Palestine Polytechnic University

College of Engineering & Technology
Electrical & Computer Engineering Department

Graduation Project

Control Unit: Study, Compare, and Discuss the Most Recent Control Units

Project Team
Asma Shehadeh Tahani Ghanam Suha Abu Al-Jadayel

Project Supervisor
Eng. Mazen Zalloum

Hebron-Palestine

June- 2005

Palestine Polytechnic University

College of Engineering & Technology
Electrical & Computer Engineering Department

Graduation Project

Control Unit: Study, Compare, and Discuss the Most Recent Control Units

Project Team
Asma Shehadeh Tahani Ghanam Suha Abu Al-Jadayel

Project Supervisor
Eng. Mazen Zalloum

Hebron-Palestine

June- 2005

Palestine Polytechnic University
Hebron-Palestine

College of Engineering & Technology
Electrical & Computer Engineering Department

Control Units: Study, Compare, and Discuss the Most Recent Control Units

Project Team
Asma Shehadeh Tahani Ghanam Suha Abu Al-Jadayel

Upon the direction of the supervisor and the approval of discussion committee members, this project is submitted of the Electrical and Computer Department at Palestine Polytechnic University as a partial fulfillment for B.E Degree in Computer Engineering.

Supervisor signature

.....

Discussion committee member's signatures

....

Head of the Department signature

.....

Palestine Polytechnic University
Hebron-Palestine

College of Engineering & Technology
Electrical & Computer Engineering Department

Control Units: Study, Compare, and Discuss the Most Recent Control Units

Project Team
Asma Shehadeh Tahani Ghanam Suha Abu Al-Jadayel

Upon the direction of the supervisor and the approval of discussion committee members, this project is submitted of the Electrical and Computer Department at Palestine Polytechnic University as a partial fulfillment for B.E Degree in Computer Engineering.

Supervisor signature

.....

Discussion committee member's signatures

....

Head of the Department signature

.....

Abstract

Control Unit: Study, Compare, and Discuss the Most Recent Control Units

By

Asma Shehadeh Tahani Ghanam Suha Abu Al-Jadayel

Supervisor

Eng. Mazen Zalloum

Control Unit: Study, Compare, and Discuss the Most Recent Control Units research includes an introduction to computer system in general, the research also discuss the most resent architecture RSC, CISC and superscalar machines and the characteristics for each machine.

The main am for this research is to find a proper simulation that will help us to determine which architecture s the best. In addition, will contain names for simulators, which help in tracing the organization of any architecture to study its performance.

بحث وحدة التحكم : دراسة، مقارنة ومناقشة لاحدث وحدات التحكم يناول مقدمة عامة عن نظام الحاسوب والبحث يناول ايضاً دراسة لاحدث العمارات (CISC, RISC, Superscalar).

الهدف الاساسي من البحث هو ايجاد البرنامج المناسب الذي يساعد في تحديد أي عمارة هي الافضل.

يتضمن البحث ايضاً أسماء لبرامج تساعد في عملية تتبع تنظيم العمارة لدراسة الإنجاز

الإهداء

إلى الذين بذلوا من اجلي الكثير

إلى الذين ضحوا براحتهم من اجل أن أكون

إلى الذين حملوا على راحتيهم الشموع وفي مقلتيهم الدموع دموع الفرح وشموع المستقبل

أمي وأبي

إلى من هم سندي وذخري ماضيا وحاضرا ومستقبلا

إلى أساتذتي الأفاضل

نهديهم جميعا هذا العمل المتواضع

فريق البحث

شكر وتقدير

تتقدم بجزيل الشكر وعظيم الأمتنان الى الصرح العلمي الشامخ جامعة بوليتكنيك فلسطين والى الهيئة التدريسية في دائرة الهندسة الكهربائية والحاسوب ونخص بالذكر رئيس الدائرة د . غاندي مناصرة ومشرف المشروع الاستاذ م . مازن زلوم والى كل من مد يد العون لأنجاز هذا العمل .

والشكر أولا وأخير لله عز وجل على اتمام هذا المشروع

Contents

List of Figures

List of Tables

1

Introduction

Chapter 1

Introduction

1.1 General Idea about Project and its Importance

The project study the most important part in the computer system inside CPU, it is the control units (CUs) its structure and functions. This study expanded to touch the most recent architectures and those are: Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC), and SuperScalar architecture, and differentiate between the three architectures and try to determine which is the most one is better than others according to the performance.

1.2 Project Objectives

Our project has the following purposes:

1. Study and understand the control units structure and function.
2. Study the differences between the three architectures CISC, RISC, and superscalar
3. Simulation that will help in determining which architecture is the better according to its performance.

1.3 Literature Review

The following literature review concerned with our project:

1. "Application of Transient Signal Analysis to a RISC Microprocessor"
   By Dhruva Acharyya, Chintan Patel, Smita Pawar, Abhishek Singh and Jim Plusquellic

In this paper, hardware data is presented from a set of custom- designed 16-bit RISC microprocessor chips. The analysis supports previous simulation and hardware results obtained from a method that analyzes supply rail transient waveforms as a means of identifying faulty chips.

2. "Improving CISC Instruction Decoding Performance Using a Fill Unit"

   By Mark Smotherman Manoj Franklin

   Current superscalar processors, both RISC and CISC, require substantial instruction fetch and decode bandwidth to keep multiple functional units utilized. While CISC instructions can sometimes provide reduced fetch bandwidth requirements, they are correspondingly more difficult to decode.

   This design is accompanied by a microengine-register allocation and renaming scheme that prevents the increased supply of microoperations from placing excessive demands on the normal register renaming hardware.

1.4 Time Plan

Table1.1 shows the tasks according to the total time

Tasks	Period\week during (20-Sep-2004 To 20-Jun-2005)															
	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
determine the project idea																
collect data																
prepar the introduction and the outline																
update the collected data																
write the research to be discuss																
search for basic details and information																
prepar for simulation																
prepare for the final copy for research																

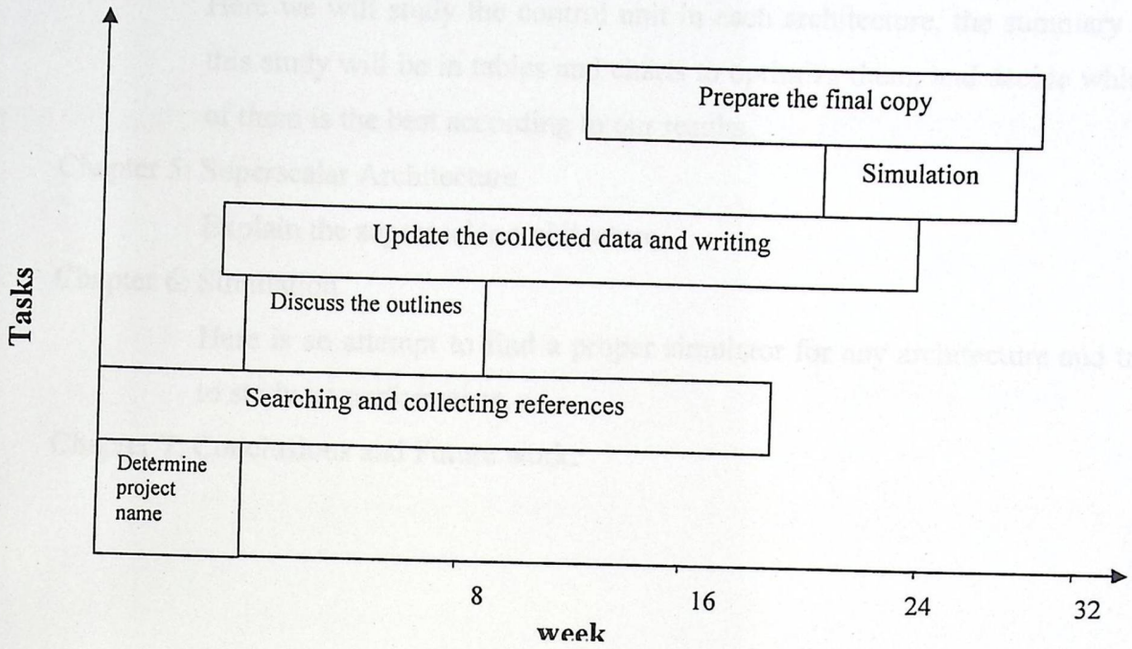

Figure1.1 a chart represent the task per week

1.5 Report contents

The project consists of seven chapters that are distributed according to the following:

Chapter 1: Introduction

This chapter talks about the computer in general, its hardware and software components.

Chapter 2: Introduction to Computer Systems

This chapter talks in more details about the basic components of the computer system and its central processing unit (CPU).

Chapter 3: Control Unit Structure and Functions

This chapter is project heart; here the internal structure of control unit is taken in details. In addition, to describe how CU will execute instructions and the control unit different types.

Chapter 4: RISC and CISC Architectures

Here we will study the control unit in each architecture, the summary of this study will be in tables and charts to optimize them, and decide which of them is the best according to our results.

Chapter 5: Superscalar Architecture

Explain the superscalar architecture.

Chapter 6: Simulation

Here is an attempt to find a proper simulator for any architecture and try to study its performance.

Chapter 7: Conclusions and Future work.

2

Introduction to Computer Systems

Chapter 2

Introduction to Computer Systems

2.1 Overview

This chapter provides a general introduction to computers systems. A computer system is made up of both hardware and software. Software controls the computer and makes it do useful work. Hardware refers to the physical components that make up a computer system. These include the computer's processor, memory, monitor, keyboard, mouse, disk drive, and printer. This chapter will evolve the basic components of computer system.

2.2 Hardware

As shown in Figure 2.1, the hardware of a computer system is made up of a number of electronic devices connected together.

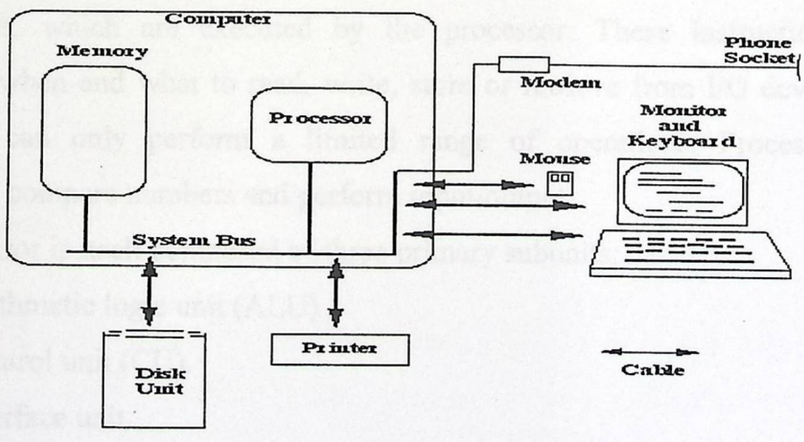

Figure 2.1: Basic hardware components

A computer has two major internal components namely its processor and its memory. The term device is used to describe any piece of hardware that we connect to a computer such as a keyboard, monitor, disk drive, and printer. Such devices are also sometimes described as peripheral devices or simply peripherals. They may be classified as input/output (I/O) devices and storage devices. I/O devices are responsible for communicating with the computer, providing input for the computer to process and arranging to display output for computer users. The keyboard and mouse are commonly used input devices. The monitor is the commonest output device, followed by the printer for hardcopy output. Storage devices are used to store information in a computer system. The memory is used to store information inside the computer while the computer is switched on. Disk storage is the commonest form of external storage, followed by the tape storage. External storage devices can store information indefinitely or more realistically, for some number of years. A very important component of a computer system is the system bus, which is used to transfer information between all system components.

2.2.1 The Processor

The processor executes computer programs. Computer programs composed of instructions, which are executed by the processor. These instructions tell the processor when and what to read, write, store or retrieve from I/O devices. So the processor can only perform a limited range of operations. Processor can do arithmetic, compare numbers and perform input/output.

The processor is itself composed of three primary subunits:

1. Arithmetic logic unit (ALU)
2. Control unit (CU).
3. Interface unit.
4. Storage locations (Registers).

The processor can be classified according to the register size, the processor with register size of n-bits are called n-bit processor. So that processors with 8-bit register

are called 8-bit processors. The greater the number of bits the more powerful the processor is, since it will be able to process a larger unit of information in a single operation. An n-bit processor will usually be capable of transferring n-bits to or from memory in a single operation. This number of bits is also referred to as the memory word size.

An alternative method of classifying a processor is to use the width of the data bus in which case an n-bit processor describes one operating with a data bus of n-bits. This means that the CPU can transfer n-bits to another device in a single operation. Using this classification. The data bus width is very important in a computer system, since it determines the amount of information that can be transferred to or from the CPU, in a single operation. I/O devices and memory operate at very slow speeds compared to the speed of the CPU. As a result, the CPU is frequently delayed by these slower devices, waiting for information to be transferred along the data bus. So, the more information we can transfer in a single operation, between an I/O devices and the CPU, the less time the CPU will spend waiting for information to process.

In addition, to what are mentioned about Central Processing Unit (CPU) components the clock is an important part in it. The clock controls the rate at which the CPU carries out activities. It generates a stream of cycles. The more cycles per second, the more actions that the CPU can carry out. The speed of the clock is measured in millions of cycles per second. One cycle per second is one Hertz (Hz)

2.2.2 Memory

For the CPU to function efficiently it needs a place to store data and instructions. Main memory is the internal storage component of a computer. The amount is important in determining the software that can be used because each program requires a specific amount of memory to be functional. The two type of main memory are random access memory and read only memory.

