adder.
- • The ripple carry adder's hardware requirements grow linearly as n rises.
- • The input carry value determines the value of each bit in the total output.

- • Ca must wait for Ca in a 4-bit adder. Ca must hold off till Ca, and so forth.
- • Carry signal ripples over all n stages of the adder in the worst case scenario.
- • The latency in carry propagation is significant.

3.6. Pipelined Ripple Carry Adder

The architecture of a pipelined ripple-carrier adder for carrying out a series of N adds is depicted in Figure 3.3 below. When combining two integer vectors with N components, add sequences of this kind. If T is the pipeline's clock period, then computing the single sum x, #y requires 4T of processing time, or 4T of pipeline delay. This number is the sum of the delayed duc to the buffer registers and the time needed to do one addition on a nonpipelined processor. A new total appears after the data has been added to each of the pipeline's four steps. As a result, N consecutive additions are possible.

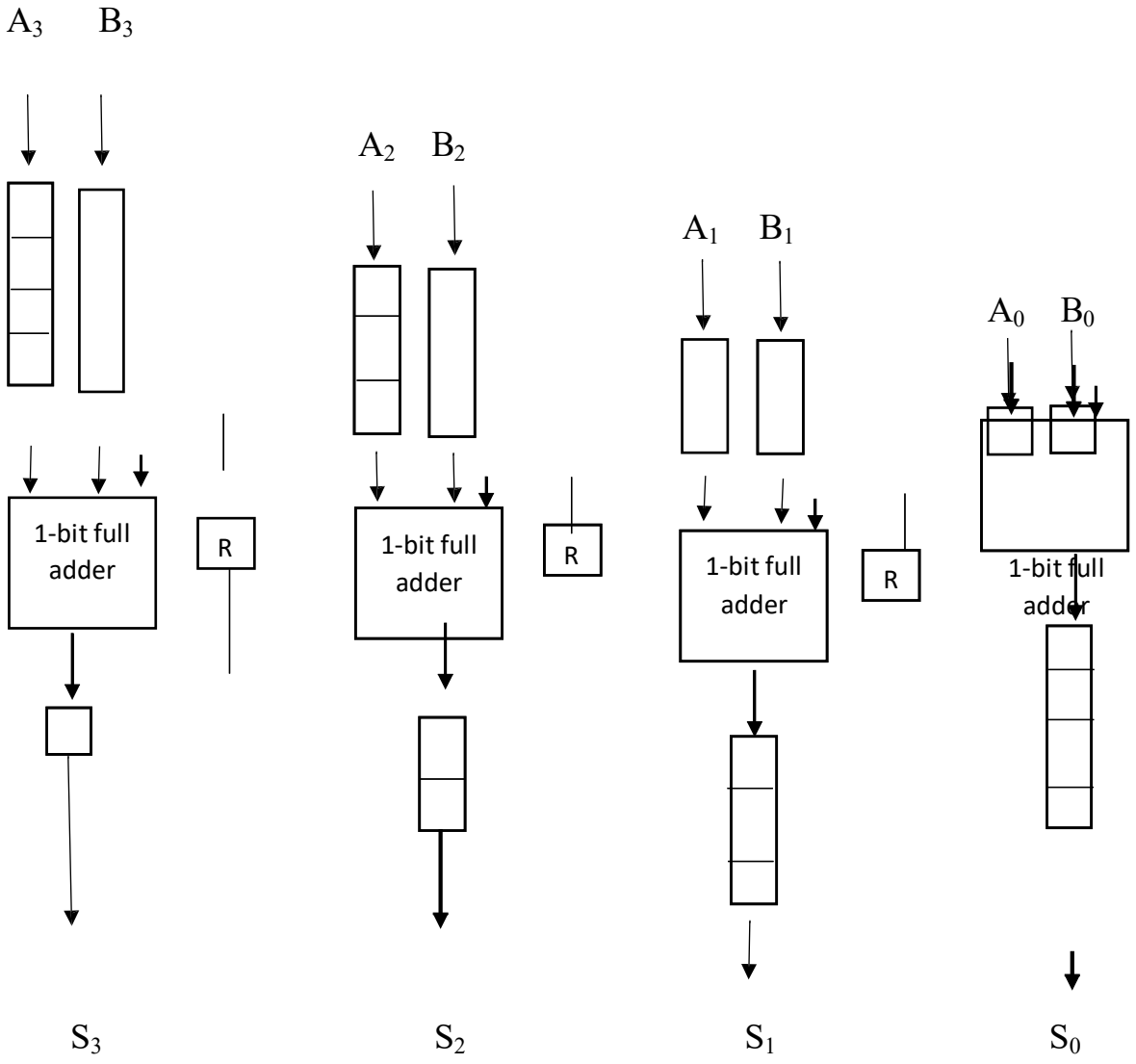


Figure 3.3: Design of Pipelined Ripple Carry Adder

3.7. Carry Look Ahead Adder

It follows that each of the bits of a bone expansion and addends are accessible for calculation at once when two binary integers are multiplied in parallel. The signal must pass through the gates, just as in any multiplexing, for the output terminals to show the correct output total. By dividing the number of gate levels in the circuit by the usual gate's propagation delay, the total propagation time can be calculated. With a parallel adder, the carry propagation over all adders takes the longest. As each bit of the overall output depends on the value that the input carry provides, the value of S at each given stage in the

instrumentation amplifier did not reach its steady-state state final value until that step's intake carry was supplied.

The speed at which two numbers are added concurrently is controlled by the carry propagation time. Even though there will always be some at the output terminals of combinational circuits, including parallel adders, the output won't be precise until the signals have had enough time to travel through the gates connecting the inputs to the outputs.The time spent adding is very important since all other mathematical operations are handled via a series of adds. Using 1aster gates with shorter delays is a clear way to lower the carry propagating delay time. Yet, the capabilities of physical circuits have a limit. Another option is to make the equipment more sophisticated in order to decrease carry delay time. With a parallel adder, the carry propagation time may be decreased using a variety of methods. The most common method, which is explained here, uses the look-ahead-carry concept.

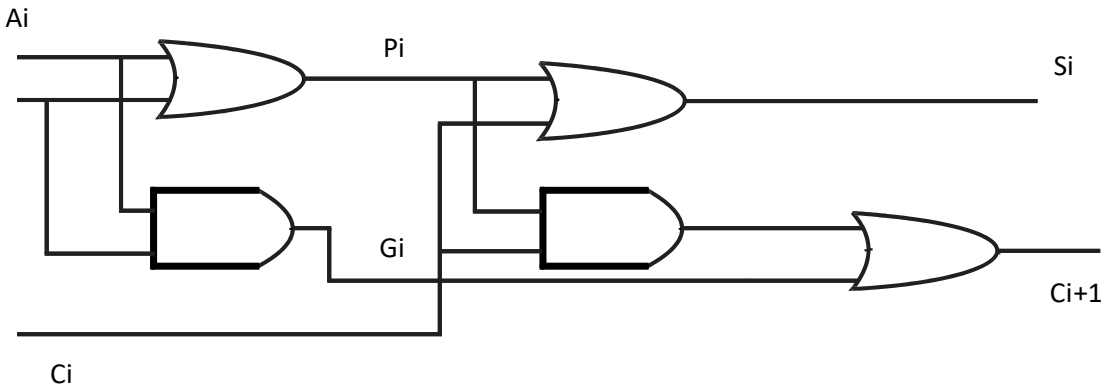


Figure 3.4: Full adder circuit

Think about the whole adder circuit seen in fig. 4.3. If we introduce two additional binary variables:

Pi = Ai + Bi

Ai + Bi + Gi

Si = Pi + Ci is a formula for expressing the output sum and carry.

Gi + Pi = Ci+1 Ci

When both Ai and Bi are one, independent of the input carry, Gi is referred to be a carry generator and it generates an output carry. Due to its association with the carry's propagation from Ci to Ci+1, Pi is also known as the carry propagate. The carry output for each stage is now expressed as a Boolean function, and for each Ci, the value from the prior equations is substituted:

C4 = G3 +P3 C3 =G3 +P3G2 +P3P2G1 +P3P2P1C1 C2 = G1 + P1C1 C3 =G2 + P1C1 G1 + P1C1 = G2 + P2G1 + P2 P1 C1

The term "produced" describes how stage I results in a carry of 1 (Ci = 1) regardless of the amount of Ci-1 if both Ai and Bi are 1, or if AiBi =1. If I propagates Ci-1, leading Ci to equal 1 in response to Ci-1 equaling 1, then Ci will equal 1 if either Ai or Bi equals 1.

Each function may be implemented using a single level of AND gates followed by an OR gate or a two level NAND since the Boolean function for each output carry is represented in sums of products.

Carry look-ahead adder is used over the much more simple ripple adder. The key is speed. Carry look ahead avoids the large propagation delays that are generated in the ripple scheme.

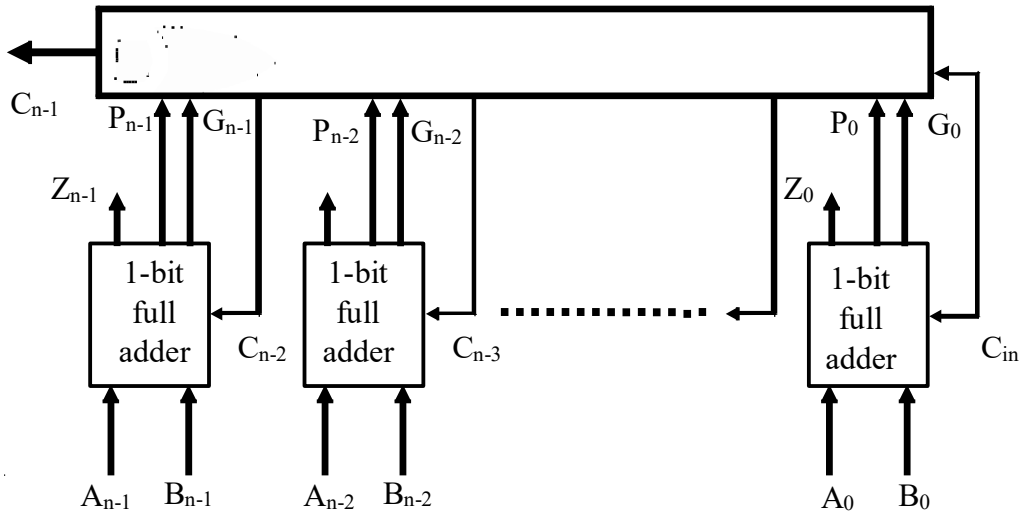


Figure 3.5: Overall structure of carry look ahead adder

Fig. 3.5 shows the general form of a carry look ahead adder circuit. The sum equation for the stage i $Z_i = A_i + B_i + C_{i-1}$ is equivalent to $Z_i = P_i + G_i + C_{i-1}$

Following plumbing diagram is a good way of thinking about the differences between ripple and carry look-ahead. The top pipe is the ripple adder's path. Each lever along the bottom represents each adder in the sequence. One step in the sequence requires the previous steps to finish. For a long pipe (large adder) this delay would be immense. The bottom pipe flows a more independently (each if the feed-ins represents the logical determination of carrying in the carry look-ahead scheme).

Figure 3.6: Plumbing diagram showing the difference between ripple carry and carry look-ahead address.

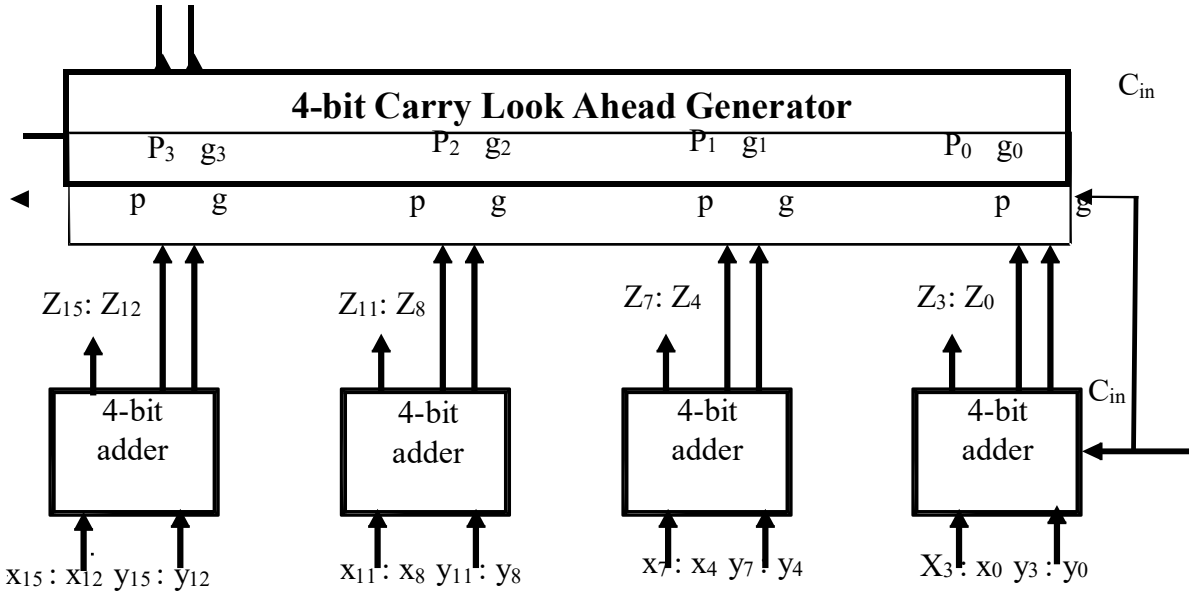


Figure 3.7: 4-bit adders connected by carry look-ahead form a 16-bit adder.

This design is an example of a 16-bit carry look adder, which is fast, expensive, and unusable owing to the complexity of its carry-generation logic. Each adder stage outputs a propagate-generate sign rather than C out after a carry-look-ahead generator converts the four sets of propagate-generate signals into the carry required by the four stages.