The most important characteristics of memory are capacity and performance. Three performance parameters are used:

- Access time: for RAM this is the time it takes to perform a read or write operation; that is the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-RAM access time is the time it takes to position the read-write mechanism at the desired location.
- Memory cycle time: This concept is primarily applied to RAM and consist of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively.
- Transfer rate: the rate at which data can be transferred into or out of memory unit. It is equal to 1/(cycle time) at RAM, while at non-RAM it is equal to $T_N = T_A + \frac{N}{R}$.

Where:

T_N: Average time to read or write N bits.

T_A: Average access time.

N: Number of bits.

R: Transfer rate, in bits per second (bps).

Computer memory is organized into a hierarchy. At the highest level (closest to the processor) are the processor registers then comes a cache memory next comes main memory Figure 2.2 shows memory hierarchy. There is a trade-off among the three key characteristics of memory: namely, cost, capacity and access time. At any given time, a variety of technologies are used to implement memory systems the following relationships hold:

- Faster access time, greater cost per bit.
- Greater capacity, smaller cost per bit.

- Greater capacity, slower access time.

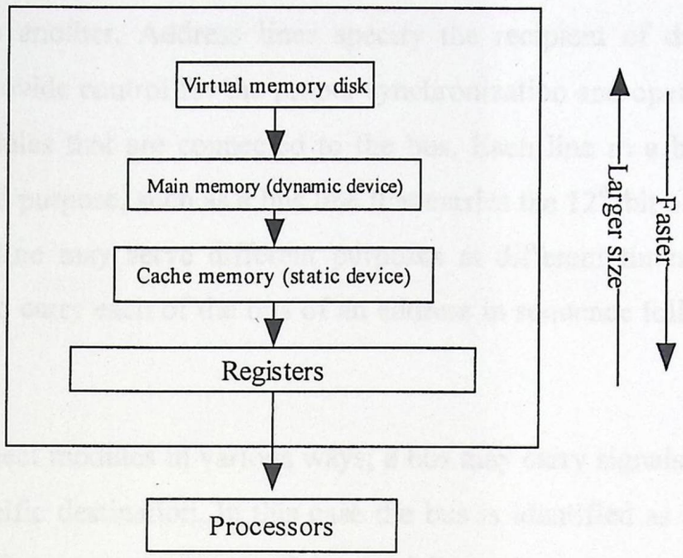

Figure 2.2: Memory hierarchies

2.2.3 Bus System

The processor must be able to communicate with all devices since its responsibility is to execute instructions related to these devices. Data (instructions and addresses) moves between the various I/O modules, memory, and the CPU through physical connection that makes it possible to transfer data from one location in the computer system to another called a bus.

A bus may be defined as a group of electrical conductors suitable for carrying computer signals from one location to another. The electrical conductors may be wires, or they may be conductors on a printed circuit. Each conductor in the bus is commonly known as a line. Lines on a bus are often assigned names, to make individual lines easier to identify. Busses are used most commonly for transferring data between computer peripherals and the CPU, for transferring data between the CPU and memory and for transferring data between different points within the CPU.

The lines on a bus can be grouped into as many as for general categories: data, addressing, control, and power. Data lines carry the data that is being moved from one location to another. Address lines specify the recipient of data on the bus. Control lines provide control for the proper synchronization and operation of the bus and of the modules that are connected to the bus. Each line in a bus may serve a single, dedicated purpose, such as a bus line that carries the 12^{th} bit of an address, for example, or a line may serve different purposes at different times. A single line might be used to carry each of the bits of an address in sequence followed by bits of data

Buses may connect modules in various ways; a bus may carry signals from a specific source to a specific destination. In this case the bus is identified as a point-to-point bus. The cable that connects the parallel or serial port in a personal computer from the computer to a printer is an example. Point-to-point buses intended for connection to a plug-in device are often called ports. Alternatively, a bus may be used to connect several points together. Such a bus is known as a multipoint bus or multidrop bus.

2.3 Software

Software is another term for computer program. Some important software that usually comes with your computer is called systems software or the operating system and in this section we discuss some of the important components of operating systems. There is a whole field of computer science concerned with the study of operating systems because of its fundamental importance in the use of computers.

2.3.1 Operating Systems

An operating system can be defines as a collection of computer programs that integrate the hardware resources of the computer and make those resources available to the user, in a way that allows the user access to the computer in a productive, timely, and efficient manner. It makes the resources available to the user and the user

programs in a convenient way, on the one hand, and controls and manages the hardware, on the other. In doing so it provides three basic types of services.

1. It accepts and executes commands and requests from the user and from the users programs.
2. It manages, loads, and executes programs.
3. It manages hardware resources of the computer.

The relationship between the various components of a computer system is shown schematically in Figure 2.4

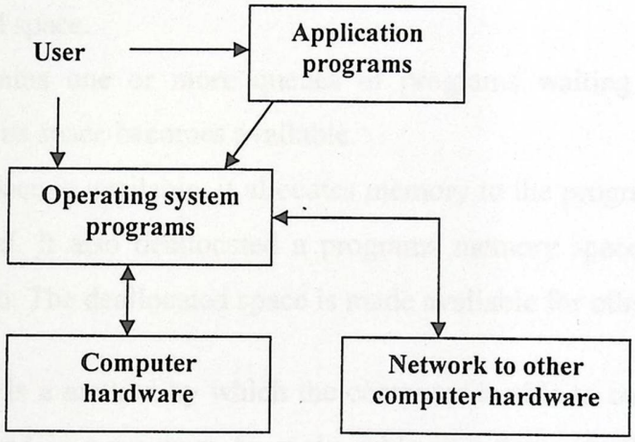

Figure 2.4: the modern integrated computer environment

The easiest way to think of an operating system is to consider it as a master program that accepts a request from a user and the users programs, and then calls its own programs to perform the required tasks. At the same time, it also calls programs to control and allocate the resources of a machine, including the use of the memory, the use of I/O devices, and a time available for various programs thus, if the user issues a command to load a program, a program loader is executed which then loads the desired program into memory and transfers control to the users program to run.

2.3.2 Memory Management

The purpose of the memory management system is to load programs into memory in such a way as to give each program loaded the memory that it requires for execution, The memory management system has three primary tasks. It attempts to perform these tasks in a way that is fair and efficient to the programs that must be loaded and executed.

1. It keeps track of memory-maintaining records that identify each program loaded into memory together with the space being used and also keeps track of available space. It allocates additional space for running program as required. It prevents programs from reading and writing memory outside their allocated space.
2. It maintains one or more queues of programs waiting to be loaded into memory as space becomes available.
3. When space is available, it allocates memory to the programs that are next to be loaded. It also deallocated a programs memory space when it complete execution. The deallocated space is made available for other programs.

Virtual memory is a method by which the computer is able to conceptually separate the addresses used in a program from the addresses that actually identify physical location in memory. The program addresses are referred to as logical addresses, since they represent locations in the program, but they do not have any reality outside the program it self. Logical addresses are also called virtual addresses; the words are used interchangeably. The actual memory addresses are called physical addresses. Virtual memory creates a correspondence between the logical and physical addresses so that each logical address is automatically and invisibly transformed into a physical address by the computer system during program execution. This transformation is known as mapping.

Virtual storage provide many important capabilities, including the ability to relocate programs from one part of memory to another, the ability to relocate programs easily

is fundamental to the concept of multitasking, it provide the same logical memory locations for two different programs by transforming them into different physical locations, or it can transform two logically independent programs and share the same physical memory locations for the program code for both, with independent physical memory data areas for each.

The advantage of virtual storage for memory management relies on the separation of logical and physical memory and the realization that logical memory and physical memory do not have to be of the same size. The size of the logical memory is established by the number of bits in the address space of an instruction word. The size of physical memory is theoretically determined by the size of the memory address register and the size of the word in the page table.

2.3.3 Scheduling

Program scheduling is not an issue with a single-tasking system; only one program is admitted to the system at a time, and when it is loaded the operating system simply transfers control to it for program execution. At completion of execution, the program transfers control back to the operating system. In a multitasking system the operating system is responsible for allocation of CPU time in a manner that is fair to the various programs competing for time.

There are two levels of scheduling. One level of scheduling determines which tasks will be admitted to the system and in what order (put in queue on some order of priority) and assigned memory space that will allow the program to be executed. This scheduling function known as high-level scheduling. The other level of scheduling known as dispatching. It is responsible for the actual selection of processes that will be executed at any given instant by the CPU.

2.4 CPU Structure and Functions

This section is devoted to discuss the internal structure and the function of the processor. The overall organization (ALU, control unit, register file) is reviewed.
The organization of the register file is also discussed. The functioning of the processor in executing machine instruction. The instruction cycle is examined to show the function and interrelationship of fetch, indirect, execute, and interrupt cycles.
The CPU consists of a control unit, registers, and the interconnections among these components. Figure 2.5 shows these basic components.

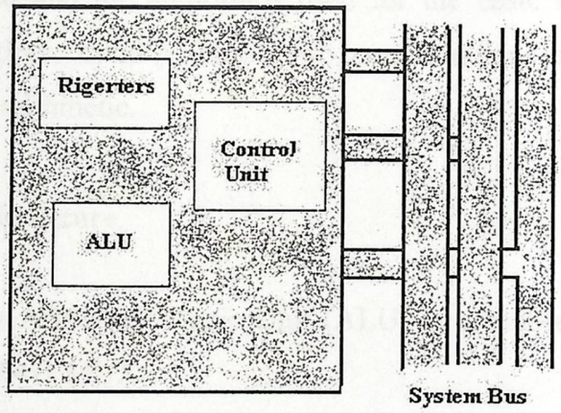

Figure2.5: CPU with system bus

2.4.1 The Arithmetic and Logic Unit

ALU is the smart part of a processor chip that actually performs arithmetic and logic operations on data. It also knows how to read logic commands like OR, AND, or NOT. Messages from the Control Unit instruct the ALU what to do and then it takes data from its close companion, the Registers, to perform the task. The reminder elements of the computer system (control unit, registers, memory, I/O) are there mainly to bring data into the ALU for it to process and then take the result backs out.

ALU like the rest components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. There is a strong relationship between the ALU and the processor components. Data are presented to ALU in registers and the results of the operations are stored on the registers. These registers are temporary storage locations within processor that are connected by signal paths to the ALU; ALU may also set flags as a result of an operation. The flag values are also stored in registers within the processor.

The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

The two principle concerns for computer arithmetic are the way in which numbers are represented and the algorithms used for the basic arithmetic operations (add, subtract, multiply, divide). These two considerations apply both to integer and floating-point arithmetic.

2.4.1.1 ALU Structure

A simple 1-bit Arithmetic-Logic Unit (ALU) is composed of AND gate, OR gate, Full adder, and 4-1 MUX.

The operation of the 1-bit ALU is represented as follow:

ADD $a + b + c_{in}$

AND a AND b

OR a OR b

In addition, ALU can subtract, by using the adder to add the negated form of the operand.

$$a - b = a + (-b)$$

ALU structure will contain an inverter to negate b; this gives the 1's complement.

To get the 2C value use $c_{in} = 1$ for least significant bit

$$a + \sim b + 1 = a + (\sim b + 1) = a + (\sim b) = a - b$$

Another MUX with control input b invert can select b or –b. This simple ALU can perform most of the data operations in the MIPS instruction set. Another operation, useful for branching; Set on Less Than (SLT) .Set the LSB to1, and 0 otherwise:
If (a-b) is negative, then a < b:

$$(a-b) < 0$$
$$(a-b) + b < 0 + b$$
$$a < b$$

The result is the same as sign bit from subtraction: by entering the sign bit from adder to LSB of output, here we can only do 1-ALU operation at a time (add or LST), there is a need extra ALU to the MSB and extra MUX.
The K-bit ALU, to operate on k-bit values, there are k 1-bit ALU's, so the 32-bit ALU is constructed using 32 1-bit ALU's as shown in Figure 3.2.

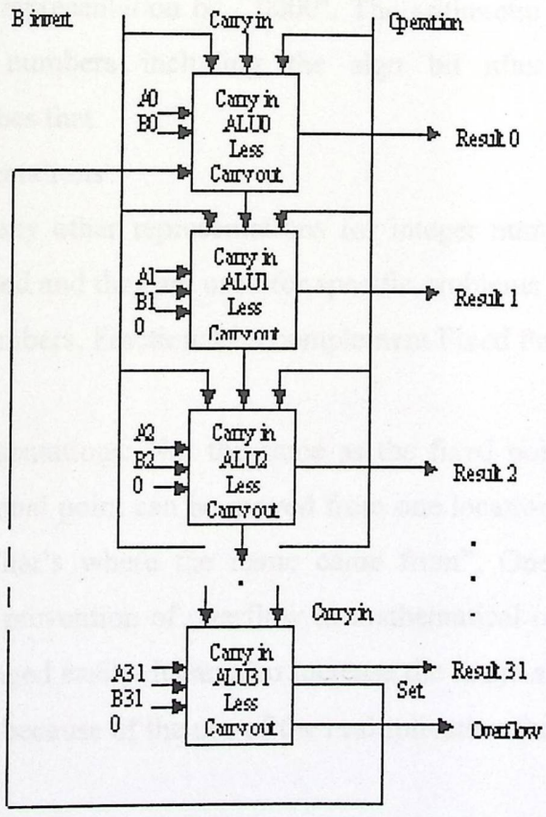

Figure2.6: 32-bit ALU

2.4.1.2 Number Representations

In computer system number can be represented in the different ways, the following subsections explain them.

Integer Representation: Integer or fixed-point numbers describe whole numbers. There are many ways to represent these numbers using the binary system .

- 1's complement signed representation

  It is the same concept as the signed-magnitude representation- the most left digit denotes the sign of the number -but the negative numbers are complemented by the 1's complement.

- 2's complement signed representation

  It is the same representation as the 1's complement but instead of calculating the 1's complement calculates 2's complement. In this representation the zero has a single representation by ``0000". The arithmetic subtraction is simple, adding the numbers including the sign bit after complementing the subtrahend does that.