3.7.1 Advantages of Carry Look Ahead over Ripple Carry Adder

- Fast addition speed over ripple carry adder.
- Very few carry ripples.
- But incorporates high area.
- Cost and complexity is too much as compared to ripple carry adder.
- Exponential rise in the area of look ahead adder with increasing number of bits.

3.8. Pipelined Floating Point Adder

Before the matching mantissa can be added or removed in floating point addition, the exponents of the two input operands must be identical. Exponent equalization may be achieved by right-shifting the mantissa linked to the lower exponent to create a new mantissa, which can then be merged directly. There are three basic phases in floating point addition:

- Do a fixed-point subtraction of YE- XE.
  - To derive the formula XM 2 X E - Y E, move XM YE-XE positions to the right.
- For XM 2 X E - Y E YM,
- do a fixed-point subtraction or addition.

Two normal flying values, x and y, may be added using the following four steps: check the exponent, align the mantissa (equalize the algebraic expressions), add the mantissa, and standardize the results.

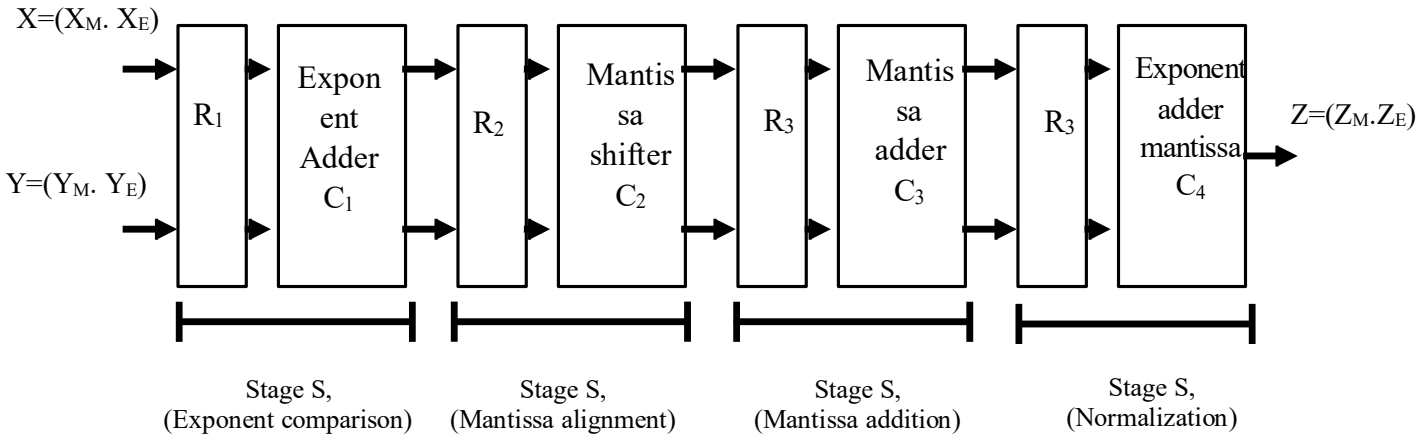


Figure 3.8: Four stage floating-point adder pipeline

The four-stage pipeline processors that is shown in the accompanying fig. 4.5 is necessary for these activities.

Assume that x has the standardised precision floating-point form (XM, XE), where XE is the exponents with respect to a certain base B = 2K and XM is the mantissa.

At stage Si of the pipeline, X = (XM, XE) is added to Y = (YM, YE), which compares XE and YE, as the first step. The exponents are subtracted in this comparison, which necessitates the use of a fixed-point adder. At the second stage of the pipeline, S2, Si detects the smaller exponent, let's call it XE, whose mantissa XM may be altered by shifting to produce a new mantissa XM and YM that are now perfectly aligned, and is then added.

A fourth and final step is required to normalize the result since this fixed-point addition may result in an unnormalized result.

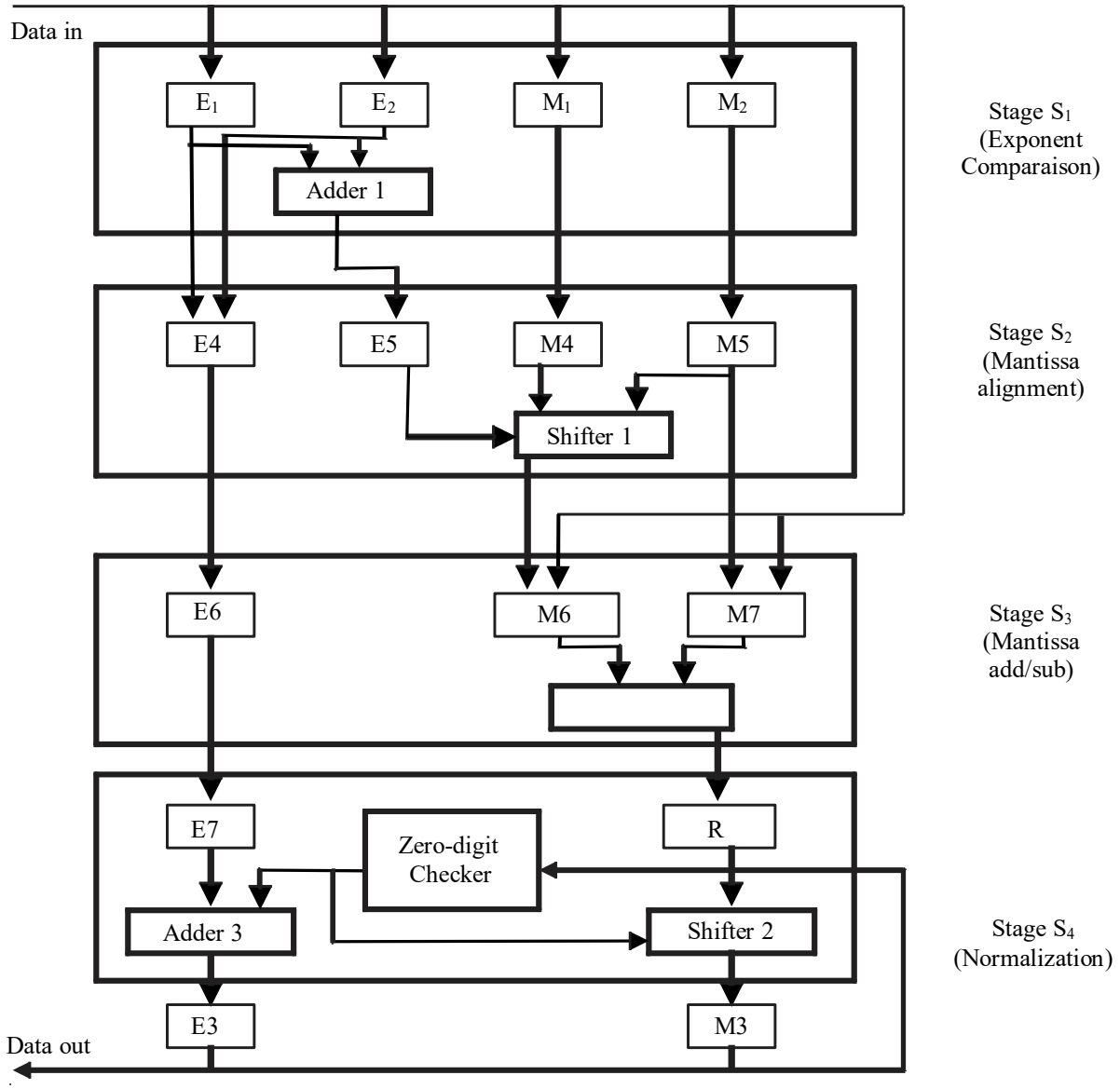


Figure 3.9: Pipelined version of floating point adder

Finding an appropriate multistage sequential method to calculate the specified function is the first step in designing a pipelined circuit for that function. The pipeline stages that implement this algorithm's phases should be balanced in that they should all take about the same amount of time to complete. Rapid buffering registers are used to separate the stages so that pertinent data (full or incomplete results) may be transferred from one without affecting the other. The buffer are designed to operate at the highest clock rate feasible to provide reliable data transmission between stages.

A fly adder pipe with a registered architecture based on unnecessary delay design is shown in Fig. 3.9. The usage of buffer values to define and differentiate the four phases is the fundamental departure from the nonpipelined situation. The implementation of repaired and flying addition has undergone another modification. The input operands are only sent to step S, which then conducts a fixed-point addition; the other three steps are bypassed. The circuit in Figure 3.9 is an example of a multifunctional pipeline that can be set up as either a multiple hanging adder or a one-stage fixed-point adder. Moreover, this circuit is capable of floating-point subtraction.

The following describes how this floating-point adder unit works. Whereas registers E1 and E2 have the exponent of the input operands, registers M1 and M2 store the corresponding mantissas. The result is used to choose the mantissa that shifter I will right shift to and to calculate the duration of the shift. Adder 1 then subtracts E2 from E1 to get to E2. For instance, M2 is right shifted by k digit places, or 4k bit positions, if E1 is bigger than E2 and E1 - E2 = k. The other is among and the shifted mantissa are then added to or subtracted from one another using adder 2, a 56-bit concurrent adder with a few carry look-ahead levels. The resultant sum or difference is then checked by the zero-digit checker, a specialized combinational circuitwhich is placed in a temporary register R. The amount of leading zero digits—or leading ones, in the case of negative numbers—that make up the number in R is shown on the circuit's output z. The last normalization step is then managed by the value of z. Using shifter 2, data from R is left-shifted by z digits before being placed into register M3. The necessary adjustment to the exponent is made by deleting Z using adder