- Other representations

  There are many other representations for integer numbers but they are not commonly used and they are used for specific problems .

  Fractional numbers, Fractional 2's complement Fixed Point representation

Floating Point representations: Not the same as the fixed point representation; the location of the fractional point can be moved from one location to another according to the precision. "That's where the name came from". One importance for this representation is the prevention of overflow in mathematical operations because the precision can be changed easily. It can also increase the range of the numbers that the computer can handle because of the use of the multiplication factor.

The floating-point representation is set of representations but all take the general format as "0. Fraction"

- Exponent-Fraction representation

  This representation is a direct mapping from the scientific representation that takes the form re*m. Where r is the radix, e is the exponent and m is mantissa.In computer representation r is 2 for binary representation and the location of the fraction point is assumed to be fixed so to change its location in the number the exponent must be changed.

- Biased Exponents

  In the biased exponent format, the exponent is not used, as a signed number - because the sign needs extra bit- it is biased by a base number so the exponent is subtracted from the base. This means when the exponent field has the value X the number has the exponent X - Base. This biasing makes all internal exponent calculations as positive only while keeping the whole range of the exponent intact .

2.4.2 Instruction Set

The operation of the CPU is determined by the instructions it executes, referred to as machine instructions or computer instructions.

An instruction set, or instruction set architecture (ISA), describes the aspects of a computer architecture visible to a programmer, including the native datatypes, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O (if any).

An ISA is a specification of the set of all binary codes (opcode) which are the native form of commands implemented by a particular CPU design. The set of opcodes for a particular ISA is also known as the machine language for the ISA.

An ISA can also be emulated in software by a interpreter. Due to the additional translation needed for the emulation, this is usually slower than directly running

programs on the hardware implementing that ISA. It is nowadays common practice for vendors of new ISAs or micro-architectures to make software emulators available to software developers before the hardware implementation is ready.

2.4.2.1 Machine Instruction Elements

The essential elements of a computer instruction are the opcode, which specifies the operation to be performed the source and destination operand references, which specify the input and output locations for the operation, and a next instruction reference, which is usually implicit.

Each instruction must contain the information required by the CPU execution

- Operation code (Opcode): specifies the operation to be performed. The operation is specified by a binary code.
- Source operand reference: the operation may involve one or more source operand; that is, operands that are inputs for the operation.
- Result operand reference: the operation may produce a result.
- Next instruction reference: this tells the CPU where to fetch the next instruction after the execution of this instruction is complete.
- Main or virtual memory: as with the next instruction reference the main or virtual memory address must be supplied. Virtual memory is a computer design feature that permits software to use more main memory (the memory which the CPU can read and write to directly) than the computer actually physically possesses.
- CPU registers: CPU contains one or more registers that may be referenced by machine instructions.
- I/O devices: the instruction must specify the I/O module and device for the operations.

Each instruction has at least two addressing modes, with most of them having four. The instruction set is orthogonal, i.e., each instruction implements every relevant addressing mode.

A system of codes directly understandable by a computer's CPU is termed this CPU's native or machine language. Although machine code may seem similar to assembly language they are in fact two different types of languages. Assembly code consists of both binary numbers and simple words whereas machine code is composed only of the two binary digits 0 and 1. Every CPU has its own machine language, although there is considerable overlap between some.

The "words" of a machine language are called instructions; each of these gives a basic command to the CPU. A program is just a long list of instructions that are executed by a CPU.

Instructions are simply a pattern of bits; different patterns correspond to different commands to the machine. The more readable rendition of a machine language is called assembly language.

Some languages give all their instructions the same number of bits, while the instruction length differs in others. How the patterns are organized depends largely on the specific language.

2.4.2.1.1 Instruction Format

An instruction formats defines the layout of the bits of an instruction, in term of its constituent parts. Each instruction has an opcode, and implicitly or explicitly, zero or more operands, each explicit operand is referenced using one of the addressing modes. The format most, implicitly or explicitly, indicate the addressing mode for each operand, for most instruction set more than one instruction format is used.

The design of an instruction format is a complex art; we examine the key design issues.

Instruction length: the most basic design issue to be faced is the instruction format length. This decision affects, and affected by, memory size, memory organization, bus structure, CPU complexity, and CPU speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer. The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space, here either the instruction length

should be equal to the memory transfer length (data bus length) or one should be a multiple of the other.

Allocation of bits: this issue determines how to allocate the bits in the format, there is a clearly trade-off between the number of opcode and the addressing capability.

To determine the addressing bits there are many factors: number of addressing modes, number of operands, register versus memory, number of register sets, address range and address granularity.

Variable length instructions: This format makes it easy to provide a large repertoire of opcode, with different opcode length. Addressing can be more flexible. The use of variable length instructions does not remove the desirability of making all of the instruction lengths integrally related to the word length because the CPU does not know the length of the next instruction to be fetched, atypical strategy is to fetch a number of bytes or words equal to at least the longest possible instruction. This means that some times multiple instructions are fetched.

2.4.2.1.2 Type of Operand

Machine instructions operate on data. The most important general categories of data are:

- Addresses:The range of addresses (memory,I/O) that can be referenced.
- Numbers: machine languages include numeric data types, numbers stored in a computer is limitted in magnitude .numerical data consists three types integer, floating-point, decimal.
- Characters: they cannot be easily stored or transmitted by data processing and communication systems, such systems are designed for binary data. Thus anumber of codes have been devised by which characters are represented by a sequence of bits. The most commonly used character code is the international reference alphabet (IRA) refereed to as the ASCII code.

- Logical Data: each word or other addressable unit is treated as a single unit of data. It is useful to consider an n-bit unit as consisting of n1- bit items of data

each item having the value zero or one. When data are viewed this way, they are considered to be logical data.

2.4.2.1.3 Type of Operation

The general types of operations are found on all machines, a useful and typical categorization is the following:

- Data transfer: This instruction must specify several things. first, the location of the source and destination operands must be specified. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands the mode of addressing for each operand must be specified.
  In terms of CPU action, data transfer operations are the simplest type if both source and destination are registers, the CPU causes data to be transferred from one registers to another. If one of both operands are in memory, then the CPU must perform the following actions: calculate the memory address based on the address mode, if the address refers to virtual memory, translate from virtual memory to actual memory address, determine whether the addressed item is in cache, if not issue a command to the memory module.

- Arithmetic: the basic arithmetic operations are add, subtract, multiply, and divide. These are provided for signed integer numbers; they are also provided for floating point and packed decimal numbers. There are other possible operation include a variety of single operand instructions (absolute, negate, increment, decrement).

- Logical: some of the basic logical operations that can be performed on Boolean or binary data are (NOT operation inverts a bit, AND, OR, and exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.).

- Conversion: those instructions that change the format or operate on the format of data, an example is converting from decimal to binary.

- System control: are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged. Typically, these instructions are reserved for the use of the operating system.

- Transfer of control: These instructions can change the sequence of instruction execution. Here the operation performed by the CPU is to update the program counter to contain the address of some instruction in memory. reasons for using transfer of control operations are: the ability to execute each instruction more than once and perhaps many thousands of times, you would like the computer to do one thing if one condition holds, and another thing if another condition holds, transfer to control instructions help if there are a mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

2.4.2.1.4 Addressing Modes

Form part of the instruction set architecture for some particular type of CPU. Some machine instructions will need to refer to (addresses of) operands in memory. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction

Four common addressing modes have been selected they are:

1. Direct. This is the same as absolute addressing. The address of the required data is part of the instruction. In this case, it will be the second byte of the instruction.
2. Indirect. The address containing the address of the required data is specified. There are normally two types of indirect modes:

a. Memory-indirect; and
b. Register-indirect.
    i. An instruction with memory-indirect addressing specifies the memory address in which the address of the required data is stored. A specified register contains the address of the data when register-indirect addressing is used. The indirect mode for this instruction set will be limited to register-indirect, and the register containing the address will always be the R Register.

3. Register. This is sometimes called inherent addressing. The required data is in a register.
4. Immediate. The required data is part of the instruction. For this architecture, it is the second byte of the instruction.

2.4.2.2 Instruction Set Characteristics

One of the requirements of a good instruction set is completeness. A complete instruction set can be used to evaluate any mathematical or logic function. The instruction set of an educational CPU should be complete, and should illustrate the types of instructions normally supported by actual CPUs.

The instruction set is both complete and reasonably simple. Writing a sequence of instructions to implement common operations that have not been included can informally prove its completeness. For example, multiplication can be performed, even though there is no multiply instruction, with a sequence of adds and shifts.

Another desirable characteristic of an instruction set is regularity. A regular instruction set includes normally expected operations. In this case, some commonly encountered instructions have been omitted.

Regularity also implies orthogonal. An orthogonal instructions set is one where each instruction can use every relevant addressing mode. This simplifies compiler design by making the rules for operand address specification consistent.

Additionally, there are several common instructions that cannot be fully implemented on the CPU: -

1- Conditional Jump: The only condition flag is zero. Conditional jumps can only be based on whether the zero flag is set. This illustrates the principle while maintaining a simpler hardware implementation.
2- Compare: Only equality can be evaluated.
3- Push & Pop: Eliminating the stack pointer simplified the hardware significantly, but precluded the use of these instructions.
4- Call & Return: Again, the lack of a stack pointer makes these difficult to implement. Jumping to predetermined addresses could perform the functions, but the programming would not be straightforward.
5- Loop: This operation must be implemented with a sequence of simpler instructions.
6- Increment: The additional circuitry required to implement this would not contribute to the goal of simplicity. If an increment is required, it can be done by the add instruction with immediate data of $(01)_H$.

2.4.2.2.1 Instruction Cycle

- Instruction Subcycle consists of Fetch (get next inst. from memory), Execute (interpret opcode & do it) & Interrupt (process if enabled.
- Additional Subcycle - the Indirect Cycle. Figure 3.3 shows standard Instruction Cycle.

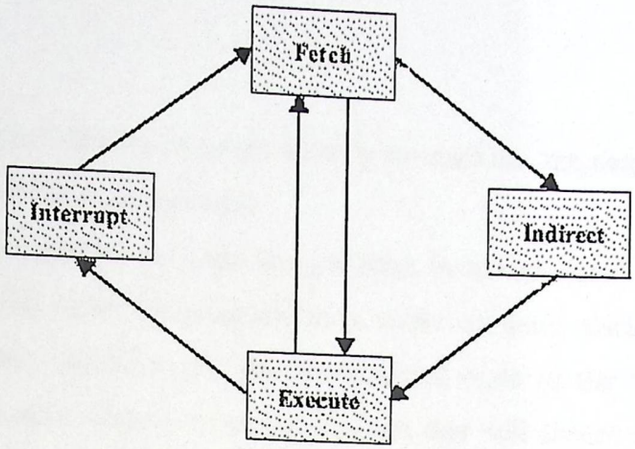

Figure2.7 instruction Cycle

- May well be that further indirect addressing is required after initial fetch .If so, operands are fetched - but obviously not those that are in the existing register set already...
- The actual data flow is a function of the CPU design - figure shows data flows for fetch, indirect & interrupt cycles...
- Fetch: instruction read data from memory PC given address of (next) instruction; this address moved to MAR and placed on the address bus; CU requests mem. Read, result placed on data bus, copied to MBR, and moved to the IR..., PC incremented by 1 (next instruction).
- After Fetch cycle, CU checks IR to see if an additional indirect addressing call is required for this operand. If so.
- Indirect: address field bits of the MBR transferred to the MAR, CU requests memory read, puts desired address into MBR... etc.
- Interrupt?: in this case, current contents of PC put in MBR for transfer to memory, MAR has the special location from the CU. Figure 3.4 shows full state diagram.

2.4.2.3 Assembly Language

A CPU can understands and execute machine instructions. These instructions are simply binary numbers stored in the computer, the programmer should enter the

program as a binary data to program directly in machine language. This is clearly a tedious and very error prone process.

There is an improvement to write the program in hexadecimal rather than binary notation, we could write the program as a series of lines. Each line contains the address of memory location and the hexadecimal code of the binary value to be stored in that location. Then we need a program that will accept this input, translate each line into binary number, and store it in the specified location. For more improvement we can make use of the symbolic name of each instruction. Each line of input represents one memory location. Each line consists of three fields separate by spaces. The first field contains the address of the location; the second field contains the three-letter symbol of the opcode if it is a memory-referencing instruction then a third field contains the address. According to previous implementation the program accept each line of input, generates a binary number based on the second and third field and stores it in the location specified by the first field.

A much better system is to use symbolic addresses. Each line still consist three fields, the first for address but a symbol is instead of an absolute numerical address, some lines does not have address implying that the address of that line is one more than the address of the previous line. The second field contains opcode symbol. For memory reference instruction the third field contains symbolic address.

With this last refinement we have an assembly language. Programs written in assembly language are translated into machine language by an assembler.

2.4.3 Register Organization

In computer architecture, a processor register is a small amount of very fast computer memory used to speed the execution of computer programs by providing quick access to commonly use values-typically, the values being in the midst of a calculation at a given point in time.

These registers are the top of the memory hierarchy, and are the fastest way for the system to manipulate data. Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". Registers are now usually implemented as an array of SRAMs, but they have also been implemented using individual flip-flop, high speed core memory, thin film memory, and other ways in various machines.

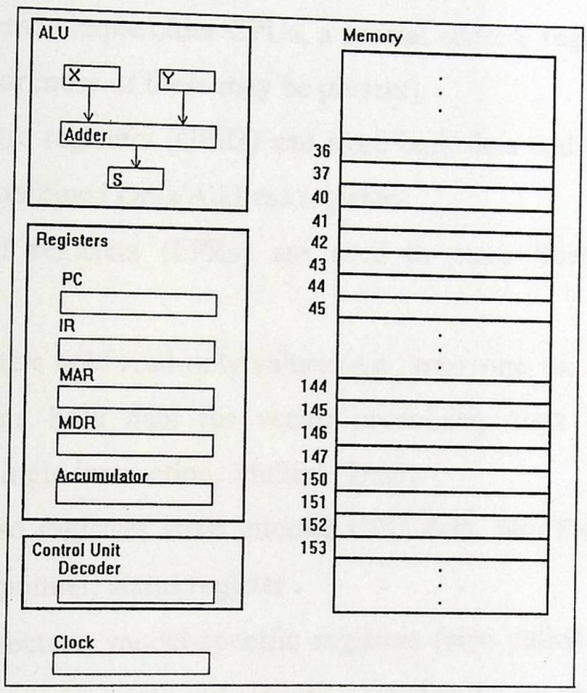

Figure2.8 CPU internal Register

The register in the CPU serve two functions:

- User-visible register: These enable the machine or assembly-language programmer to minimize main memory references by optimizing use of registers.
- Control and status registers: used by the control unit to control the operation of the CPU.

2.4.3.1User-visible Registers

The following paragraph mentions the register types:-

- Data registers are used to store integer numbers in some simple/older CPUs, a special data register is the accumulator, used for arithmetic calculations.
- Address registers hold memory addresses and are used to access memory. In some simple/older CPUs, a special address register is the index register or more of these may be present)
- General Purpose registers (GPRs) can store both data and addresses, i.e., they are combined Data/Address registers.
- Floating Point registers (FPRs) are used to store floating point numbers.
- Constant registers hold read-only values (e.g., zero, one, pi, ...).
- Vector registers hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- Special Purpose registers store internal CPU data, like the program counter ,stack pointer, status register .
- In some architecture, model-specific registers (also called machine-specific registers) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.

Two different approaches of modern physical register implementation have been pursued: The first is to have an electronic switch, such as a flip-flop, for each bit of the register. This system is stable, and is standard engineering practice. The other method is to map several states to the charge in one capacitor. This approach can sometimes be used to save money. Capacitors are the standard way to store data in electronic RAM.

2.4.3.2 Control and Status Register

There are a variety of CPU registers that are employed to control the operation of the CPU most of these are not visible to the user some of t5hem may be visible to machine instructions executed in a control or operating system mode. Different machines will have different register organizations and used different terminology.

Four registers are essential to instruction execution:

- Program counter (PC): contains the address of an instruction to be fetched and the PC is updated by the CPU after the instruction fetch so that it always points to the next instruction to be executed , a branch or skip instruction will also modify the contents of the PC.
- Instruction register (IR) :contains the instruction most recently fetched.
- Memory address register (MAR): contains the address of a location in memory, data are exchanged with memory using the MAR and MBR
- Memory Buffer register(MBR): contains a word of data to be written to memory or the word most recently read, MBR connects directly to the data bus. User-visible registers, in turn exchange data with the MBR.

All CPU designs include a register or set of registers, known as the program status word (PSW) that contains status information. And contains condition codes plus other status information. Common field or flags include the following:

- Sign: contains the sign bit of the result of the last arithmetic operation.
- Zero: set when the result is zero.
- Carry: set if an operation resulted in a carry (addition) into or borrow (subtraction) out of high- order bit. Used for multiword arithmetic operations.
- Equal: set if a logical compare result in equality.
- Overflow: used to indicate arithmetic overflow.
- Interrupt enable/disable: used to enable or disable interrupts

- Supervisor: indicate whether the CPU is executing in supervisor or user mode. Certain privileged instruction can be executed only in supervisor mode, and certain areas of memory can be access only on supervisor mode.

3

Control Unit Structure and Functions

Chapter 3

Control Unit Structure and Functions

Overview

The control unit is the brain of the microprocessor in that it contains the circuitry that coordinates the activities of all of the other circuitry in the computer. It is usually the most complex portion of an implementation, but not because it is especially sophisticated. Rather, the complexity stems from having to handle all of the special cases that naturally arise in any processor design.

3.1 Micro-Operations

If the CU is the brain, the clock is the heart of the system. It is a very simple circuit that sends out pulses at a regular interval. These pulses are like the ticking of an old-fashioned mechanical clock. They indicate the beginning and end of a basic interval of time in the processor. Every operation in the processor takes place during a clock period. Data moves from one place to another, a basic part of a computation takes place, a memory fetch is issued, all in response to a clock tick combined with signals from the CU.

On each clock tick, the CU transitions from one state to another. Usually there is a major state register that keeps track of what part of an instruction is being performed. By creating a very simple computer that takes the inputs as a jump address into a subroutine library. The simple computer is called a micro engine and the instructions it executes are called micro code. The micro code is stored in read only memory. Or by involves a read only memory in where the inputs are the address to the memory and the data stored in the location becomes the control signals but this memory is

Very large except for a really simple computer. Yet another approach is to construct a Boolean logic circuit that takes the inputs and generates the outputs. For obvious reasons, this is called a hard-wired CU.

Program executions consist of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (fetch, indirect, execute, and interrupt). The performance of each subcycle involves one or more shorter operations, which called micro-operations, that is the atomic operations of the processor.

3.2 Instruction Execution Cycle

At the grass roots of all modern processors is a process, which has basically remained, unchanged since the inception of computers that of the instruction execution cycle.

3.2.1 Fetch Cycle

To start off the fetch cycle, the address which is stored in the program counter (PC) is transferred to the memory address register (MAR). The CPU then transfers the instruction located at the address stored in the MAR to the memory buffer register (MBR) via the data lines connecting the CPU to memory. This transfer from memory to CPU is control unit (CU) task. To finish the cycle, the newly fetched instruction is transferred to the instruction register (IR) and unless told otherwise, the CU increments the PC to point to the next address location in memory. Figure 4.2 shows the fetch cycle.

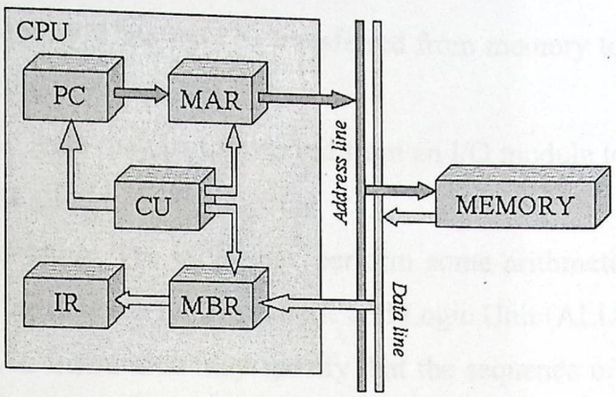

Figure3.1: Fetch cycle

The illustrated fetch cycle above (Figure 3.1) can be summarized by the following points:

1. PC => MAR
2. MAR => memory => MBR
3. MBR => IR
4. PC incremented

After the CPU has finished fetching an instruction, the CU checks the contents of the IR and determines which type of execution is to be carried out next. This process is known as the decoding phase. The instruction is now ready for the execution cycle.

3.2.2 Execute Cycle

Once an instruction has been loaded into the instruction register (IR), and the control unit (CU) has examined and decoded the fetched instruction and determined the required course of action to take, the execution cycle can commence. Unlike the fetch cycle and the interrupt cycle, both of which have a set instruction sequence, the execute cycle can comprise some complex operations (commonly called opcode).

The actions within the execution cycle can be categorized into the following four groups:

1. CPU - Memory: Data may be transferred from memory to the CPU or from the CPU to memory.
2. CPU - I/O: Data may be transferred from an I/O module to the CPU or from the CPU to an I/O module.
3. Data Processing: The CPU may perform some arithmetic or logic operation on data via the Arithmetic and Logic Unit (ALU).
4. Control: An instruction may specify that the sequence of operation may be altered. For example, the program counter (PC) may be updated with a new memory address to reflect that the next instruction fetched, should be read from this new location.

Figures 3.2 and 3.3 illustrate examples that will deal with two operations that can occur. The [LOAD ACC, memory] and [ADD ACC, memory], both of which could be classified as memory reference instructions. Instructions, which can be executed without leaving the CPU, are referred to as non-memory reference instructions.

LOAD ACC, memory

This operation loads the accumulator (ACC) with data that is stored in the memory location specified in the instruction. The operation starts off by transferring the address portion of the instruction from the IR to the memory address register (MAR). The CPU then transfers the instruction located at the address stored in the MAR to the memory buffer register (MBR) via the data lines connecting the CPU to memory. The CU coordinates this transfer from memory to CPU. To finish the cycle, the newly fetched data is transferred to the ACC.

Figure 3.2 illustrated LOAD operation that can be summarized in the following points:

1. IR [address portion] => MAR
2. MAR => memory => MBR
3. MBR => ACC

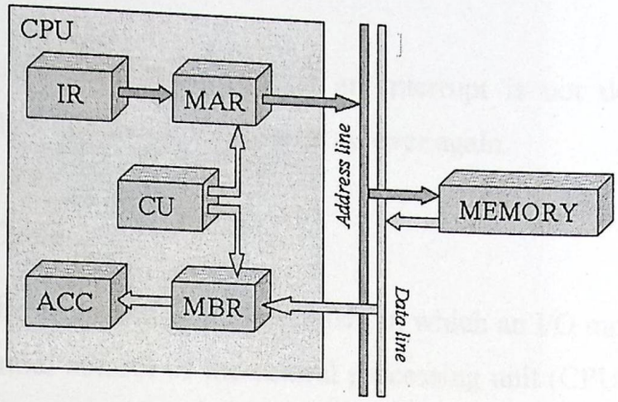

Figure3.2: Execute Cycle [LOAD ACC, memory] operation

While Figure 3.3 illustrates the ADD ACC, memory; this operation adds the data stored in the ACC with data that is stored in the memory location specified in the instruction using the ALU. The operation starts off by transferring the address portion of the instruction from the IR to the MAR. The CPU then transfers the instruction located at the address stored in the MAR to the MBR via the data lines connecting the CPU to memory. This transfer from memory to CPU is coordinated by the CU. Next, the ALU adds the data stored in the ACC and the MBR. To finish the cycle, the result of the addition operation is stored in the ACC for future use.

The illustrated ADD operation Figure 3.3 can be summarized in the following points:

1. IR [address portion] => MAR
2. MAR => memory => MBR
3. MBR + ACC => ALU
4. ALU => ACC

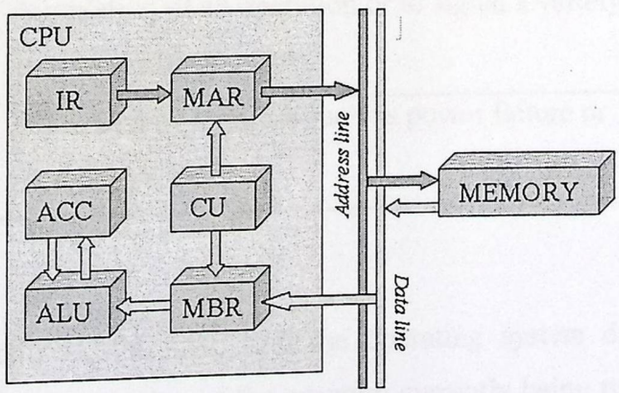

Figure3. 3: Execute Cycle [ADD ACC, memory] operation

After the execution cycle completes, if an interrupt is not detected, the next instruction is fetched and the process starts all over again.

3.2.3 Interrupt Cycle

An interrupt can be described as a mechanism in which an I/O module etc. can break the normal sequential control of the central processing unit (CPU). Table 3.1 below, summarizes the most common form of interrupts that the CPU can receive.
The main advantage of using interrupts is that the processor can be engaged in executing other instructions while the I/O modules connected to the computer are engaged in other operations.

Table 3.1: Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute am illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.

When interrupts are introduced, the CPU and the operating system driving the system, is responsible for the suspension of the program currently being run, as well as restoring that program at the same point before the interrupt was detected. To

handle this, an interrupt handler routine is executed. This interrupt handler is usually built into the operating system. Before the interrupt handler routine can ran, several processes must occur first. A typical sequence of events is illustrated in Figure 3.4 below. After the completion of the interrupt handler routine, the normal sequential fetch / execute cycle begins.

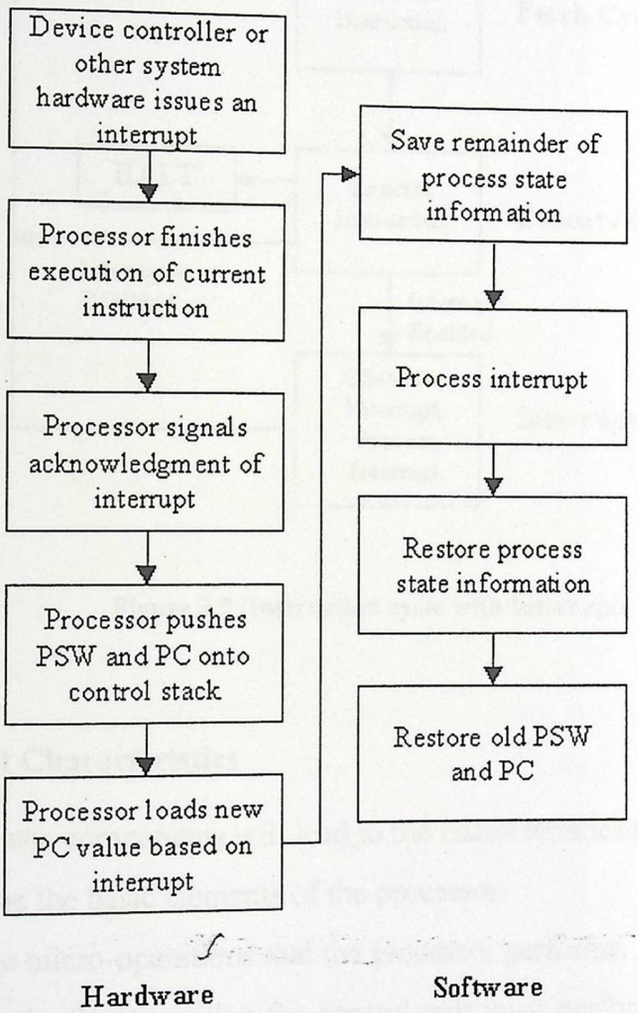

Figure3.4: Sample interrupts processing

Figure 3.5 illustrates how the interrupt cycle fits into the overall cycle.