Pipelined Adder with Feed

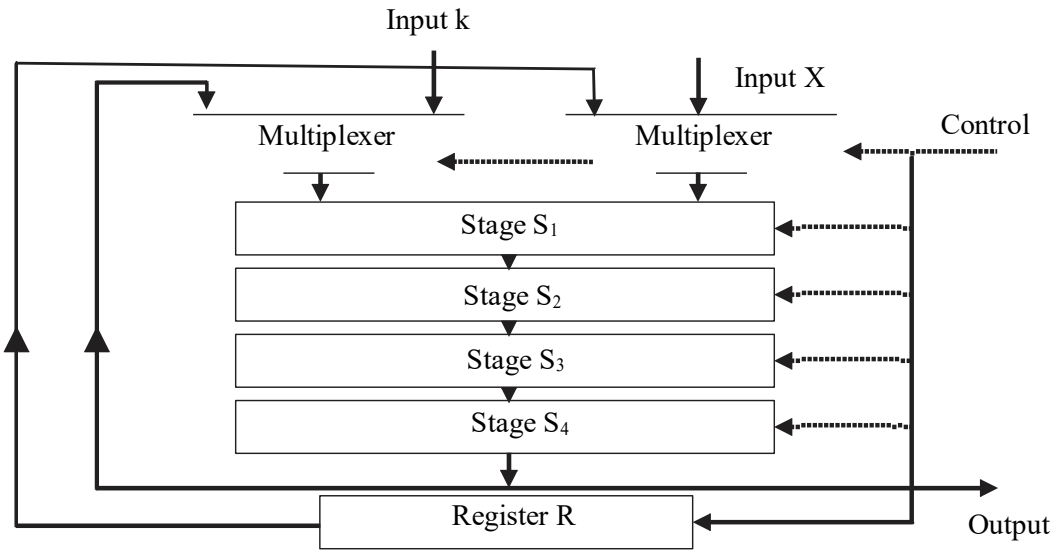


Figure 3.10: Pipelined Adder with Feed Back Path

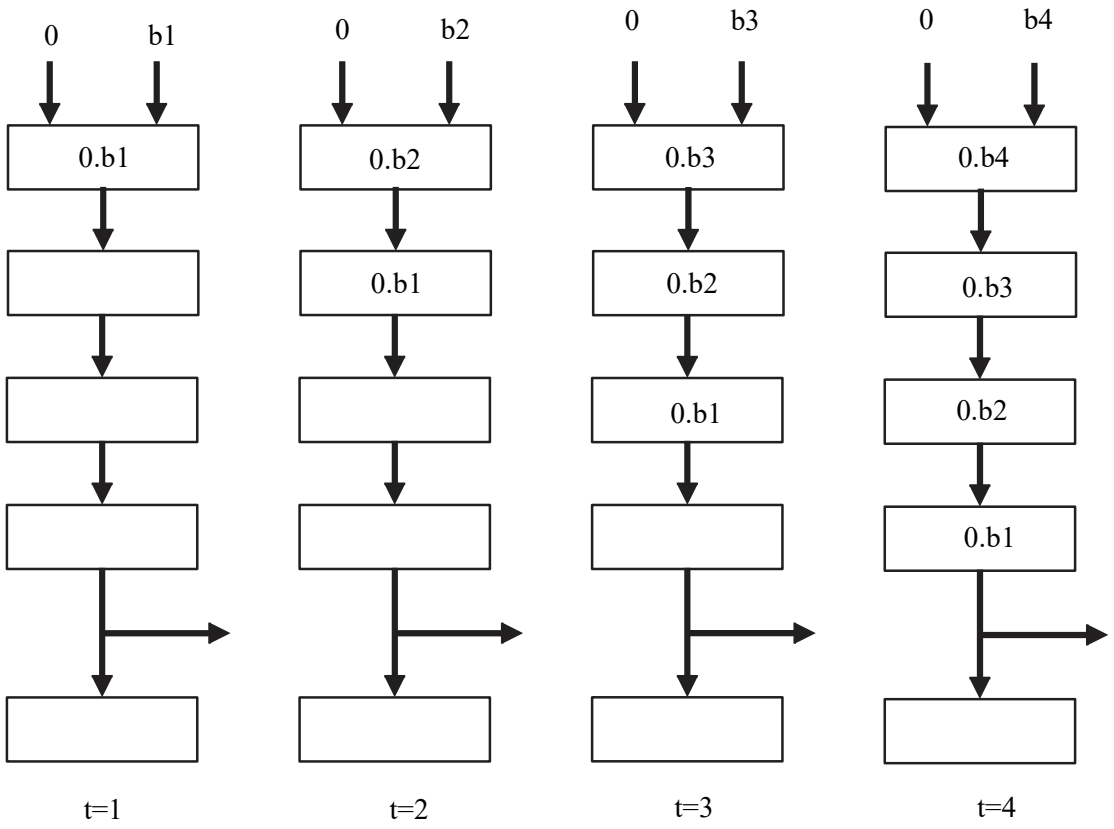


Figure 3.11: Summation of an eight-element vector

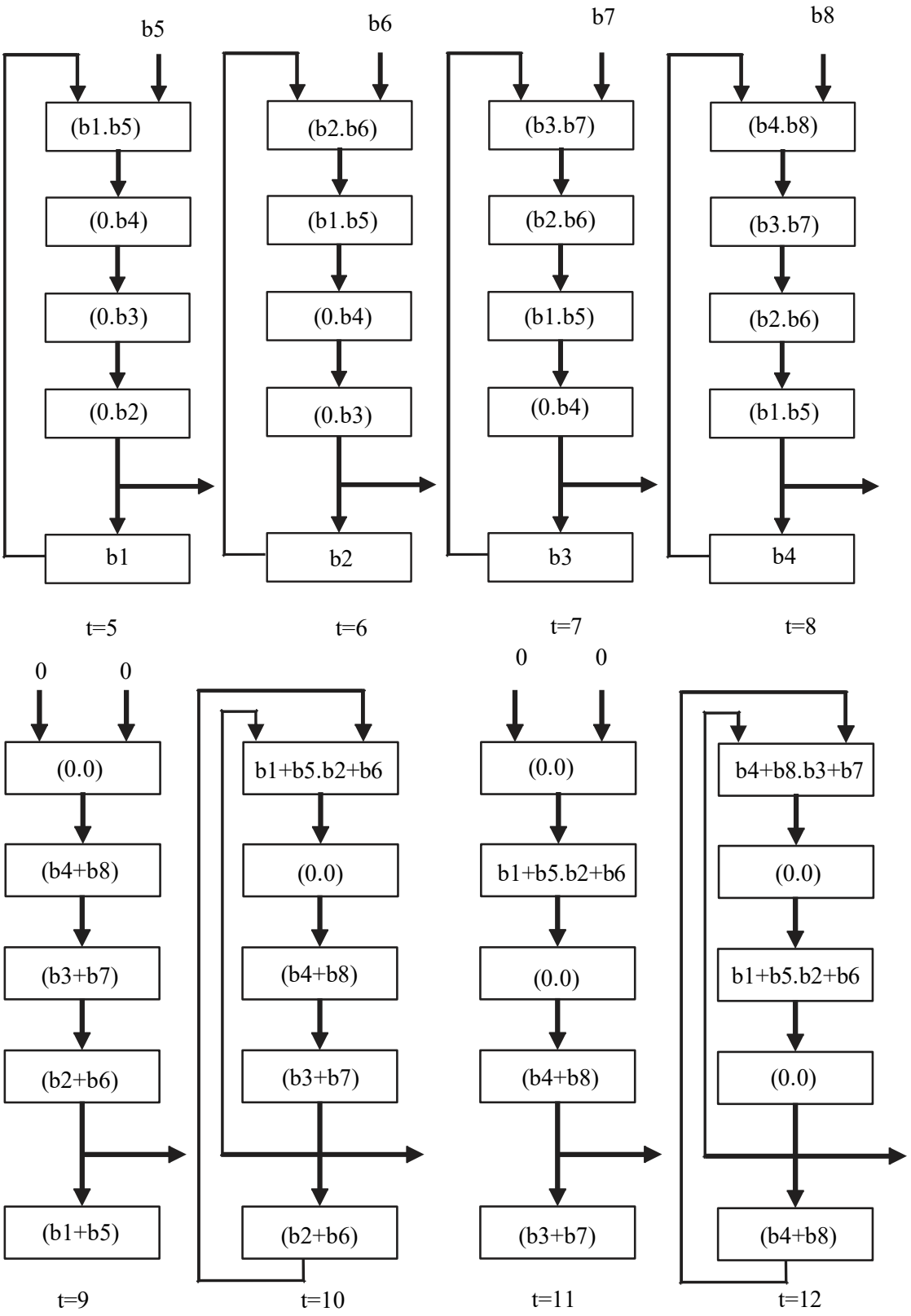


Figure 3.12: (continued)

In certain cases, introducing reward systems from the stage result to the pipeline's primary inputs might boost the effectiveness of the processor.Feedback allows the pipeline to utilize the calculations made by certain stages as input for further computations.

The N-number summation issue is resolved in the following manner by the pipeline of Fig. 3.11. With input x, the external operands bl, b2, b3, b4,... b8 are streamed continuously into the pipeline. If the operands are kept in adjacent register/memory locations, it is simple to implement the required series of fetch operations for this procedure. The pipeline input K is given the all-0 word designating the floating-point number zero while the first four values bl, b2, b3, and b4 are being entered, as shown in fig. 3.11 for times t=1:4. The initial sum 0 + bl = b1 emerges from S4 at time t=5, or four clock cycles later, and is fed back into the Pipeline's main inputs. At this stage, the outcome S4 = b1 replaces the constant input K = 0. Now, the pipeline starts to calculate bl + b5. The calculation of b2 + b6 starts at time t=6, b3 + b7 computation starts at time t=7, and so on. At time t = 8, when bl + b5 exits the pipeline, it is sent back to Si to be added to the most recent incoming number. The total b2 + b6 emerges from the pipeline in the following time period.

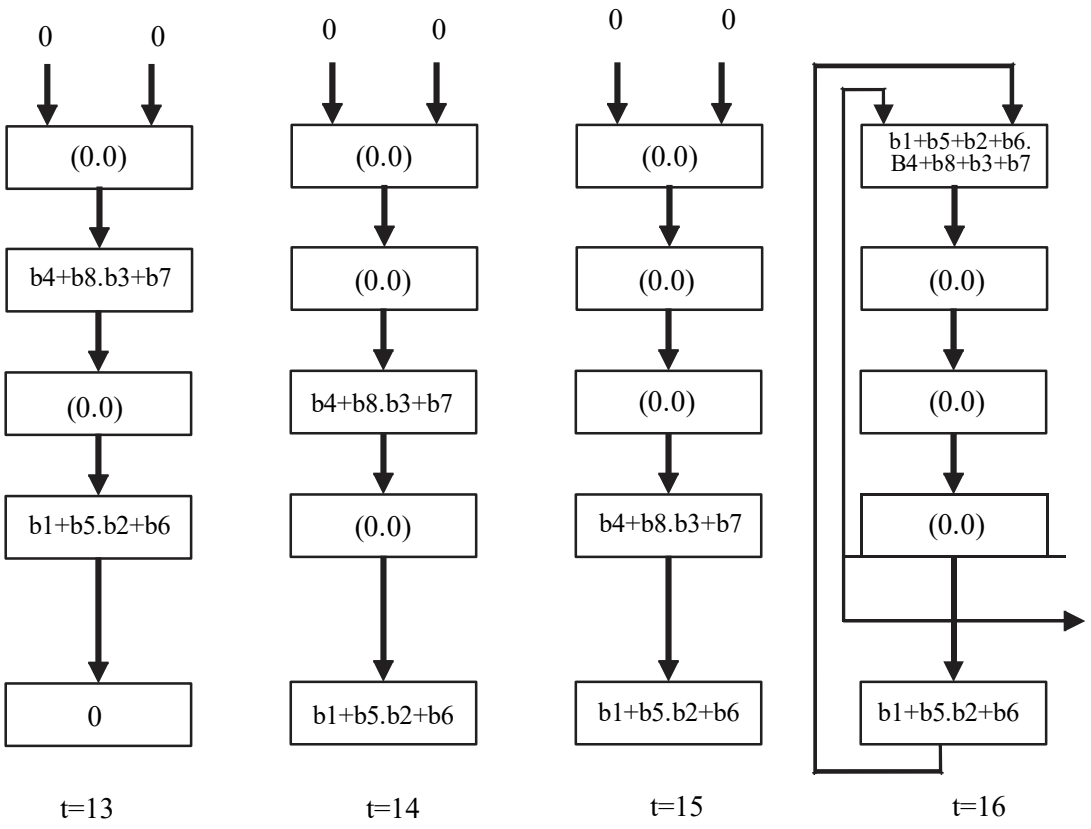


Figure 3.13: Summation of an eight-element vector (continued)

The output structure is once again changed once the last input component, b8, has entered the pipeline so that the four incomplete sums may be summed to get the intended effect, bl+b2+b3+.....b8. The first of the four partial sums at the output of step S4 is placed in register R at time 9, and the external inputs to the pipeline are disabled by setting them to zero. At time 10, the newly acquired result from Sa is returned to the pipeline inputs together with the prior result—b2 + b6—obtained from the register R. At this point, the total of the input operands, bl + b5 + b2 + b6, is computed. The second half total—b3 + b7 + b4 + b8—is computed after a further one-time delay. At time t = 14, bl + b5 + b2 + b6 emerges from Sa. It is kept in r until time t = 16, when b4 + b8 + b3 + b7 emerges from S. The outputs from Sa and R are now sent back to S1. In the instance of N=8, the final result is achieved at t=20, four time periods later.

It is obvious that the design of fig. 3.11, where T is the clock period of the pipeline or the delay per step, can calculate the sum of N>4 floating point integers in time (N + 11)T for the general case of N operands. We get a speedup of around 4N/(N+11) here, which becomes closer to 4 as N rises, compared to an equivalent nonpipelined adder, which needs time 4NT to calculate the total.

3.9 Carry Save Adder

Carry save addition is a method often used by multipliers that is especially well suited for pipelining. N discontinuous complete adders make up an n-bit carry-save adder. Three n-bit values to be added are its input, and its output is made up of the n sum bits, which make up the word S, and the n carry bits, which make up the word C. There is no carry propagation inside the individual adders, unlike the adders. The outputs S and C may be used to add to a third n-bit number W by feeding them into another n-bit carry save adder. To match to standard carry propagation, the carry connections are moved to the left. In general, a tree-like network of carry-save adders may add m integers to generate a result of the form (S, C). A traditional adder with carry propagation must add S and C to get the final amount.

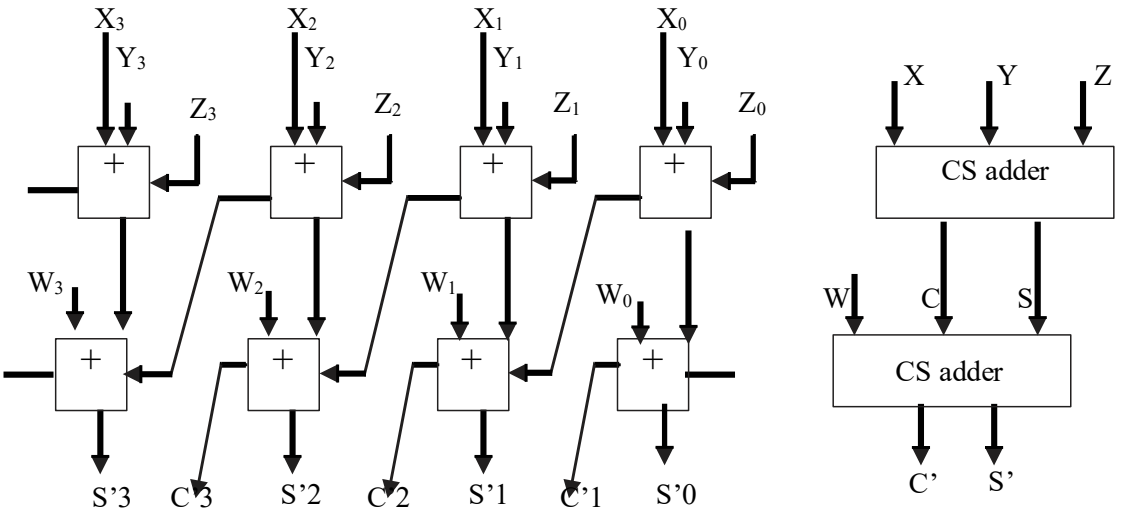


Figure 3.14: A two-stage Cary Save Adder

Example:

W	1	0	1	1	
X	0	1	0	1	
Y	1	0	1	0	**Stage I**
Z	0	0	0	1	
SUm	0	0	1	0	
Carry 1	1	1	1		**Stage II**
Z	0	0	0	1	
Sum 1	1	0	0	1	
Carry 1	1	1	0		

CHAPTER 4

ARCHITECTURE OF MULTIPLIERS

This section deals with the different types of multipliers:

- Carry-Save (Wallace Tree)
- Pipelined Carry Save (Wallace Tree)
- Booth's
- Dadda

The Wallace tree, named after its inventor Wallace [Wallace 1964], is a multiple stage carry-save adder circuit that can do multiplication. The adder tree's inputs consist of n terms with the formula Mi = xi Y 2k. Mi is represented as a Y-value in this instance and is equal to Y times the ith multiplier bit, weighted by the appropriate power of 2, and. Imagine the scenario when a complete double-length product is needed and Mi is 2n bits long. The sum of M, where is a value between 0 and n-1, is the required product. The carry-save adder tree, which generates a 2n-bit and 2n-bit carry word, is used to get this total. The final carry assimilation is carried out by a fast adder using regular internal carry propagation.