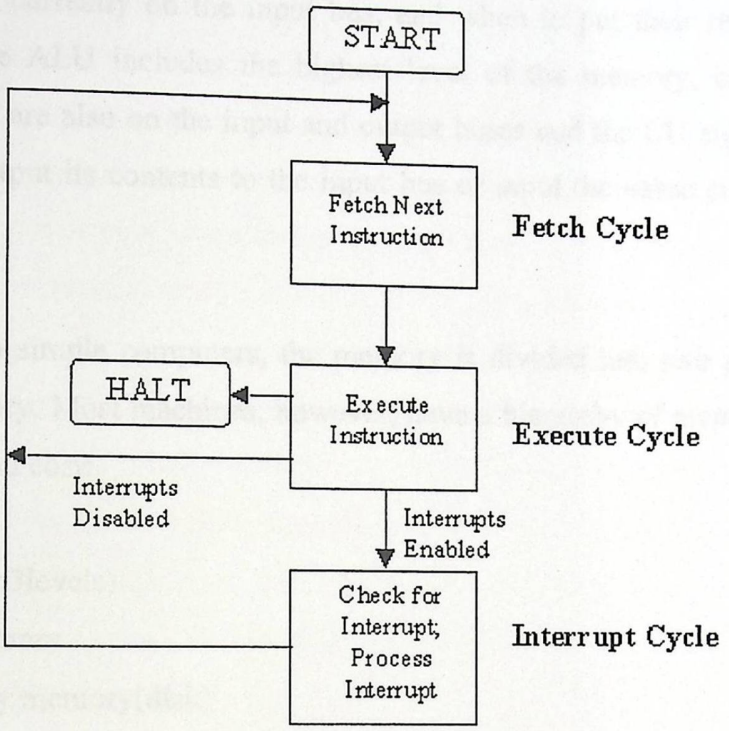

Figure 3.5: Instruction cycle with Interrupts

3.3Control Unit Characteristics

All computers basic components will lead to the characteristics of the control unit.

- Determine the basic elements of the processor.
- Define he micro-operations that the processor performs.
- Describe the functions that the control unit must performto cause the micro-operation to be performed.

The basic elements of the processor have been mentioned in the previous chapters in details.

ALU, this part of the system performs the actual computations: arithmetic, logical operations, comparisons, data movement and reformatting. It usually consists of a

collection of modular circuits that are connected to share sets of input and output wires called buses. The control unit signals the different circuits when they are to accept the data currently on the input bus, and when to put their results onto the output bus. The ALU includes the highest level of the memory, called the data registers. These are also on the input and output buses and the CU signals which of them should output its contents to the input bus or input the value currently on the output bus.

The Memory in simple computers, the memory is divided into two parts: registers and main memory. Most machines, however, have a hierarchy of memory types that vary in speed and cost:

- Registers
- Cache(1to3levels)
- Main memory
- Secondary memory(disk)

We have more of the cheaper forms of memory in a system. This recognizes that the processor is only accessing a small portion of all of the available data at any one time. So rather than pay for all of the memory to be fast and expensive, a small fast memory is used to hold the current "working set" and cheaper, slower memory can hold everything else. The registers and one or two levels of cache typically appear on the same chip in today's microprocessors.

Input/Output, There are many forms of I/O. In early computers, the ALU had I/O registers and the data would be input and output under direct program control. Modern systems mostly use direct memory access (DMA) in which a separate I/O processor has access to the main memory of the system and either a set of parameters or a simple program is stored by the CPU in registers in the I/O processor which is then released to perform the I/O operation independently of the subsequent actions of the CPU.

The control unit does not input, output, process, or store data; rather, it initiates and controls the sequence of these operations. In addition, the control unit communicates with input devices to begin the transfer of data or instructions into storage, with the ALU to initiate arithmetic or logic operations, and with output devices to begin the transfer of results from storage. Data transfer involves the moving of data or instructions from one location in a computer to another. It is noteworthy that when an item of data is stored in a given location, it replaces the previous contents of that location, but when an item of data is moved form one location in storage to another, the item of data is not physically removed from its initial storage location; what happen is that the data is copied to the new location. When the computer is executing a program contained in primary storage, the control unit obtains the instructions in the sequence in which they will be executed, interprets those instructions, and issues signals or commands that cause other units of the system to execute them. To accomplish this, the control unit must communicate with both the arithmetic/logic unit and primary storage.

In executing an instruction, the control unit generally performs all or most of the following functions:

- Determines the instruction to be executed
- Determines the operation to be performed by the instruction
- Determines what data, if any, are needed and where they are stored
- Determines where any results are to be stored
- Determines where the next instruction is located
- Causes the instruction to be carried out or executed
- Transfers control to the next instruction

The activities of the control unit, as all other activities in a computer system, are actually composed of thousands of individual steps, each of which takes place in a fixed interval of time. These intervals are controlled by an internal electronic clock that emits up to a billion regular electronic pulses every second. All operations within the CPU of the computer take place within a fixed number of clock pulses. This fixed number of pulses determines the machine cycle for the computer. Within a

machine cycle, the computer can perform one machine operation. The number of machine operations required to execute a single instruction will vary from instruction to instruction. Execution of such instructions takes place under the direct supervision of the control unit.

Registers, as described earlier in this research, are temporary high-speed storage locations. They are generally contained within the CPU or MPU, and used by both the control and the arithmetic/logic units to perform their functions. As mentioned in previous chapters there are special purpose registers used by the computer for only one purpose, and general-purpose registers, used by the programmer or computer for different purposes. Special-purpose registers include an accumulator that accumulates totals, a storage register that contains information taken from or going to primary storage, and an instruction register that contains the instruction or command being executed. Registers can vary in both size and capacity from computer to computer or even within the same computer system.

The processor is logically and electronically connected to primary storage by a system of wires called a bus that links these internal components and is capable of transmitting electrical impulses. Most microcomputers use one or more internal or local buses for communicating within the CPU and a common or system bus for communicating with components outside the CPU.

Counters are temporary storage areas that operate much like registers and may perform similar functions. Counter may be used to count up or down. A typical use of a counter is to maintain the storage address of the next instruction to be executed.

The following points can summarize control unit functions; control unit performs the major two tasks:

- Sequencing: The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
- Execution: control unit causes each micro-operations to be performed.

3.4 Control Unit Control Signals

Control unit to perform its function, it must have inputs that allow it to determine the state of the system and the outputs that allow it to control the behavior of the system externally, but control unit internally must have a logic required to perform its basic tasks; sequencing and executing functions. Figure 3.6 shows the general model for the control unit, showing all of its inputs and outputs. The inputs are as follows:

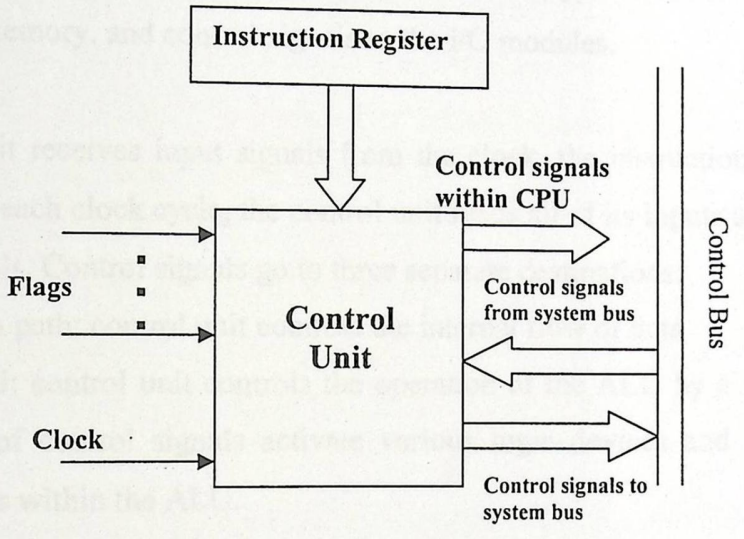

Figure 3.6: General model of the control unit

1. Clock: also known as the clock cycle time; control unit causes one micro-operations or a set of simultaneous micro-operations to be performed for each clock pulse.
2. Instruction register: the opcode of the current instruction is used to determine which micro-operations to be performed during the execute cycle.
3. Flags: these are needed by the control unit to determine the status of the processor and the outcome of the processor and the outcome of the previous ALU operations.

4. Control signals from control bus: interrupt signals and acknowledgements are provided by the control bus which is part of the system bus to the control unit.

Output signals are the following:

1. Control signals within the processor: these are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
2. Control signals to control bus: these are also two types: control signals to the memory, and control signals to the I/O modules.

The control unit receives input signals from the clock, the instruction register, and the flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

1. Data path: control unit controls the internal flow of data
2. ALU: control unit controls the operation of the ALU by a set of control signals activate various logic devices and gates within the ALU.
3. System bus: control unit sends control signals out onto the control lines of the system bus.

Control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading tall of its inputs, the control unit emits a sequence of control signals that causes micro-operation to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize.

3.5 Control Unit Implementation

CPU - whether simple or complex - basically consists of a control unit, plus a data part, encompassing:

1. A set of registers (including registers that interface to the system bus).
2. A set of D-units (adders, shifters etc.)
3. A set of data paths (busses) connecting the above.

There are number of techniques used to implement the CPU control unit and most of them either a hardwired implementation or a microprogrammed implementation.
The data part is capable of performing a set of micro-operations or primitive computations that can be performed in one minor cycle (clock pulse). Each micro-operation changes the contents of a single register. An instruction in the user-visible instruction set must be programmed as a series of micro-operations (some of which may be done in parallel on the same clock pulse.) The bus system is also capable of various micro-operations - e.g. performing a read cycle, write cycle etc.

Control of the system is accomplished by a control unit that -at the start of each minor cycle - activates the necessary control functions to cause the data part to perform the desired micro-operation(s) on the next clock pulse. The set of control signals that pass from Control to the data part and bus system is called a micro-word or control word. Conceptually, each bit of this micro-word corresponds to the enabling of one particular micro-operation that some system component can perform. On a sophisticated machine, a microword could well comprise hundreds of bits - most of which will be zero.
The description above is somewhat simplified from reality. On many machines, advantage is taken of the fact that certain micro-operations are mutually exclusive - e.g. one does not simultaneously gate two different registers onto the same bus or load the same register from two different sources. Thus, what passes from the control unit to the data part is often an encoding - e.g. 4 bits in the control word may be used

to select one of 16 registers is gated onto a specific bus. In this case the bus will contain a MUX to decode the 4 bits and select the correct register. Note, however, that the ultimate micro-word does in fact contain the full 16 bits - what has been done, in effect, is to place part of the control unit (the MUX) in the data part, so that the complete micro-word only exists internally there.

The job of the control unit designer is to develop a means whereby an orderly sequence of control words may be presented to the data part (and other hardware such as the memory) - one per clock pulse.

For example, consider a very simple CPU that deals only with one size of binary integer operands, has a single accumulator and uses one address instructions, each one word long, with a format like the following:

Op-code operand-address

For such a machine, performing an instruction involves some combination of instruction fetch, instruction decode, operand address calculation, operand fetch, execution, and operand store phases.

- The instruction fetch phase of each instruction on this machine might involve the following sequence of micro-operations - where each line represents the operations to be performed on one clock pulse:

  MAR <- PC

  MBR <- M [MAR], PC <- PC + 1 (This is common to all instructions)

- The instruction decode phase of each instruction on this machine might involve the following micro-operation:

  IR <- MBR (op-code part)

  (This is common to all instructions. From here on, the contents of the IR will now determine what micro-operations are done next.)

- For an ADD instruction, the remaining phases might look like this.

  MAR <- MBR (address part) (Operand address calculation)

  MBR <- M [MAR] (Operand fetch)

  AC <- AC + MBR (Execute)

  (No operand store)

- For an unconditional branch instruction, these phases might look like this:

MAR <- MBR (address part)	(Operand address calculation)
	(No operand fetch)
PC <- MAR	(Execute)
	(No operand store)

- For a store accumulator instruction, these phases might look like this:

MAR <- MBR (address part)	(Operand address calculation)
	(No operand fetch)
MBR <- AC	(Execute)
M [MAR] <- MBR	(Operand store)

- For a shift accumulator left one place instruction, we need only an execute phase:

	(No operand address calculation)
	(No operand fetch)
AC <- shl AC	(Execute)
	(No operand store)

- If indirect addressing is used, then we must add to the operand address calculation in each case:

MBR <- M [MAR]

MAR <- MBR

3.5.1 Hardwired Control

The control unit is implemented as a state machine, with combinatorial circuits generating each of the control functions on the basis of the current state and certain variables such as the op-code of the user instruction undergoing execution.

Figure 3.7 is a block diagram showing the internal organization of a hardwired control unit for our simple computer. Input to the controller consists of the 4-bit opcode of the instruction currently contained in the Instruction Register and the

negative flag from the accumulator. The controller's output is a set of 16 control signals that go out to the various registers and to the memory of the computer, in addition to a HLT signal that is activated whenever the leading bit of the op-code is one. The controller is composed of the following functional units: A ring counter, an instruction decoder, and a control matrix.

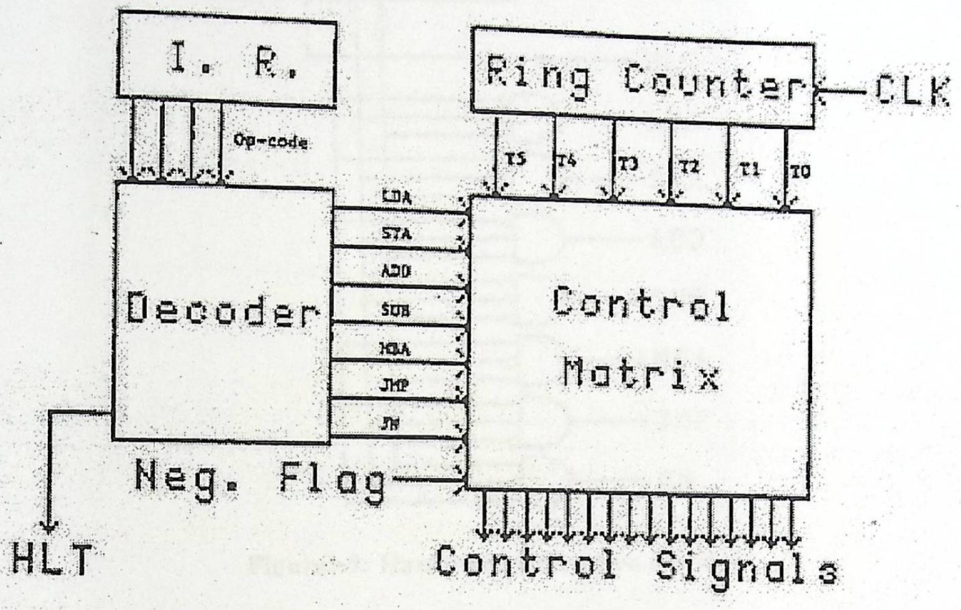

Figure3.7:General block diagram for the basic computer's hardwired control unit

The ring counter provides a sequence of six consecutive active signals that cycle continuously. Synchronized by the system clock, the ring counter first activates its T0 line, then its T1 line, and so forth. After T5 is active, the sequence begins again with T0. Figure3.8 shows how the ring counter might be organized internally.

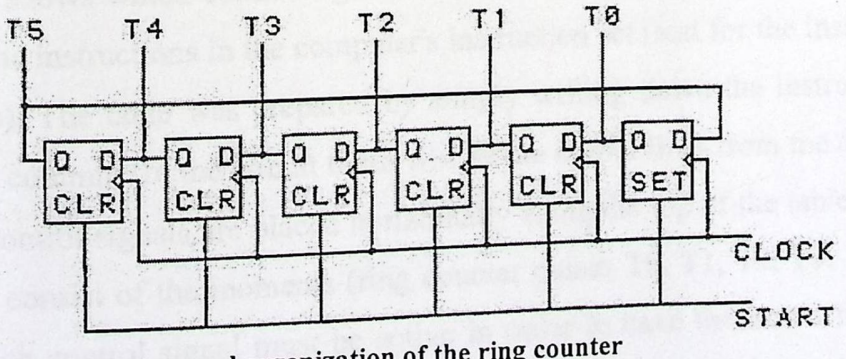

Figure3.8:The internal organization of the ring counter

The instruction decoder takes its four-bit input from the op-code field of the instruction register and activates one and only one of its 8 output lines. Each line corresponds to one of the instructions in the computer's instruction set. Figure3.9 shows the internal organization of this decoder.

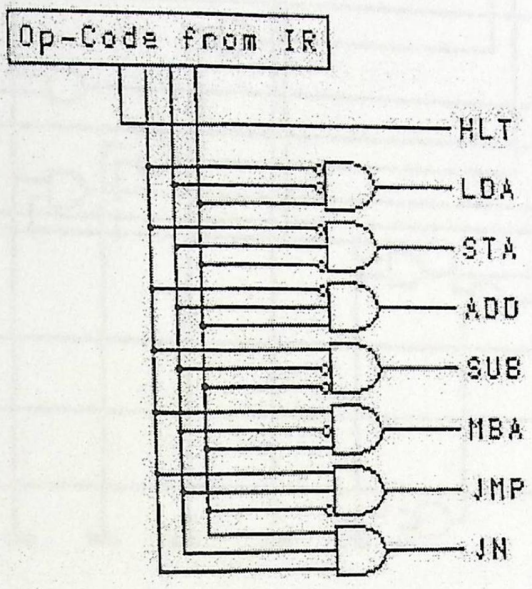

Figure3.9: Hardwired instruction decoder

The most important part of the hard-wired controller is the control matrix. It receives input from the ring counter and the instruction decoder and provides the proper sequence of control signals. Figure3.10 is a diagram of how the control matrix for our simple machine might be wired. To understand how this diagram was obtained, we must look carefully at the machine's instruction set (Table 3.1).

Table3.2 shows which control signals must be active at each ring counter pulse for each of the instructions in the computer's instruction set (and for the instruction fetch operation). The table was prepared by simply writing down the instructions in the left-hand column. (In the circuit these will be the output lines from the decoder). The various control signals are placed horizontally along the top of the table. Entries into the table consist of the moments (ring counter pulses T0, T1, T2, T3, T4, or T5) at which each control signal must be active in order to have the instruction executed. This table is prepared very easily by reading off the information for each instruction

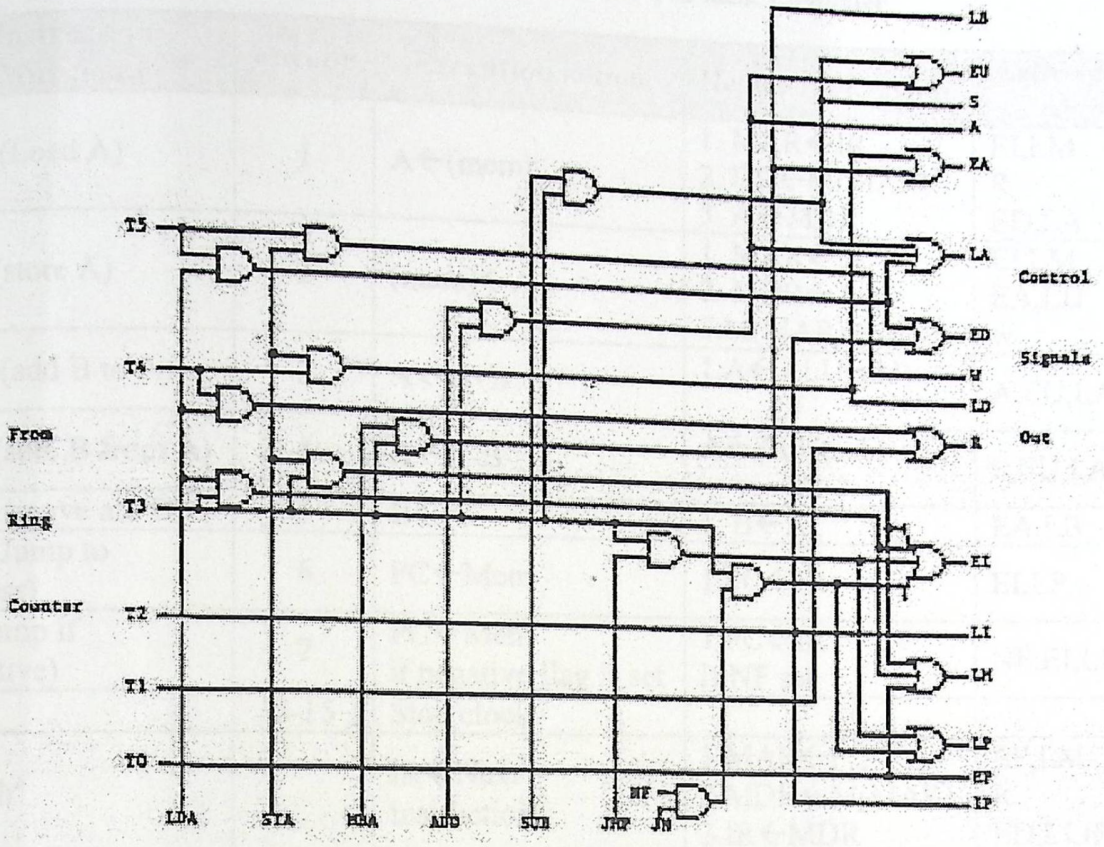

Figure3.10: the hardwired control matrix

given in Table 3.1 For example, the Fetch operation has the EP and LM control signals active at ring count 1, and ED, LI, and IPC active at ring count 2. Therefore the first row (Fetch) of Table 2 has T0 entered below EP and LM, T1 below R, and T2 below IP, ED, and LI.

Once Table 3.2 has been prepared, the logic required for each control signal is easily obtained. For each an AND operation is performed between any active ring counter (Ti) signals that were entered into the signal's column and the corresponding instruction contained in the far left-hand column. If a column has more than one entry, the output of the ANDs are ORs together to produce the final control signal. For example, the LM column has the following entries: T0 (Fetch), T3 associated with the LDA instruction, and T3 associated with the STA instruction. Therefore, the

Table3.1 An Instruction Set For The Basic Computer

Instruction Mnemonic	opcode	Execution action	Register Transfers	Active control signal
LDA(Load A)	1	A←(mem)	1. MAR←IR 2. DR←M(MAR) 3. A←MDR	EI,LM R ED,LA
STA(store A)	2	(mem)←A	1. MAR←IR 2. MDR←A 3.M(MAR)←MDR	EI,LM EA,LD w
ADD(add B to A)	3	A←A+B	1.A←ALU(add)	A,EU,LA
SUB(sub. B from A)	4	A←A-B	A←ALU(sub)	S,EU,LA
MBA(move ato B)	5	B←A	1. B←A	EA,LB
JMP(Jump to address)	6	PC←Mem	1.PC←IR	EI,LP
JN(Jump if Negative)	7	PC←Mem if negative flag is set	1.PC←IR If NF set	NF:EI,LP
HLT	8-15	Stop clock		
"Fetch"		IR←Next Instruction	1.MAR←PC 2.MDR←M(MAR) 3.IR←MDR	EP,LM R ED,LI,IP

logic for this signal is:

LM = T0 + T3*LDA + T3*STA

This means that control signal LM will be activated whenever any of the following conditions is satisfied: (1) ring pulse T0 (first step of an instruction fetch) is active, or (2) an LDA instruction is in the IR and the ring counter is issuing pulse 3, or (3) and STA instruction is in the IR and the ring counter is issuing pulse 3.

The entries in the JN (Jump Negative) row of this table require some further explanation. The LP and EI signals are active during T3 for this instruction if and only if the accumulator's negative flag has been set. Therefore the entries that appear above these signals for the JN instruction are T3*NF, meaning that the state of the negative flag must be ANDed in for the LP and EI control signals.

Figure 3.11 gives the logical equations required for each of the control signals used on our machine. These equations have been read from Table 3.2, as explained above. The circuit diagram of the control matrix Figure3.10 is constructed directly from these equations.

IP = T2
W = T5*STA
LP = T3*JMP + T3*NF*JN
LD = T4*STA
LA = T5*LDA + T3*ADD + T3*SUB
EA = T4*STA + T3*MBA
EP = T0
S = T3*SUB
A = T3*ADD
LI = T2
LM = T0 + T3*LDA + T3*STA
ED = T2 + T5*LDA
R = T1 + T4*LDA
EU = T3*ADD+T3*SUB
EI = T3*LDA + T3*STA + T3*JMP + T3*NF*JN
LB = T3*MBA

Figure 3.11: The logical equations required for each of the hardwired control signals on the basic computer.

It should be noticed that the HLT line from the instruction decoder does not enter the control matrix, Instead this signal goes directly to circuitry (not shown) that will stop the clock and thus terminate execution.

Table3. 2:A Matrix of Times at which Each Control Signal Must Be Active in Order to Execute the Hard-wired Basic Computer's Instructions

Control Signal	IP	LP	EP	LM	R	W	LD	ED	LI	EI	LA	EA	A	S	EU	LB
Instruction																
"Fetch"	T2		T0	T0	T1			T2	T2							
LDA				T3	T4			T5		T3	T5					
STA				T3		T5	T4			T3		T4				
MBA												T3				T3
ADD											T3		T3		T3	
SUB											T3			T3	T3	
JMP		T3								T3						
JN		T3*NF								T3*NF						

3.5.2 Microprogrammed Control

Control word (CW) is a sequence of n-bits, each bit in a CW corresponds to one control signal. Each control step during execution of an instruction defines a certain CW; it represents a combination of 1s and 0s corresponding to the active and nonactive control signals. Microroutine is a sequence of CWs corresponding to the control sequence of a machine instruction. An individual CW in a microroutine is called a microinstruction.

The basic idea in Microprogrammed control is that all microroutines corresponding to the machine instructions are stored in the control store. And the control unit generates the sequence of control signals for a certain machine instruction by reading from the control store the CWs of the microroutine corresponding to the respective instruction. The control unit is implemented just like another very simple CPU, inside the CPU, executing microroutines stored in the control store. Figure3.12 illustrates the microprogrammed control.