For moderate values of n, the solely combinational multiplier of fig. 4.1 is practical, depending on the level of circuit integration used. For huge n, a large number of carry-save adders may be required. Carry-save techniques may still be employed, even if the multiplier is partitioned into k m-bit portions.The carry saving adder circuit generates and adds just m terms, Mi. The procedure is carried out k times, and the totals that result are then added together. As a result, k iterations are required to produce the product.

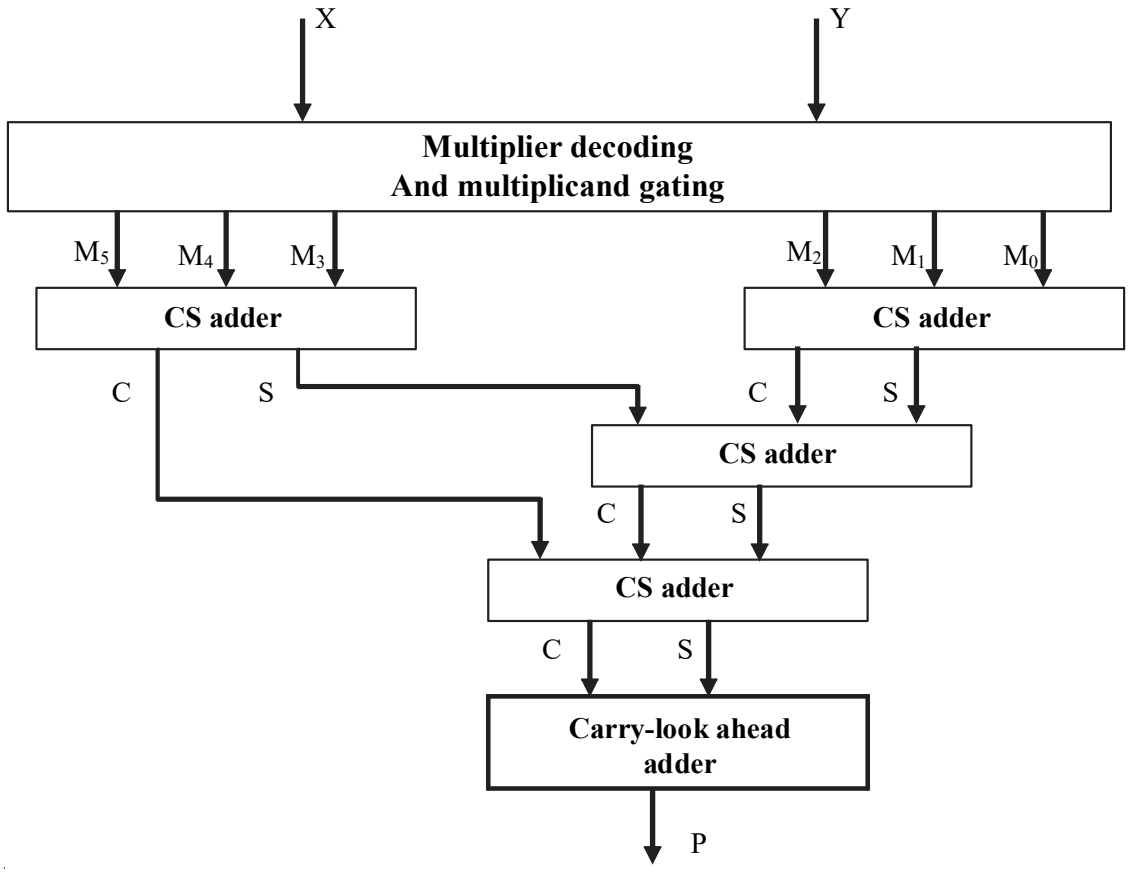


Figure 4.1: A carry-save (Wallace tree) multiplier

4.2 Pipelined Carry Save Multiplier

For pipelined implementation carry-save multiplication is well-suited figure 4.2 shows the pipelined multiplier. The first stage decodes the multiplier and feeds the carry-save adders with properly shifted copies of the multiplicand.

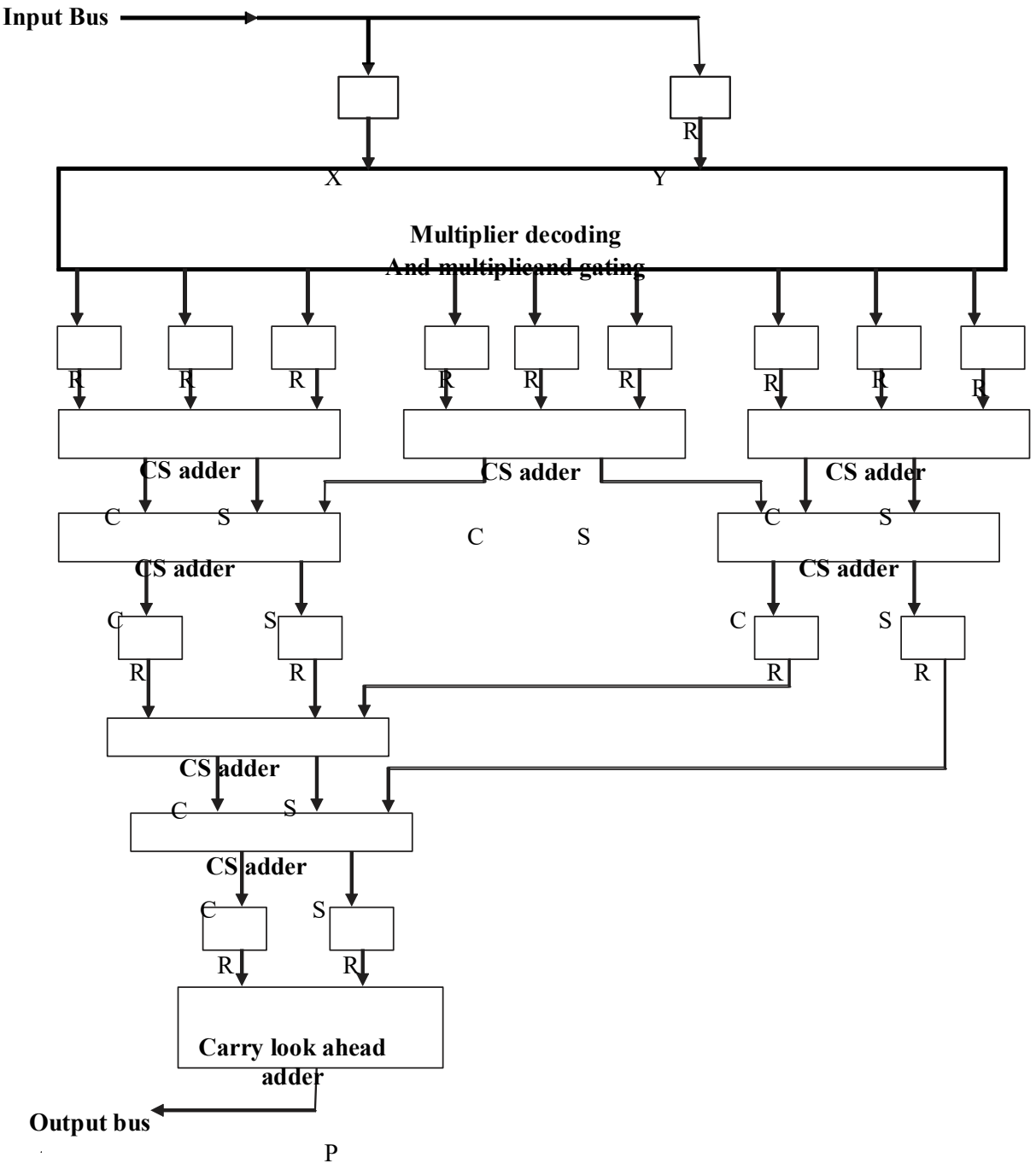


Figure 4.2: A pipelined carry save multiplier

The first step produces a collection of numbers (partial products), which are then added together by the carry save adder tree. By adding buffer registers, shown in the diagram as R, the carry saves logic has been split into two steps. A carry look-ahead adder is included in the fourth and final step to incorporate the carries. It is simple to adapt this sort of multiplier to work with floating point numbers. A fixed-point multiplier pipeline

handles the input mantissa's processing. A separate fixed-point adder is used to combine the exponents, and a leveling circuit is also included Booth's Multiplier.

In the 1950s, Andrew D. Booth devised the two's complement multiplication system that is now in use. While doing a multiplication, the multiplier X is scanned from right to left to identify which action to carry out: adding the multiplicand Y, subtracting Y, or adding zero, or performing no operation. In Booth's method, each step involves looking at two neighboring bits, Xi Xi-1. If Xi Xi-1-= 01, Y is added to the current partial product Pi, and if Xi Xi-1-= 10 or 11, Y is removed from Pi. If Xi Xi-1-= 00 or 11, no addition nor subtraction is done; instead, Pi is simply shifted to the right thereafter. Hence, when Booth's algorithm sees runs of 1s and runs of 0s in X, it basically ignores them. This shifting lowers the approximate rating of add-subtract steps and enables the construction of faster multipliers, but at the cost of more intricate timing and control circuitry..

Step	Action	Accumulartor A	Register Q
0	Initialize registers	000000000	10110011 = Multiplier
	Set Q |-1| to 0	000000000	101100110
1		11010101	= multiplicand Y = M
	Subtract M from A	00101011	10110011<u>0</u>
2	Right shift A.Q	00010101	110110011
	Skip add/subtract	00010101	11011001<u>1</u>
3	Right-shift A.Q	00001010	111011001
		11010101	
	Add M to A	11011111	11101100<u>1</u>
4	Right-shift A.Q	11101111	111101100
	Skip add/subtract	11101111	111110110
5	Right-shhift A.Q	11110111	11111011<u>0</u>
		11010101	011111011
	Subtract M from A	00100010	01111101<u>1</u>
	Right shift A.Q	00010001	101111101
6	Skip add/subtract	00010001	
	Right shift A.Q	00001000	10111110<u>1</u>
7		11010101	110111110
	Add M to A	11011101	
	Right-shift A.Q	11101110	11011111<u>0</u>
8		11010101	110111100 = Product P
	Subtract M from A	00011001	
	Set Q[0] to 0	00011001	

Figure 4.3: 1llustration of the Booth Multiplication algorithm

A multifunction cell with the ability to do addition, subtraction, and no operation skip is necessary for a booth multiplier. Figure 4.4a depicts a cell of this kind. A set of control lines H and D are used to choose from among its different functionalities. The following logical equations clearly show that the necessary function of B is defined by them.

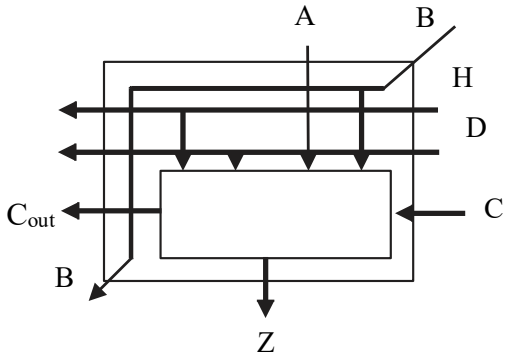
A=B H CH in Z

(A A D) (B + C) (BC) Cout

These equations decrease to complete adder equations for HD = 10 and to full subtractor equations for HD = 11, respectively.

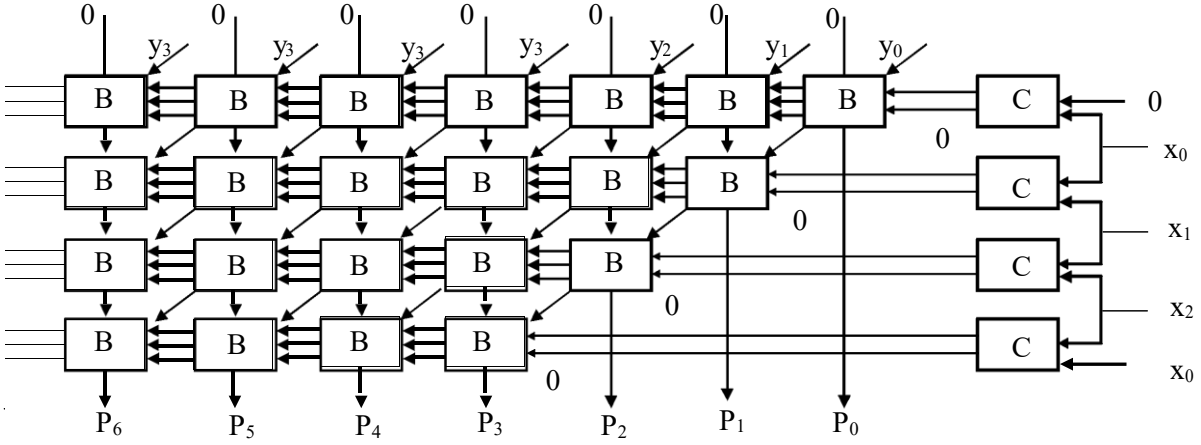
Z = A B C

A'B + A'C + BC = Cout



H	D	Function
0	X	Z = A (no operation)
1	0	Cout Z = A *Plus* B *Plus* C (add)
1	1	Cout Z = A – B – C (subtract)

Fig 4.4 Main cell B (a)



(b)

Figure 4.4: (continued) Combinatorial array that implements the Booth method

(b) An array multiplier for a 4*4-bit value, where C and Cout, respectively, serve as the borrow-in and borrow-out functions. When H = 0, Z turns into A, and the carry lines have no effect on the result.

Fig. 4.4 illustrates the connection of n2 + n (n-1)/ 2 copies of the B cell to create an n-bit multiplier (a). During addition and subtraction, the additional cells on the upper left are used to sign-extend the multiplicand Y. Every row of B cells receives the sign-extended Y straight from the diagonal lines designated B. Leading 0s are used to prolong the sign of Y when it is positive; whereas, leading 1s are required to extend the sign of Y when it is negative.