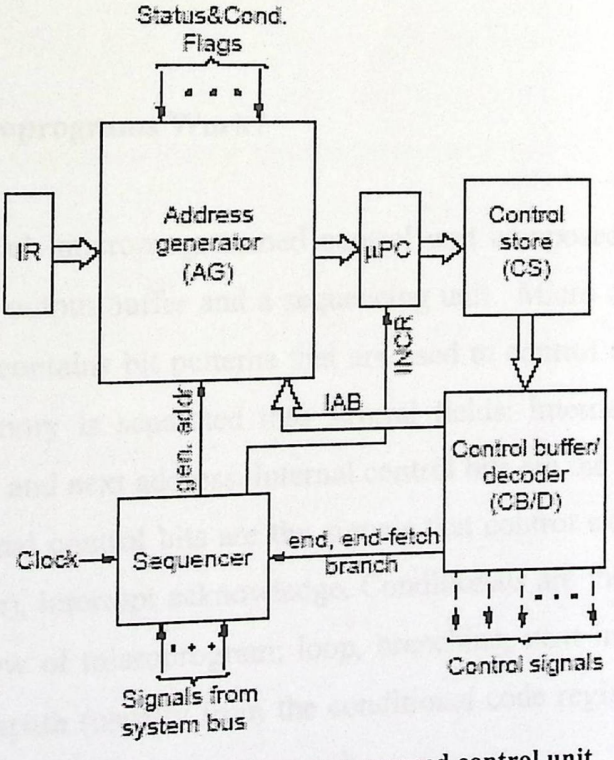

Figure3.12:Microprogrammed control unit

The various control words needed to implement the user instructions are stored in a ROM, with a sequencer causing the appropriate control word to be fetched at each clock cycle and fed to the rest of the CPU. Microprogram that made a control unit easier to design and is more flexible, is that a control unit can be implemented as a memory which contains patterns of the control bits and part of the flow control for sequencing those patterns. Microprogram control unit is actually like a miniature computer which can be "programmed" to sequence the patterns of control bits. Its "program" is called "microprogram" to distinguish it from an ordinary computer program. Using microprogram, a control unit can be implemented for a complex instruction set which is impossible to do by hardwired.

Microprogram approach for control unit has several advantages:

- One computer model can be microprogrammed to "emulate" other model.
- One instruction set can be used throughout different models of hardware.
- One hardware can realized many instruction sets. Therefore it is possible to choose the set that is most suitable for an application.

3.5.2.1 How Microprograms Work?

Like the RAM model, microprogrammed control unit composed of microprogram PC, micro memory, output buffer and a sequencing unit. Micro memory (sometime called micro store) contains bit patterns that are used to control the datapath. Each word of micro memory is separated into several fields: internal control, external control, conditional, and next address. Internal control bits are the signals that control the datapath. External control bits are the signals that control external unit such as memory (read, write), interrupt acknowledge. Conditionals are the bits that are used to determine the flow of microprogram; loop, branching, next instruction. Its input comes from the datapath (usually from the conditional code register). Next address determines the next microword to be executed.

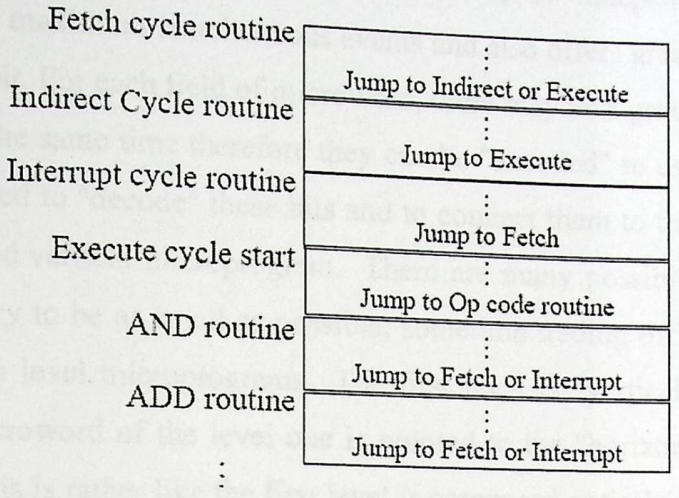

Figure 3.13:Control store/memory

A Microprogram is executed as follow:

- A word from microprogram at the location specified by the microPC is read out; control bits are latched at the output buffer which is connected to the datapath.
- If conditional field is specified and the test for conditional is true, the next address of microprogram will come from the next address field otherwise the microPC will be incremented (execute the next microword).

What that has been described is called horizontal microprogram. The microword can have other formats. In horizontal microprogrammed wide CW reserve one bit of the CW for each control signal many micro-operations can be executed in parallel, but a large space is used. There are several possibilities:

1. Single format, one address as just described above.
2. Single format, two addresses, contain two next addresses field, one for result of test true, and the other for result of test false.
3. Multiple formats, such as, one format for the control bits without the next address field and another format for "jump on condition" with the address field. The advantage is that the microword can be shorter than the single format. The disadvantage is that to "jump" will take one extra clock.

Horizontal microprogram allows each control bit to be independent from other therefore enables maximum simultaneous events and also offers great flexibility. It is also waste a lot bit. For each field of microword, there may be a group of bits that are not activated at the same time therefore they can be "encoded" to use a fewer bit. A decoder is required to "decode" these bits and to connect them to the datapath. This approach is called vertical microprogram. There are many possibilities to compact the micro memory to be as small as possible, sometime trading off speed for space, for example, two level microprograms. The first level is "vertical" i.e. maximally encoded; the microword of the level one is pointed to the "horizontal word" of the second level. This is rather like the first level is composed entirely from "subroutine call" and the second level is the subroutine.

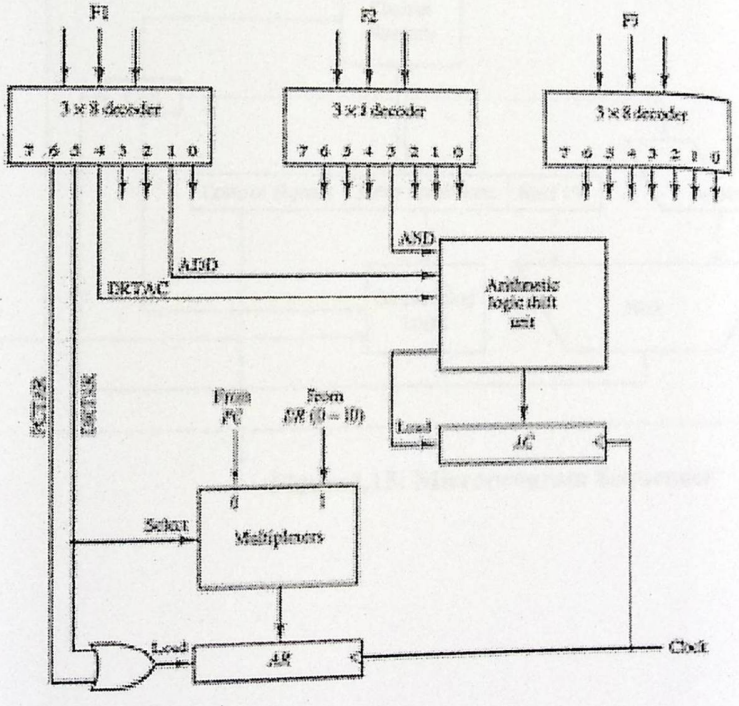

Figure3.14: Decoding micro-operation

Microprogram becomes obsolete mainly because the present design emphasizes the performance and microprogram is slower than hardwired. The change in instruction set design toward a minimum number of clock per instruction simplifies the instruction set to the point that microprogram is not really required. Also the design of hardwired control unit can be mostly automated as opposed to microprogram

which must be "written" and "debug". Hence, for the current instruction set architecture, hardwired control unit offers a lower engineering cost.

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. Figure3.15 illustrates the general sequence techniques to get the next microinstruction from the control memory.

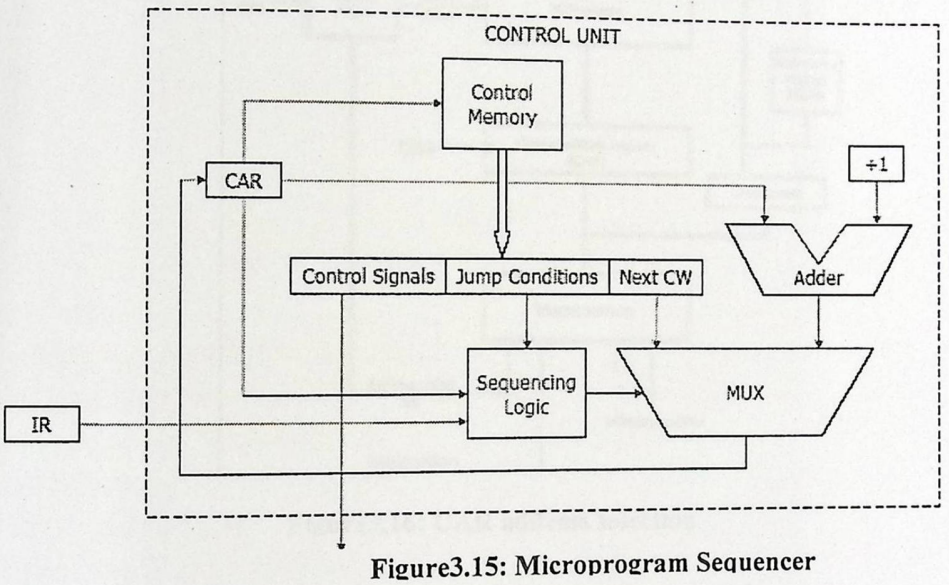

Figure3.15: Microprogram Sequencer

The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction. Figure 3.16 illustrate the address selection

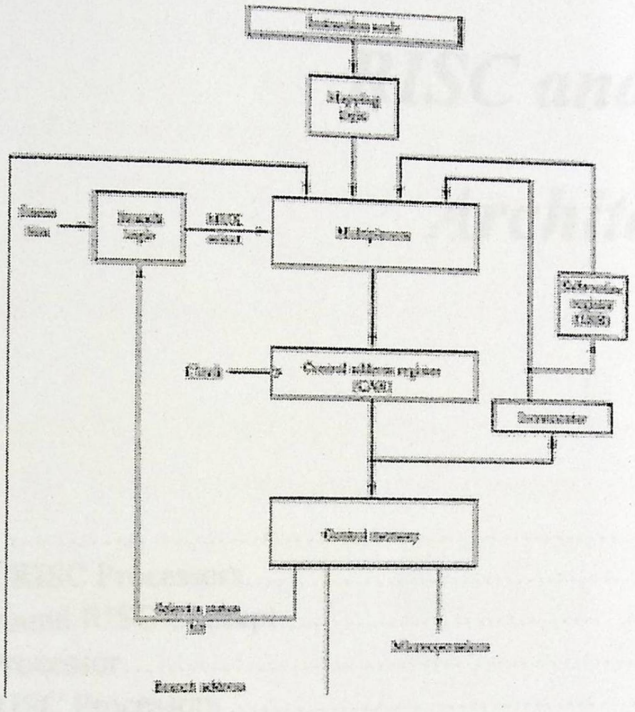

Figure3.16: CAR address selection

4

RISC and CISC Architectures

Chapter 4

RISC and CISC Architectures

4.1 Introduction

There seems to be now an overwhelming case in favor of Reduced Instruction Set Computers (RISC) as high performance computing engines. RISC processors, first developed in the eighties, seem predestined to dominate the computer industry in the nineties and to relegate old microprocessor architectures into oblivion.

But what does RISC mean? What are the essential features of this new approach to computer architecture? There is a widespread misunderstanding of what RISC really means and of the way in which the new processors are capable of reaching performance levels reserved before for much larger systems. "RISC" is generally interpreted as meaning that a processor should implement only a small instruction set capable of running faster than in traditional designs.

What is meant when we speak of RISC systems? Understanding the basic tenets of the RISC design philosophy makes it possible to find out where the performance advantage of the new processors comes from and, more important, what type of new features could be expected in the future.

4.2 Characteristics of RISC Processors

The most relevant characteristics that should be taken into account when discussing RISC designs. The simplest method to achieve this is to use a top-down approach, in which successive features are examined by focusing the attention in ever finer

subsets of the computer architecture. Following this approach we come to the architectural characteristics discussed below.

1. **Word width**

   The first important feature of the processor and memory ensemble is the word width used by the processor. Most current RISC processors use a 32 bit internal and external word width. This means that the integer registers, the address and data paths are restricted to this number of bits. There are nevertheless a few RISC processors which already use partial 64 bit architecture.

2. **Split or Common Cache**

   RISC processors need a cache between them and main memory. But this cache can be a common one, in which instructions and data are mixed, or it can be a split unit, in which two separate caches hold respectively instructions or data. The efficiency of both caching methods is very similar, but the split approach is used in many RISC designs.

3. **On-chip or off-chip cache**

   Some RISC processors use an on-chip cache because it is faster to access, although it increases the chip complexity and therefore the chip area. Other processors were designed with an off-chip cache in mind (like the SPARC chip "Scalable Processor ARChitecture"), in order to simplify the design of the integer unit. CISC processors, like the Intel 80486, use an on-chip cache in order to cut the performance advantage of RISC processors.

4. **Harvard or Princeton Architecture**

   In systems with a split cache it is possible to use separate data and address buses for each cache separately. In this case an instruction fetch can be handled in parallel with a data access. This is called Harvard architecture. Princeton architecture uses a common bus to access data and instruction cache. It should be noticed that Harvard architecture does not mean separate buses from the cache to main memory. From the processor to the two cache units two buses are used, but the cache units share a single bus to main memory.

5. **Prefetch Buffer**

The instruction stream to the processor can be handled with an additional level in the memory hierarchy. Fast prefetch buffers can access the instruction cache sequentially in advance in order to hold several instructions ready to be consumed by the processor. This structure is called a prefetch buffer. Only few RISC processors use prefetch buffers. This kind of buffer is very important for processors which try to achieve the maximal instruction issue rate.

6. **Write Buffer**

The equivalents to prefetch buffers on the data stream side are write buffers. The processor does not have to wait until some data has been written on the cache. It just gives a write request to the write buffer and special hardware handles the request autonomously.