Bits Xi Xi-1 of the operand X determine which operation will be carried out by each row I of the B cells. The second cell type designated in fig. 4.4(b) produces the control input signal H and D needed by the B cells, allowing each potential x; Xi-I pair to control row operations. Cell C performs an x-to-x comparison and produces the value of HD needed by Figure 4.4(a): these values are _____ H = Xi Xi-1.

4.3 Dadda Multiplier

A 12*12 Dadda tree reduction strategy is shown in Fig. 4.5. Similar to a Wallace tree, it takes 5 layers of complete adders to decrease 12 partial products to 2. It has the same decrease rate as a Wallace tree. The Dadda tree's sole advantage is that it has the ideal amount of computational components, or full adders and half adders. Smaller size and less power dissipation are the primary results of this. To decrease the 12 partial products to two using the Wallace tree multiplier, 102 full adders and 34 half adders are needed. Just 100 full adders and 14 half adders are required to complete the same reduction in the Dadda tree. Even if there are not many complete adders, there is a significant reduction in the number of half adders.

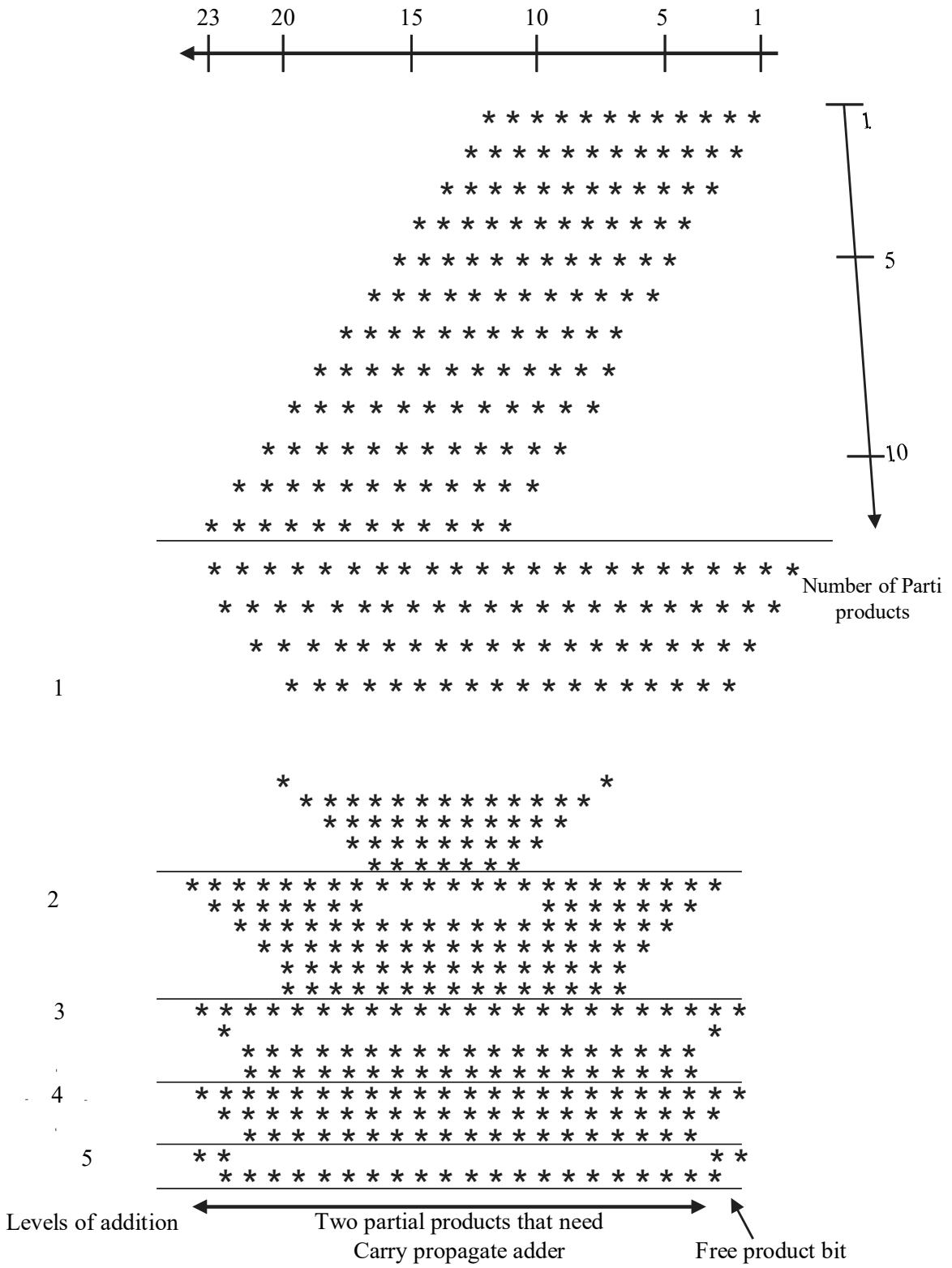


Figure 4.5: 12 * 12 Dadda Tree Multiplier

Use the following numerical series to comprehend how the savings are achieved: This list simply explains the 3 to 2 reduction method backward, with each number denoting the

amount of partial products still present at each level of addition: 2, 3, 4, 6, 9, 13, 19, 28, 42, 63, etc. Two partial products may be decreased to a maximum of three partial products, three to a maximum of four, four to a maximum of six, and so on. This suggests that it is not necessary to reduce an array to parlay products since 5 partial products are equal to 6 in terms of the remaining number of reduction levels. Hence, one would decrease the 12 partial products in the example above to 9 partial products rather than 8 partial products as in the Wallace method. In addition, no reduction is made if a row has 9 or fewer incomplete items. The Walla, of which just two remain. The Dadda technique attempts to eliminate all partial products except computational components, resulting in an optimum reduced number of Wallace implementations.

As previously noted, the Wallace tree and the Dadda tree both have the same rate of partial product decrease. It comes from:

$$Bj + 3\,[(bj)/2] + Bj \bmod 2$$

Where bj is the quantity of incomplete products at the jth adder level still remaining. The number of adders levels b for big n is roughly given by: $b = \log_{3/2}(n/2)$

The quantity of free product pieces is different from a Wallace tree, however. One can observe from fig. 4.5 that there is just one free product bit obtained: f=1

CHAPTER 5

ARCHITECTURE OF SUBTRACTERS DESIGN OF SUBTRACTERS USING 2'S COMPLEMENT

5.1 Introduction to 2's complement

In digital computers, complements are employed to make logical operations and the subtraction operation simpler. Every base-r system has one of two complement kinds. When the base value is changed, the two types are referred to as 2's complement and 1's complement for binary numbers. The l's complement of a binary number and the 2's complement of a binary number are important because they enable the representation of negative integers.Computers often handle negative integers using the 2's complement arithmetic approach.

5.1.1 Finding the 2's complement of a Binary Number

There are two ways to use the 2's complement: (1) Adding I to the 1's complement's least significant digit.

Example 5.1

Binary number 01001101 is the solution to 10110010's 2's complement. l's complement 1 plus 1 equals... 010011 (2) By converting all 1s into 0s and all 0s into ls after leaving all leading 0s in the least important position and the initial 1 alone.

Example5.2

Solution 10111000 for 2's complement of 10111000 the binary value 01001000 complement of 2

They remain the same.

5.1.2 Subtraction with 2's complement

These are several methods for subtracting two positive integers, both with base 2:

1. To the subtrahend's complement of two, add the minuend.

2 Examine the first-step outcome for an end carry. If an end carry happens, throw it away; if not, take the 2's complement of the number you got in step 1 and put a negative sign in front of it.

Example

Perform M-N using 2's complement with the given binary number

M =1010100

N = 1000100

2's complement of N = 0111100

1010100

1000100

End carry 1/ 0010000

Result: 10000

Signed Numbers

Both sign and magnitude information may be found in a signed binary integer. The magnitude represents the value of the number, and the sign shows whether it is positive or negative. Signed integer numbers may be represented in binary form in three different ways: sign-magnitude complement, ,'s and 2's complement. In a signed binary number, the sign bit is the leftmost bit. An O denotes something good, and a l something bad.

5.1.3 Signed Numbers in the system of two complements

The following symbols are used to represent positive integers in the 2's complement system: An 8-bit signed binary integer is used to represent the decimal value +25.

Number bit 00011001 for the sign

Quantity bits

In the 2's complement system, a negative number is the 2's complement of a positive integer. A decimal number having a value of -25 may be represented as the 2's complement of +25 (00011001), or 11100111.

The twos-complement algorithm makes subtracting very straightforward due to how simple it is to implement negation (changing X to -X).

If X = Xn-1 Xn-2 Xn-3, then Xo is an integer with two's complement; negation is accomplished by

- X = Xn-1, Xn-2, Xn-3, etc., where + stands for addition modulo 2n. How to get the ones-complement part X Xn-1 X Xn-2 Xn-3 effectively? The word-based EXCLUSIVE OR function X s with a control variable s is used in X0 of -X. X s=X when s = 0, but X s=X when s = 1. Let's say that an n-bit adder now has Y and X applied to its inputs.

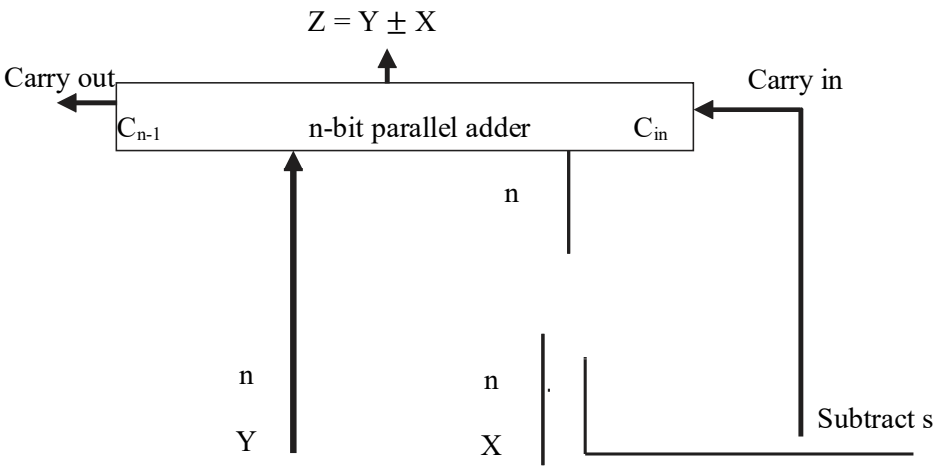


Figure 5.1: An n-bit twos-complement adder-subtracter

Applying s to the carry input line of the adder will make it possible to achieve the addition of I necessary to convert X to -X. The control line chooses the addition operation Y + X in the circuit shown in Fig. 5.5 when s = 0 and the subtraction operation Y - X= Y + X + 1 when s = 1. Hence, to execute twos-complement subtraction, the adder's input must be connected to n two-input EXCLUSIVE-OR gates; these gates are represented by a single n-bit word gate in fig.5.1.

Numbers that are unsigned or have a sign-magnitude do not lend themselves to subtraction as easily. The complete (1-bit) subtracter function zi = yi-xi-bi-1 may be used to build a subtracter for such integers on occasion. The following logical equations characterize this operation: zi = xi yi bi-1

bi=xi yi plus xi bi-1 plus yi bi-1

Thus, bi-1 and bi serve as the borrow-in and borrow-out bits, respectively, with zi serving as the difference bit. The construction of n-bit serial or parallel binary subtracters is substantially the same as that of comparable adders, with borrow signals used in lieu of carry signals.

CHAPTER 6

ARCHITECTURE OF DIVIDERS

6.1 Introduction to division

Two numbers—a dividend, D, and a divisor, V—are provided in fixed-point division. The goal is to determine the third integer, or division, Q, so that Q* V equals or approaches D. For example, the computation of Q is done in a way that, if integer variable formats are utilized,

$$D=Q*V+R$$

Where R, the remaining amount, must be less than V, i.e., 0=RV. We can write

$$D/V = Q + R/V$$

Thus, R/V is a tiny amount that represents the mistake in substituting Q for D/V; if R = 0, this error is equal to zero. Given that the dividend, quotient, and divisor are equivalent to the product, approximately 5 percent, and multiplier, respectively, the equation D = Q * V shows that there is a tight link between division and multiplication. This link allows for the employment of similar techniques and circuits to execute multiplication and division. When multiplying two numbers, the shifted multiplier is added to the multiplicand to get the result. The dividend is deducted from the shifting divisor to get the quotient in division. Division often starts with a double-length dividend, much to how multiplication concludes with the double-length product. Despite these similarities, division is a more difficult operation than multiplication since it requires knowledge of the number of multiples the current partial payment D1's divisor V is in order to determine a specific quotient bit q. Often, the answer to this issue requires experimentation. Subtract the result from D, multiply V by a test value for q1, and then determine the value of the leftover fraction. It is impossible to identify the next quotient bit, q1+1, unless q is unknown. Uncertainty is

present in this division that is absent with multiplication. The quotient is calculated bit by bit once the dividend has been scanned from left to right. The partial dividend D, also known as partial remainder, is compared to the divisor V in each step. It is possible to tell whether the current quotient is bit 0 or bit 1 by comparing V with the current partial remainder. This comparison is the challenging part of decimalimal division and is more difficult than binary in this way because q must be chosen from a pool of ten possible values rather than two. D will not fit in the usual word size if V is too small compared to D, leading to quotient overflow. When V is zero, it is said that a divide by zero error has happened and that the quotient Q is undefined or infinite. Special circuits are employed to search for and indicate quotient under, flow, and zero prime factors before division begins.

6.2 Division Algorithm

Suppose that the divisor V and dividend D are both unsigned numbers, and that the quotient Q=qn-1 9n-2 qn-3..... must be computed one bit at a time. At step I of each iteration, the current partial residual R1 is matched to 2"V, which indicates the divisor shifted I bits to the right. The quotient bit q is set to 1 (0) and a new partial residual R1+1 is generated using the formula:

R1+1 = R1 - q1 211 V if 2'V is less than (greater than) R1.

The application of the aforementioned equation is as follows: In machine implementation, shifting the partial residual to the left in regard to a fixed divisor is more feasible.

$R_1+1=2R_1-q_1V$

In Figure 6.1, the revised calculation is shown. The final partial residual R is now equal to 23R4 after the overall remainder R has been relocated three bits to the left. The hardest part of the division is locating the quotient digit, q. If radix-r numbers are being represented, one of the r possible values for q must be chosen. With r = 2, comparing V

and 2R in the ith step might result in q. Q = 0 if V > 2R; else, Q = 1. A combinational magnitude comparator circuit may not be practical if V is too long. In such a case, q is often determined by deducting V from 2R and examining the sign of 2R, - V. If 2R is negative, then Q1 equals 0; if positive, then Q1 equals 1.

Divisor V			Quotient Q
101	100110	Dividend D= $2R_0$	
	000	q_3V	0
	100110	R_1	
	1001100	$2R_1$	
	101	q_2V	01
	100100	R_2	
	1001000	$2R_2$	
	101	q_1V	011
	100000	R_3	
	1000000	$2R_3$	
	101	q_0V	0111
	011000	$R_4 = 2^3R$	

Fig no.6.1 the division modified for machine implementation

The division circuitry is depicted in Figure 6.2. two-bit n-bit shift register The remaining components are kept in A.Q. The whole payout, which might include up to 2n bits, is first retained in A.Q. The divisor V is kept in the M register during division. A.Q is shifted to the left on each step. The holes at the correct side of the Q register may be used to store the quotient's bits after they have been created. After division is finished, Q holds the quotient, and A has the shift remainder.

Through the use of a trial subtract of the type 2R, - V, it is possible to identify the quotient bit q.

When 2R,-V is positive, as it is when q=The method of calculating q, and R1 may be integrated, this subtraction also yields the new partial residual R1+1. On the basis of how they combine the computations for q and R, two primary division algorithms may be separated from one another. Perhaps the answer is 4, 0, in which case the trial subtraction gives the value 2R, - V, but the necessary new partial remainder is 2R. By adding V back to the trial subtraction result, one may get the partial residual R-1. This simple method is known as restoring division. Using the formula R1+1=2R1-V, each step is calculated. If the outcome of the subtraction is negative, a restoring addition is made using R1 = R1 + V.

This method necessitates n subtractions and a mean of n/2 additions if the probability of q. 1 is 1/2. Fixed-point subtract with exponents and fixed-point division with the integer part are both necessary for floating-point division.

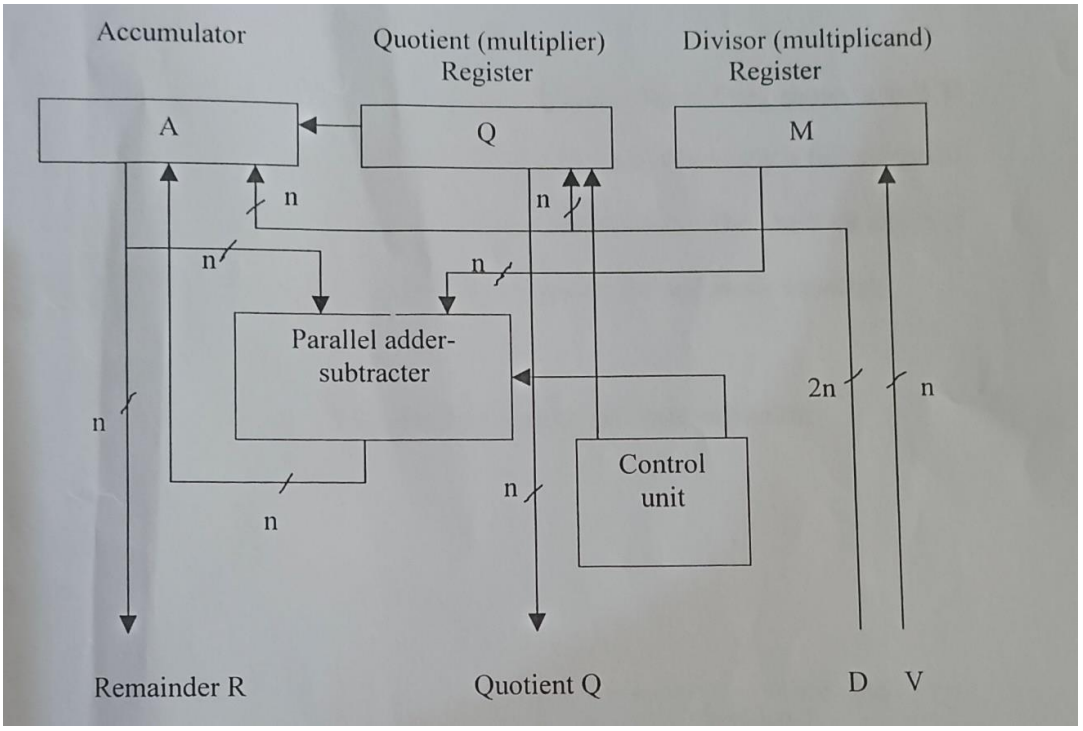


Fig 6.2 the data path of a sequential n-bit binary divider

Depending on whether the result is too big or too little, a floating-point operation results in overflow or underflow. By moving the mantissa of the result and changing its exponents, overflow or underflow caused by mantissa operations may often be rectified when employing floating point processing. One or more guard bits are temporarily connected to the right end of the mantissa in order to ensure correctness while doing floating-point nt calculations. X1 Xo, Xn-1, Xn-2, and Xn-3. Sometimes results must be rounded rather than reduced to n bits, for example, a guard bit x. Rounding is done by adding 1 to x0 and truncating the result to n bits. The bits that were relocated from the right end may be kept as guard bits when a mantissaright-shifted during the alignment step of addition or subtraction.

6.3 .Dividers design USA ing combinational array

Using circuits with combinational arrays, division may be done. Fig. 6.3 depicts a cell D that might be used to produce a restored division technique (a). This cell is a full subtracter, with t and u acting as the borrow-in and borrow-out bits, respectively. The main output z is controlled by input a. When a=1, the difference bit, designated by the mathematical equation as z, is 1.

Since z = x minus y minus t, z equals x because a = 0m. Hence, the behavior of cell D is determined by the logic equations.

Z = x + a (y @ t), xy + xt, and yt

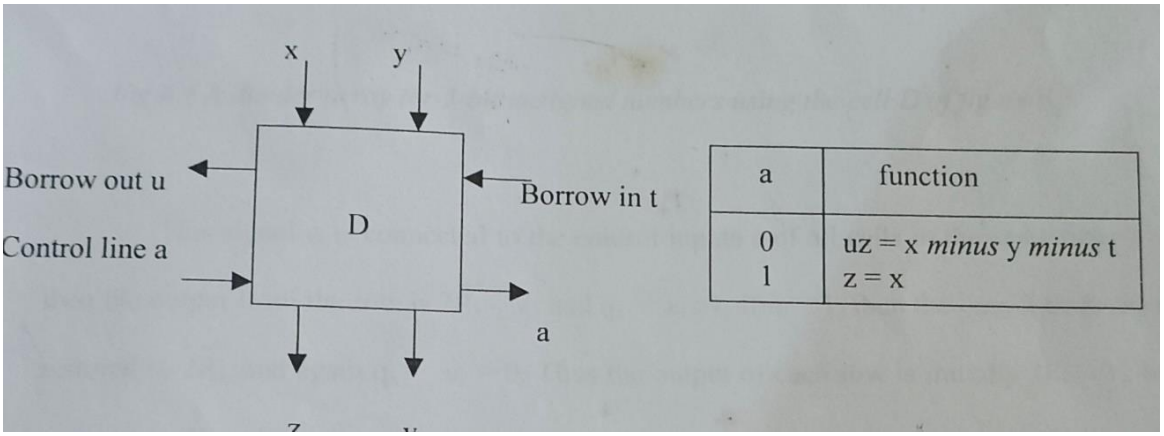


Fig no 6.3 a cell D for array implementation of restoring division

An array of D cells for splitting 3-bit unsigned integers into 4-bit quotients is shown in Fig. 6.4.

V is subtracted from the shifted partial residual in each row of the array.

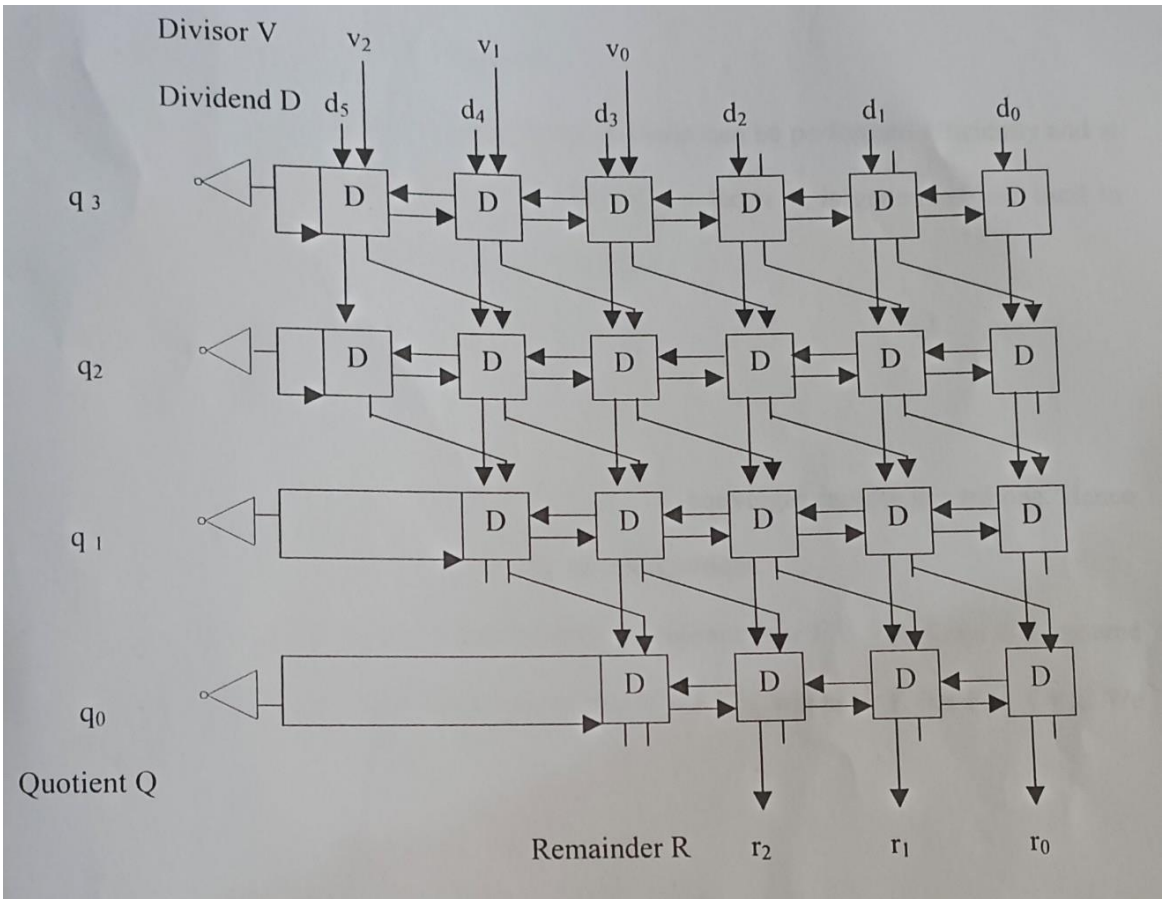


Fig no. 6.4 A divider array for 3-bit unsigned number using the cell D of figure 6.3

2R, which was produced by the row before. The sign of the result and, therefore, of the quotient bit.out signal is determined by the borrow-word from the row's leftmost cell. Figure 6.4 displays a 3-bit unsigned number divider array utilizing cell D from figure 6.3.

This signal, u, is connected to the control inputs of every cell in a row. If u, = 0, the row outputs 2R, V and q, u, 1 respectively. If u, 1, the row's output is reset to 2R, and q, u is once again equal to 0, the result is true. As a consequence, the output of each row initially begins as 2R, -V and is then transformed back to 2R as needed. Restoration is carried out via overriding the row's subtraction rather than explicitly inserting the divisor back.

Let d and d' stand for a cell's carry (borrow) propagation and restore times, respectively.

Assume that the dividend and divisor are both n bits long. As each row of the divider array functions as an n-bit ripple borrow subtractor, the calculation of one quotient bit takes the longest, nd + d'. As a result, the calculation time for an m-bit quotient and its associated remainder is m(nd+d'), and m(n+1)-1 cells are required.

6.4 Division by repeated multiplication

Repetition of multiplication may be used to divide effectively and cheaply in systems with a high-speed multiplier. A factor F is generated after each iteration, which is then multiplied by the dividend D and the divisor V.

Hence, Q=D* Fo* F1* F2 *

. V* Fo* F1* F2*.

In order for the sequence

V * Fo* F1* F2* D* Fo* F1* F2* to converge toward the required quotient, the factor F is selected.

The choice of the Fi's affects the method's ability to converge. For the sake of simplicity, consider the case where D and V are positive normalized fractions, resulting in V = 1- x, where x 1. Decide Fo= 1 + x. We quickly come together as one. As a result, we may write

V* Fo=(1-x) (1 + x) = 1 - x2

Now V Fois is more like one than like V. Set F1 next to 1 plus x2 Hence

V* Fo* F1 is equal to (1-x2)(1+x2) = 1

and so on. Let V1 stand for V* Fo* F1 F2.....Fi.

At each level, the multiplication factor is calculated as F1 = 2 - Vi-1, which is just V1-two-s 1's complement. V1 soon approaches unity as I rises. The operation is complete when V1 equals the closest integer to one for the chosen word size, which is between 0.11 and 111.

Conclusion

An improved throughput was achieved by utilizing the pipelining techniques in order to increase the processors' performance by overlapping the multiple instruction execution. The instructions were broken down into the stages whereas each stage was executed in a separate unit within the processor. This allowed several instructions to be executed in parallel, which increased the overall performance of the processor.

It was observed that by introducing the pipelining into the super scalar processors, numerous instructions can be issued and executed in parallel. Similarly, in DSP processor, the overall performance was increased by allowing the processor to run multiple instructions at the same time. However, in DSP processors, the payload is huge and many instructions are inter-dependent, and have to be executed in a pre-defined order. This requires complex algorithmic setup in order to achieve parallelism and performance at the same time. This will be covered in future work.

Overall, pipelining increases performance in both super scalar and DSP processors with a minimum catch due to inter-dependency of instructions, which can also be reduces by writing specialized and complex DSP algorithms by dividing the instructions in smaller units and pre-fetching the independent units and executing them in advance and then conclude the results in sequence. Thus, improving the overall throughput of the processor.

REFERENCES

1. Hennessy, J, and Patterson, D. [1996]. Computer Architecture A Quantitative Approach, Morgan Kaufrnann, San Francisco, CA.
2. John Sweeney, Superscalar Techniques Applied to Digital Signal Processing ICSPAT, Toronta, Canada, 1998, pp. 615-619.
3. C. Barrent, D. Dill, and J. Levitt. A decision procedure for bit=vector arithmetic. In proceedings of ACM/IEEE Design Automation Conference (DAC'98), pages 522-527/ACM, 1998.
4. R. Bryant Graph-based algorithms for Boolean function manipulation. IEEE Transaction on Computers, C-35 (8): 677-691 August 1986.
5. M. Bickford and M.srivas. Verification of a pipelined microprocessor using CLIO. In Proceeding of Workshop on Hardware Specification, Verification and Synthesis:
6. Mathematical Aspects. Springer, 1989.
7. John P. Hayes. Computer Architecture and Organization, McGRAW W. HILL INTERNATIONAL EDITION.
8. William Stallings. Computer Organization and Architecture. Prentice Hall International Edition.
9. M. Moris Mano. Digital logic and Computer Design.
10. Floyd. Digital Fundamentals. Prentice Hall International Edition.
11. ALLIANT COMPUTER SYSTEMS CORP. [1987]. Alliant FX/Series. Product summary (June). Action. Mass.
12. BANERJEE, U. [1979]. Speedup of ordinary programs. Ph.D. Thesis. Dept of Computer Science. Univ. of Illinois at Urbana-Champaign (October).

13. BUCHER. I. Y. [1983]. “The Computational speed of supercomputers.” Proc. SIGMETRICS Conf. On Measuring and Modeling of Computer Systems. ACM (August). 151-165.

14. DONGARRA. J.J. [1986]. “A survey of high performance processors.” COMPCON, IEEE (March), 8-11.

15. FLYNN.M.J.[1996]. “Very high speed computing system.” Proc. IEEE 54:12 (December), 1901-1909.

16. SCHNECK. B.J. [1981]. Super processor Architecture, Kluwer Academic Publishers, Norwell, Mass.

17. WATSON. W.J. [1972]. “The TI ASC-A highly modular and flexible super processor architecture.” Proc. AFIPS Fall Joint Computer Conf. 221-228

APPENDICES

Appendix A

Verilog Code for Bit Serial Adder

```
/////design for one bit full adder

module fulladd(sum, c_out,a,b,c_in);

input a,b,c_in;

output sum,c_out;

wire p,g,c;

xor(p,a,b);

and(g,a,b);

xor(sum,p,c_in);

and(c,p,c_in);

or(c_out,c,g);

endmodule

                ///design of d-flipflop///

module dff(q,d,clk);

output q;

input d,clk;
```

```
reg q;

always (posedge clk)

q<=d;

endmodule

                ///design of serial in parallel out shift register///

module serial_to_parallel(q0,q1,q2,q3,shift,d,clk);

output q0,q1,q2,q4;

input d;

input clk,shift;

wire a;

assign a = shift;

//instantiate d-flipflops

dff dff0(q0,d,clk);

dff  dff1(q1,q0,clk);

dff  dff2(q2,q1,clk);

dff  dff3(q3,q2,clk);

endmodule
```

```
////design of parallel in serial out shift register///

module parallel_in_serial_out(q3,shift_load,d0,d1,d2,d3,clk);

output q3;

Input d0,d1,d2,d3,clk,shift_load;

wire a,a_inv;

wire   x,y,z,q2,q1,q0;

assign   a=shift_load,

a_inv=~ (shift_load),

x=(q0 & a) | (d1 & a_inv),

y=(q1 & a) | (d2 & a_inv),

z=(q2 & a) |(d3 & a_inv);

dff dff1(q0,d0,clk);

dff  dff1(q1,x,clk);

dff  dff2(q1,x,clk);

dff  dff3(q2,y,clk);

dff  dff4(q3,z,clk);

endmodule
```

```
                ///stimulus for bit serial adder///

module stimulus;

reg [3:0] A,B;

reg clk,shift,shift_load;

wire SUM, c_in;

wire C_OUT,q0,q1;

wire p0,p1,p2,p3;

///instantiate serial in parallel out shift register///

serial_to_parallel s_to_p(p0,p1,p2,p3,shift,SUM,clk);

///instantiate parallel in serial out shift register///

parallel_in_serial_out p_to_s0(q0,shift_load,A[0],A[1],A[2],A[3],clk);

parallel_in_serial_out p_to_s1(q1,shift_load,B[0],B[1],B[2],B[3],clk);

///instantiate one bit full adder///

fulladd FA(SUM,C_OUT,q0,q1,c_in);

//instantiate d-flipflop for storing carry out//

dff dff5(c_in,C_OUT,clk);

///set up the monitoring for the signal values///
```

```
initial

begin

$monitor($time,"A=%b,B=%b,C_IN=%b,…..C_OUT=%b,SUM=%b/n",q0,

q1,C_OUT,SUM);

end

 //stimulate inputs

initial

begin clk=1'b0;

A=4'b0011;B=4'b0111;

shift<=1;

shift_load=0

#2 shift_load=1;

End

always #1 clk=~clk;

initial

begin

$vw_dumpvars;
```

```
#30 $stop;

end

endmodule
```

Verilog Code for Ripple Carry Adder

```
///design for fulladder///

module fulladd(sum,c_out,a,b,c_in);

output sum,c_out;

input a,b,c_in;

wire s1,c1,c2;

xor(s1,a,b);

and(c1,a,b);

xor(sum,s1,c_in);

and(c2,s1,c_in);

or(c_out,c2,c1);

endmodule

///design for Ripple Carry Adder///

module ripple_carry_adder(sum,c_out,x,y,c_in);
```

```
input [3:0] x,y;

input c_in;

output [3:0]sum;

output c_out;

wire c1,c2,c3,c4;

//instantiate full adders

fulladd FA1(sum[0],c1,x[0],y[0],c_in);

fulladd FA2(sum[1],c2,x[1],y[1],c1);

fulladd FA3(sum[2],c3,x[2],y[2],c2);

fulladd FA4(sum[3],c4,x[3],y[3],c3);

assign c_out=c4;

endmodule

        ///stimulus for Ripple Carry Adder///

module stimulus;

reg [3:0] x,y;

reg c_in;

wire c_out;
```

```
wire [3:0]sum;

//instantiate the design of ripple carry adder

ripple_carry_adder R_C_A(sum,c_out,x,y,c_in);

initial

begin

$monitor("\t\t %b<-C_IN\n A %b\n B %b\n\t……\nC_OUT->%b%b<-SUM\n",c_in,x,y,c_out,sum);

end

//stimulate inputs

initial

begin

x=4'b1001;4'b1101;c_in=0;

#5 x=4'b1110;y=4'0110;c_in=0;

end

endmodule
```

Verilog Code for Pipelined Ripple Carry Adder

```
///1-bit full adder///

MODULE FULLADD(SUM,C_OUT,A,B,C_IN);
```

```
//I/O port declaration

outpur sum,c_out;

input a,b,c_in;

//internal nets

Wire s,c1,c2;

Xor(s,a,b);

and(c1,a,b);

xor(sum,s,c_in);

and(c2,s,c_in);

or(c_out,c2,c1);

endmodule

                ////D – Flip Flop////

//define d-flipflop

module dff(q,d,clk);

output q;

input d,clk;

reg q;
```

```
always @(posedge clk)

q<=d;

endmodule

                ////serial in serial out shift register for storing numbers///

//shift register for storing number A0

module serial_in_serial_out_1(p0,a0,clk);

input a0,clk;

output p0;

dff dff A_0(p0,a0,clk);

endmodule

//shift register for storing number B0

module seria_in_serial_out_2(q0,b0,clk);

input b0,clk;

output q0;

dff dffB_0(q0,b0,clk);

endmodule

//shift register for storing number A1
```

```
module serial_in_serial_out_3(p1,a1,clk);

input a1,clk;

output p1;

wire A1;

dff  dffA1_0(A1,a1,clk);

dff  dffA1_1(p1,A1,clk);

endmodule

//shift register for storing number B1

module serial_in_serial_out_4(q1,b1,clk);

input b1,clk;

output q1;

wire B1;

dff  dff1_B0(B1,b1,clk);

dff  dff1_B1(q1,B1,clk);

endmodule

//shift register for storing number A2

module serial_in_serial_out_5(p2,a2,clk);
```

```
input a2,clk;

output p2;

wire A2,A_2;

dff dff2_A0(A2,a2,clk);

dff dff2_A1(A_2,A2,clk);

dff dff2_A2(p2,A_2,clk);

endmodule

//shift register for storing number B2

module serial_in_serial_out_6(q2,b2,clk);

input b2,clk;

output q2;

wire B2,B_2;

dff dff2_B0(B2,b2,clk);

dff dff2_B1(B_2,B2,clk);

dff dff2_B2(q2,B_2,clk);

endmodule

//shift register for storing number A3
```

```
module serial_in_serial_out_7(p3,a3,clk);

input a3,clk;

output p3;

wire A3,A_3,a_3;

dff dff3_A0(A3,a3,clk);

dff dff3_A1(A_3,A3,clk);

dff dff3_A2(a_3,A_3,clk);

dff dff3_A3(A3,a_3,clk);

endmodule

//shift register for storing number B3

module serial_in_serial_out_8(q3,b3,clk);

input b3,clk;

output q3;

wire B3,B_3,b_3;

dff dff3_B0(B3,b3,clk);

dff dff3_B1(B_3,B3,clk);

dff dff3_B2(b_3,B_3,clk);
```

```
dff dff3_B3(B3,b_3,clk);

endmodule

        //shift register for storing sum//

        //shift register for storing sum S0//

module serial_to_serial_0(SUM[0],s0,clk);

input s0,clk;

output SUM[0];

wire A,B,C;

dff  dff_s0_0(A,s0,clk);

dff   dff_s0_1(B,A,clk);

dff dff_s0_2(C,B,clk);

dff dff_s0_3(SUM[0],C,clk);

endmodule

        //shift register for storing sum S1//

module serial_to serial_1(SUM[1],S1,clk);

input s1,clk;

output SUM[1];
```

```
wire D,E;

dff dff_s1_0(D,s1,clk);

dff dff_s1_1(E,D,clk);

dff dff_s1_2(SUM[1],E,clk);

endmodule

                //shift register for storing sum S2//

module serial_to_serial_2(SUM[2],s2,clk);

input s2,clk;

output SUM[2];

wire F;

dff dff_s2_0(F,s2,clk);

dff dff_s2_1(SUM[2],F,clk);

endmodule

                //shift register for storing sum S3//

module serial_to_serial_3(SUM[3],s3,clk);

input s3,clk;

output SUM[3];
```

```
dff dff_s3_0(SUM[3].s3.clk);

endmodule

                //stimulus for pipelined adder//

module stimulus;

reg [3:0],A,B;

reg clk,C_IN;

wire [3:0] SUM;

wire p0,q0,p1,q1,p2,q2,p3,q3;

//instantiate four 1-bit full adders and registers for carry out from one full adder to carry in for next full adder

fulladd FA0(SUM[0],C_OUT0,A0,B0,C_IN);

dff R0(c_in1,C_OUT0,clk);

fulladd FA1(SUM[1],C_OUT1,B1,c_in1);

dff R1(c_in2,C_OUT1,clk);

fulladd FA2(SUM[2].C_OUT2,A2,B2,c_in2);

dff R2(c_in,C_OUT2,clk);

fulladd FA3(SUM[3],C_OUT,A3,B3,c_in3);

//instantiate the shift registers for numbers A and B
```

```
serial_in_serial_out_1S_to_S1(p0,A[0],clk);

serial_in_serial_out_2S_to_S2(q0,B[0],clk);

serial_in_serial_out_3S_to_S3(p1,A[1],clk);

serial_in_serial_out_4S_to_S4(q1,B[1],clk);

serial_in_serial_out_5S_to_S5(p2,A[2],clk);

serial_in_serial_out_6S_to_S6(q2,B[2],clk);

serial_in_serial_out_7S_to_S7(p3,A[3],clk);

serial_in_serial_out_8S_to_S8(q3,A[3],clk);

//instantiate the shift registers for storing sums

serial_to_serial_0s_to_s0(SUM[0],s0,clk);

serial_to_serial_1s_to_s1(SUM[1],s1,clk);

serial_to_serial_2s_to_s2(SUM[2],s2,clk);

serial_to_serial_3s_to_s3(SUM[3],s3,clk);

//set up the monitoring for the signal values

initial

begin

$monitor("\t\t %b<-C_IN\nA %b\nB   %b\n\t …….\nC_OUT->%b%b<-

SUM",C_IN,A,B,C_OUT,SUM);
```

```
End

//stimulate inputs

initial

begin

A=4'd0;B=4'd3;C_IN=0;

#5 A=4'd7;B=4'd5;

#5 A=4'd9;B=4'd10;

#5 A=4'd6;B=4'd4;

end

initial

begin

$vw_dumpvars;

#240 $stop;

end

endmodule
```

Verilog Code for Carry Look Ahead Adder

```
//fulladder
```

```
module fulladd4_1(sum,c_out,a,b,c_in);

//inputs and outputs

output [15:0] sum;

output c_out;

input [15:0] a,b;

input c_in;

//internal wires

wire p0,g0,p1,g1,p2,g2,p3,g3;

wire c4,c3,c2,c1;

//compute p for each stage

assign p0=a[0]^b[0],

p1=a[1]^b[1],

p2=a[2]^b[2],

p3=a[3]^b[3];

//compute the g for each stage

assign g0=a[0]&b[0],

g1=a[1]&b[1],
```

```
g2=a[2]&b[2],

g3=a[3]&b[3];

//compute the carry for each stage assign

c1=(g0|p0&c_in), c2=(g1|p1&g0)|(p1&p0&c_in),

c3=(g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&c_in)),

c4=(g3|p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&c_in);

//compute sum

assign sum[0]=(p0^c_in),

sum[1]=(p1^c1),

sum[2]=(p2^c2),

sum[3]=(p3^c3);

//assign carry output

assign c_out=c4;

endmodule

module stimulus;

//set up varialbles
```

```
reg [15:0] a,b;

reg c_in;

wire [15:0] sum;

wire c_out;

//instantiate the 4-bit full adder

fulladd4_1 fa1_4(sum[3:0],c1,a[3:0],b[3:0],c_in);

fulladd4_1 fa2_4(sum[7:4],c2,a[7:4],b[7:4],c1);

fulladd4_1 fa3_4(sum[11:8],c3,a[11:8],b[11:8],c2);

fulladd4_1 fa4_4(sum[15:12],c_out,a[15:12],b[15:12].c3);

//setup the monitoring for the signal values

initial

begin

$monitor("\t\t\t\t\t %b<-C_IN\n A %b\nB %b\n\t\t……\nC_OUT->%b%b<-SUM\n",c_in,a,b,c_out,sum);

End

//stimulate inputs

initial

begin
```

```
a=16'd1279;b=16'd1515;c_in=0;

#5 a=16'd4376;b=16'd2345;c_in=0;

#5 a=16'd7834;b=16'd4792;c_in=1;

end

endmodule
```

Verilog Code for Carry Save Adder

```
module Half_adder(sum,c_out,x,y);

input x,y;

output sum,c_out;

xor(sum,x,y);

and(c_out,x,y);

endmodule
```

////design for one bit full adder////

```
module Full_Adder(sum,c_out,a,b,c_in);

input a,b,c_in;

output sum,c_out;

wire p,g,c;
```

```
xor(p,a,b);

and(g,a,b);

xor(sum,p,c_in);

and(c,p,c_in);

or(c_out,c,g);

endmodule

                ////design for carry save adder////

module Carry_Save(sum,c_out,w,x,y,z);

input [3:0]w,x,y,z;

outpur [5:0] sum;

output c_out;

wire s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13;

wire c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13;

//instantiate the full adders for adding w,x,y and z as first stage

Full_Adder  FA0(s0,c0,w[0],x[0],y[0]);

Full_Adder  FA1(s1,c1,w[1],x[1],y[1]);

Full_Adder FA2(s2,c2,w[2],x[2],y[2]);
```

```
Full_Adder FA3(s3,c3,w[3],x[3],y[3]);

//instantiate the full adders for adding sum and carries for first stage and z

Half_Adder HA0(s4,c4,s0,z[0]);

Full_Adder FA4(s5,c5,s1,c0,z[1]);

Full_Adder FA5(s6,c6,s2,c1,z[2]);

Full_Adder FA6(s7,c7,s3,c2,z[3]);

Half_Adder HA4(s8,c8,c3,c7);

//instantiate the half adders for adding sum and carries from second stage

Half_Adder HA1(s9,c9,s5,c4);

Full_Adder FA8(s10,c10,s6,c5,c9);

Full_Adder FA9(s11,c11,s7,c6,c10);

Full_Adder FA10(s12,c12,s8,c7,c11);

Half_Adder HA2(s13,c13,c8,c12);

assign sum=(s13,s12,s11,s10,s9,s4);

c_out=c13;

endmodule
```

////stimulus for carry save adder////

```
module stimulus;

reg [3:0]w,x,y,z;

wire [5:0] sum;

wire c_out;

//instantiate the design of carry save adder

Carry_Save C_S_A(sum,c_out,w,x,y,z);

initial

begin

$monitor("\t\tw %b\n\t\tx %b\n\t\ty %b\n\t\tz %b\n\t….\nc_out

%b%bsum",w,x,y,z,c_out,sum);

end

//stimulate inputs

initial

begin

w=4'b1001;x=4'b0011;y=4'b0101;z-4'b1101;

end

endmodule
```

Appendix B

////design for fulladder////

```
module fulladd(sum,c_out,a,b,c_in);

output sum,c_out;

input a,b,c_in;

wire s1,c1,c2;

xor(s1,a,b);

and(c2,s1,c_in);

or(c_out,c2,c1);

endmodule
```

////design for two's complement subtractor////

```
module subtractor(z,c_out,x,y,c_in);

input [3:0] x,y;

input c_in;

output [3:0]z;

output c_out;

wire s,c0,c1,c2;

wire a0,a1,a2,a3;
```

```
wire b0,b1,b2,b3;

wire cout1,cout2,cout3,cout4;

assign b0=1;

b1=0;

b2=0;

b3=0;

assign s=1;

xor(a0,x[0],s);

xor(a1,x[1],s);

xor(a2,x[2],s);

xor(a3,x[3],s);

//instantiate the full adders for adding 1 in 1's complement of Y

fulladd FA1(z[0],c0,a0,y[0],c_in);

fulladd FA2(z[1],c1,a1,y[1],c0);

fulladd FA3(z[2],c2,a2,y[2],c1);

fulladd FA4(z[3],c_out,a3,y[3],c2);

endmodule
```

```
////stimulus for two's complement subtractor////

Module stimulus;

reg [3:0]x,y;

reg c_in;

wire [3:0]z;

wire c_out;

//instantiate the design for two's complement subtractor

subtractor T_C_S(z,c_out,x,y,c_in);

initial

$monitor("\t\t\tX %b\n\t\t\t Y %b\n\t\t\t……\n\t\t\tZ

%b%b",x,y,c_out,z);

End

//stimulate input

initial

begin

x=4'b1110;y=4'b0110;c_in=1;

end

endmodule
```

Glossary

Some of the terms in this glossary are from the ANSI American National Dictionary for Information systems (1990)

A

Accumulator: The name of the CPU register in a single-address instruction format. The accumulator, or AC, is simplicity one of the operands for the instruction.

Addend: In addition, the number that is added to another number called the addend.

Adder: A logic circuit used to add two binary numbers.

Arithmetic and Logic Unit (ALU): Arithmetic Logic Unit; the key-processing element of the microprocessor that performs arithmetic and logic operations.

Analog: Being continuous or having continuous values, as opposed to having a set of discrete values.

AND: A basic logic operation in which a true (HIGH) output occurs only if all the input conditions are true (HIGH).

AND gate: A logic gate that produces a HIGH output only when all of the inputs are HIGH.

Architecture: The internal functional arrangement of the elements that give a device; its particular operating characteristics.

Array: In a PLD, a matrix formed by rows of product-term lines and columns of input lines with a programmable cell at each junction.

Augend: In addition, the number to which the addend is added.

B

Base: In the numeration system commonly used in scientific papers, the numbers that is raised to the power denoted by the exponent and then multiplied by the mantissa to determine the real number represented.

Binary: Having two values or states; describes a number system that has a bae of two and utilizes 1 and 0 as its digits.

Bit: A binary digit, which can be either 1 or 0.

Branch prediction: A mechanism used by the processor to predict the outcome of a program branch prior to its execution.

Buffer: Storage used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another.

Bus: A shared communication path consisting of one or a collection of lines. In some computer systems, a common bus connects CPU, memory and I/O

components. Since the lines are shared by all components, only one component at a time can successfully transmit.

Byte: A group of eight bits. Also referred to as an octet.

Cache Memory: A special buffer storage smaller and faster than main storage that is used to hold a copy of instructions and data in main storage that are likely to be needed next by the processor, and that have been obtained automatically from main storage.

Carry: The digit generated when the sum of two binary digits exceeds 1.

Carry generation: The process of producing an output carry in a full adder when both input bits are 1s.

Carry Propagation: The process of rippling an input carry to become the output carry in a full adder when wither or both of the input bits are 1s and the input carry is a 1.

Cascade: To connect “end-to-end” as when several counters are connected from the terminal count output of one counter to the enable input of the next counter.

Central Processing Unit (CPU): That portion of a computer that fetches and executes instructions, it consists of an arithmetic and logic unit (ALU), a controller unit and registers. Often simply referred to as a processor.

Circuit: An arrangement of electrical and/or electronic components interconnected in such a way as to perform a specified function.

Clear: An asynchronous input used to reset a flip-flop (make the Q output 0); to place a register or counter in the state in which it contains all 0s.

Clock: The basic timing signals in digital systems.

Code: A set of bits arranged in a unique pattern and used to represent such information as numbers, letters and other symbols.

Combinational Logic: A combination of logic gates interconnected to produce a specified Boolean function with no storage or memory capability some times called combinational logic.

Complement: The inverse or opposite of a number, in Boolean algebra, the inverse function, expressed with a bar over the variable. The complement of a 1 is a 0 and vice versa.

Conditional Jump: A jump that takes place only when the instruction that specifies it is executed and specified are satisfied.

Control Unit: That part of CPU that controls CPU operations, including ALU operations, the movement of data within the CPU, and the exchange of data and control signals across external interfaces (e.g., the system bus)

D

Data Bus: A bi-directional set of conductive paths on which data or instruction codes are transferred into the microprocessor or on which the result of an operation or computation is sent out from the microprocessor.

D flip-flop: A type of bi-stable multivibrator in which the output assumes the state of the D input line to several output lines in a specified time sequence.

Difference**:** The result of a subtraction.

Digit: A symbol used to express a quantity.

Digital: Related to digits or discrete quantities; having a set of discrete values as opposed to continuous values.

Direct Memory Access: A form of I/O in which a special module, called a DMA module, controls the exchange of data between main memory and an I/O module. The CPU sends request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

Dividend: In a division operation, the quantity that is being divided.

Divisor: In a division, a quantity that divides.

DMA: Direct Memory Access: a method to directly interface a peripheral device to memory without using the CPU for control.

DRAM: Dynamic Random Access Memory; a type of semiconductor memory that uses capacitors as the storage element and is a volatile, read/write memory.

E

Exclusive OR (XOR): A basic logic operation in which a HIGH occurs when the two inputs are at opposite levels.

Executive: A CPU process in which an instruction is carried out.

Exponent: A part of floating point number that represent the number of places that the decimal point (or binary point) is to be moved.

Feed Back: The output voltage or a portion of it that is connecting back to the input of a circuit.

Fetch: A CPU processes in which an instruction is obtained from the memory.

FIFO: First in First Out memory.

Flip-Flop: A basic storage circuit that can store only bit at a time; a synchronous bistable device.

Floating-point number: A number representation based on scientific notation in which the number consists of an exponent and mantissa.

Full Adder: A digital circuit that adds two bits and an input carry to produce a sum and an output carry.

Gate: A logic circuit that performs a specified logic operation, such as AND or OR; one of the three terminals of a field affect transistor.

Half Adder: A digital circuit that adds two bits and produces a sum and an output carry. It cannot handle input carries.

Hardware: The circuits and physical components of a computer system (as composed to the direction called software).

Instruction: One step in a computer program; a unit information that tells the CPU what to do.

Integer: A whole number.

Interrupt: A signal or instruction that causes the current processes to be temporarily stopped while a service outline is run.

L

Logic: In digital electronic, the decision-making capability of gate circuit, in which a HIGH represent a true statement a LOW represents a false one.

Look Ahead Carry: A method of a binary addition whereby carries from proceeding adder stages are anticipated, the elimination carry propagation delays.

LSI: Large Scale Integration; a level of IC complexity in which there are 100 to 9999 equivalent gates per chip.

Magnitude: The size or value of a quantity.

Mantissa: The magnitude of a floating-point number.

Micro Processor: A digital integrated circuit device that can be programmed with a series of instruction to perform the specified data.

Minuend: The number from which another number is subtracted.

Multiplexer (MUX): A circuit (digital device) that switches digital data from several input lines onto a single output line in a specified time sequence.

Multiplicand: The number that multiplies the multiplicand.

Operands: A variable, register, a memory location, or a value used in an assembly language program as part of the instruction.

Output: A signal or line coming out of a circuit.

Parallel: A digital system, data occurring simultaneously on several lines, a transfer or processing of several bytes simultaneously.

Pipeline: As applied to memories, an implementation that allows a read or writes operation to be initiated before the previous operation is completed.

Quotient: The result of division.

RAM: Random Access Memory; a volatile read/write semiconductor memory.

Remainder: The amount left over after a division.

Ripple Carry: A method of binary addition in which the output carry from each adder becomes the input carry of the next higher-order adder.

Shift: To shift binary data from stage within a shift register or other storage device or to move binary data into or out of the device.

Subtractor: A logic circuit used to subtract two binary numbers. **Sum:**

The result when two or more numbers are added together.

Throughput: The average speed through which a progam is executed.

ISBN: 978-625-95806-3-0