7. **Coprocessor or Multiple Units Architecture**

This is one of the decisive classification criteria for RISC processors. Coprocessor architecture means that the instruction stream is analyzed concurrently by two or more processors. Each processor takes the instructions that it can handle, the others interpret it as a NOP. The processors can communicate through memory or through special control lines. Multiple unit architecture means that there is a central decoding facility which starts execution units according to the instruction which has been decoded. The decoding unit, for example, can start an integer addition in the integer unit - one cycle later it can start the floating point multiplication unit, and so on.

8. **Common Register File or Private Registers**

In coprocessor architecture each processor handles its own registers and register interchange is managed thorough memory. In multiple unit architecture there are two possibilities: a common register file can be accessed by all execution units or the execution units themselves can work with private registers. A combination of these two extremes is also possible.

9. **Width and Number of Internal Data Paths**

The performance of execution units can be enhanced by using more and wider data paths in the internal architecture of a processor. It makes a performance difference if 64 bits have to be transferred from the registers in one or two 32 bit steps. Two write-back paths to the register file are better than one mainly in processors with multiple units.

10. Condition Codes

Control of execution flow has been achieved traditionally through the use of condition bits which are set as a side effect of some arithmetical or logical operations. Several RISC processors set condition bits explicitly in one of the general purpose registers. This register can then be tested by the branching instruction. This strategy avoids the problems associated with a long pipeline in which it is not completely clear which instruction changed the condition codes the last time

11. Register Renaming and Scoreboarding

In RISC processors the management of the register file is an essential feature. There are three different ways to solve the scheduling problem for the usage of registers: the first solution is to schedule registers in software and to avoid collisions through a sophisticated compile time analysis. The second solution relies on the help of a special hardware "scoreboard" that tracks the usage and availability of registers. Whenever a register which is not yet free is requested, the scoreboard locks the request until the register is available. The third solution is that registers are dynamically renamed by the hardware. If two instructions need register R2 to generate a temporary result, one of the two gets access to this register and the other to a "copy" of R2. The results are calculated and the real R2 is updated according to the sequential order of the calling instructions.

12. Pipelining Depth of Multiple Units

In chips with multiple units an important parameter is the pipeline depth of each unit. Floating point units are implemented with a deeper pipeline, taking into account the longer latency of floating point operations. An important question is how the pipelines of different depth are coordinated so as to avoid collisions at

the exit of the pipelines, when more than one unit could try to access the register file.

13. Chaining

Another important question is if the output of execution units is to be directly connected to the input of other execution units. If this is the case something similar to the so called "chaining" of vector processors is available. The multiplier, for example, can be directly connected to an adder and in this way the inner product of two vectors can be calculated extremely fast.

14. Multiple purpose architecture

The last architectural feature of interest is if the processor being considered exhibits general purpose architecture or not. A general purpose chip needs to implement interrupts; protection levels and uses a memory management unit. Almost all RISC processors provide these features. The ones that do not provide them have been designed for embedded applications or for simple multiprocessing nodes.

4.3 The Confusion around RISC Concept

The motivation for the design of RISC processors arose from technological developments which changed gradually the architectural parameters traditionally used in computer industry.

The philosophy of the time was to build machines which could diminish the semantic gap between high level languages and the machine language. Many special instructions were included in the instruction set in order to improve the performance of some operations and several machine instructions looked almost like their high-level counterparts. If anything was to be avoided it was, first of all, compiler complexity.

At the implementation level, microcoding provided a general method of implementing increasingly complex instruction sets using a fair amount of hardware. Microcoding also made possible to develop families of compatible computers which

differed only in the underlying technology and performance level, like in the case of the IBM/360 system.

The measures used to assess the quality of a design corresponded directly to these two architectural levels: the first measure was code density, i.e., the length of compiled programs; the second measure was compiler complexity. Code density should be maximized, compiler complexity should be minimized.

There were good reasons for microcoded designs in the past. Memory was slow and expensive, therefore compact code was required. There was a need for instructions of high encoded semantic content which could maintain the processor running at full speed with a minimum of instruction fetches. Microcode had also an additional advantage: it could be changed in different models of the same computer family, allowing for increased parallel execution of individual instructions in the high end of the family. The transition from the use of core memory (with typical cycle times 10 times slower than semiconductor memory) to the now used dynamic and static memory chips eliminated one of the advantages of microprogramming. Microprograms and real programs could be stored in the same kind of devices with comparable access times. The introduction of cache memories in the early seventies altered the equation again in favor of external programming against microprogramming.

When RISC is understood as just the name of a bundle of architectural features for processors, the most frequently mentioned are:

1. Small instruction set
2. Load/store architecture
3. Fixed length coding and hardware decoding
4. Large register set
5. Delayed branching
6. Processor throughput of one instruction per cycle in average

The difference between RISC as design philosophy and RISC as a bundle of features

is something which remains obscure in the popular computer literature. There is no clear view of the interdependence of the diverse features. Processor throughput, for example, is a dependent variable of decoding time, but not the other way around. We already mentioned that in most cases RISC is understood as meaning just a "small" instruction set.

4.4 Pipelined RISC Processor

The term RISC (Reduced Instruction-Set Computer) is somewhat misleading in the context of the technology of the 1990s. The original notion of RISC is to create a machine with a very fast clock cycle that can execute instructions at the rate of one per cycle. RISC machines are often associated with pipeline implementation because pipeline techniques are natural ones to achieve the goal of one instruction execute per machine cycle.

The implementation of this idea began at IBM in the mid-1970s, eventually led to the development of an internal machine called the 801 computer. To achieve the fastest possible cycle rate, this type of architecture reduces decoding delays by requiring that all instructions conform to a simple format.

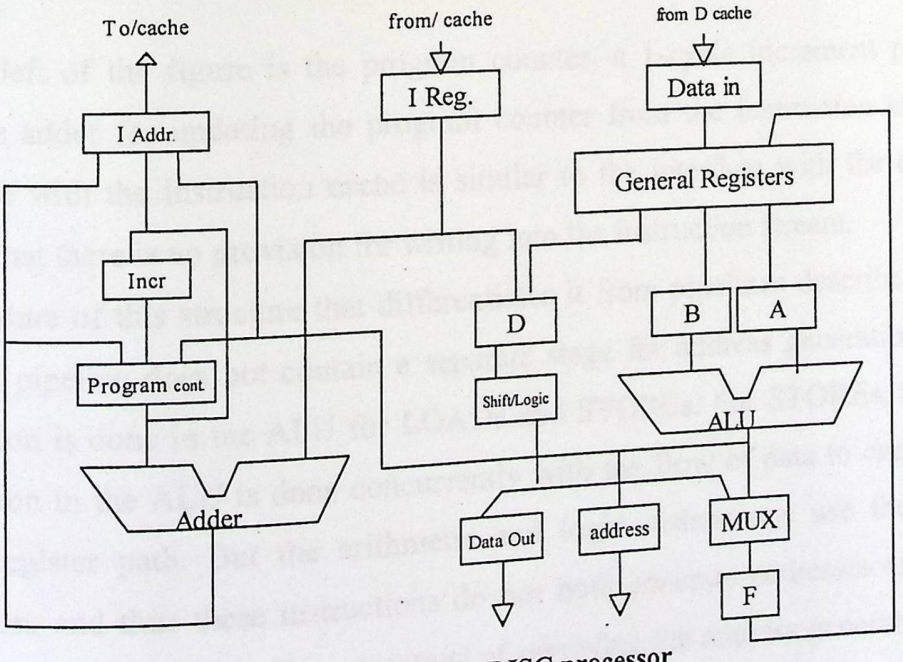

Figure4.1: the structure of pipeline RISC processor

The original development explored a format in which all memory operations are LOAD and STORE operations, and all other operations such as ADD and COMPARE operated exclusively on registers. All instructions are of one length, and the instruction fit a scheme compatible with a pipeline implementation in which each stage of the pipeline performs roughly the same type of operation as each instruction passes that stage.

Figure4.1 considers the structure of a typical RISC processor, this figure is a concrete example of RISC machines. The central part of the figure is a cyclical 3-stage pipeline. It starts at the bank of general registers whose outputs are latched in one clock cycle in the A, B, and D registers. The A and B drive an Arithmetic/ Logic Unit (ALU) to an output register, F. The F register, in turn, stores back into the general registers.

The interface to the data cache is through two paths, one for data to cache and one for the cache address. The cache address is developed at the output of ALU and is locked into the D-cache address register. Data to the cache passes from the general registers to the D register and from there through a shift/logic unit to the D-cache data out register. Data from the cache to the processor is latched into the D-cache data in register, and from there it is stored into the general registers.

At the left of the figure is the program counter, a 1-cycle increment path, and a separate adder for updating the program counter from the instruction stream. The interface with the instruction cache is similar to the interface with the data cache, except that there is no provision for writing into the instruction stream.

One feature of this structure that differentiates it from pipelines described earlier is that the pipeline does not contain a separate stage for address generation. Address generation is done in the ALU for LOADs and STOREs. For STOREs, the address generation in the ALU is done concurrently with the flow of data to cache through the D-register path. But the arithmetic and logic instructions use the ALU for execution, and thus these instructions do not both generate addresses and perform arithmetic manipulations. The advantage of removing the address generation stage is

that the pipeline is shorter and the penalty is less when interlocks or conditional execution cause the pipeline to empty. The offsetting disadvantage is that since arithmetic instructions cannot also load data from memory; this architecture requires more LOADs than one that has an address-generation stage.

Since the primary goal of the architect is to produce a machine that keeps the pipeline filled at all times, this machine has three read and two write ports to the general registers array. All five ports can be active concurrently without conflict. This is an unusual design in the light of our assumption that conventional memory can support one access per cycle. Multiple ports can be built economically for small amounts of memory such as for the general registers. But there is a definite cost in chip area for this capability. Because the function is not free, the function has to pay its way by adding dramatically to performance. And this is the case in the example. With the potential for performing two writers and three reads concurrently on any cycle, the processor can execute a sequence of register instructions at the rate of one per cycle, provided the register instructions are non-conflicting. That is, the input register instruction is not altered by its immediately preceding instructions.

The three-cycle pipeline can easily be reduced to a pipeline with an effective length of two for many cases. The idea is that when a specific general register is written and read in the same cycle, the data reported out by the read should be the new data written into the register, not its former contents. Consequently, when data from the F register is written to some general register, that data can flow from the general register array and be written into the A, B, or D registers in the same cycle.

LOAD instructions and cache misses also force the pipeline to empty occasionally. The address of a LOAD instruction is generated and latched in thee D-cache address register as a normal pipelined computation. At a later time, the data from the cache are latched in the D-cache data- in register. Until the data appear there, an interlock prevents any instructions from using the destination register of the input data. The

interlock is essentially a register interlock, except that it can be active for many cycles if a cache miss occurs.

The instruction decode pipeline is independent of the execution pipeline except that conditional branches have to be interlocked to the results of data registers. The instruction pipeline working in lock step with the execution pipeline, performing in one cycle each of the operations: generate instruction address, accept instruction from cache, and decode instruction. This will operate in synchrony with the execution unit as long as interlocks or conditional branches do not break the synchrony. If the execution pipeline unit stalls for some reason, the instruction pipeline can continue if there is work to be done. In fact, the unconditional branches can be performed in the instruction pipeline free of interaction with the execution pipeline unless the target address has to be computed from register contents that are currently locked from access.

Performance analysis of pipelined processors such as the RISC machine in this example is normally done by calculating the average number of cycles per instruction executed. The goal is to bring this down to 1, which is minimum value for a machine that executes at most one instruction per cycle. As a simple example of this analysis, assume that conditional branches occur once every 4-instruction, and that the processor logic can guess the outcome of the conditional branch correctly 80 percent of time. A correct guess produces no penalty, and the penalty for an incorrect guess we assume to be 2-cycles (1-cycle to calculate the condition, and 1-cycle to fetch the instruction for the other branch). Then on 25 percent of the instruction, there is a 20 percent penalty of 2 cycles, or an average penalty of a 0.1cycles per instruction. Hence the conditional-branch effect raises the number of cycles per instruction from 1.0 to 1.1. If the data cache misses 2 percent of the time, and each miss produces 10 cycles of empty output from the pipeline, then data cache misses add 0.2 cycles per instruction. The effect increases the number of cycles per instruction to 1.3.

The finite-cache effect may occur jointly with a conditional-branch effect, in which case one delay is overlapped with another. Adding delays caused by the two effects individually produces a number that is higher than what is actually observed. If the error introduced by simple addition of delays is sufficiently large, the performance estimate may require a more detailed model or may have to be done through simulation.

According to the previous figure of RISC architecture for attaining higher speed by simplifying the work done per stage.

For example registers (A, B, and D) serve as staging registers for data from the general registers. This machine takes two clock cycles to move data from the general registers through the ALU to the F register. The fist clock cycle is associated with the time required to decode the register fields of an instruction, send these fields to the register array, and retrieve the data. The cycle ends by latching the data into the A, B, and D registers. The next cycle carries the data through the ALU to the F register.

Although the pipeline is shorter, the performance is not necessarily better. The maximum rate of completion in either design is one instruction per clock time. The shorter pipeline has a longer clock cycle, so its maximum rate of computation is lower than for the original design in the previous figure.

This is the major advantage of RISC architecture. But the advantage is partially offset by paying a potentially greater penalty for an empty pipeline in the original design. Register and conditional-branch interlocks potentially can leave the pipeline empty for a greater proportion of time. For example, it is possible to design the timing in a way that a result produced in the F register on one cycle can participate in the next cycle and influence the result to be stored in the F register on that cycle. In this example no register interlocks are required, but the original design has to be capable of deferring the execution of an instruction when one of its operands is in the process of being computed and has not yet reached the F register. When comparing the RISC design to more conventional designs, it is clear that the cycle time of a RISC computer can be made somewhat smaller than the cycle time of computers

with richer and more complex instructions. Although this advantage of pure RISC architecture is fast cycle time, the RISC architecture could easily be a poor performer if the reduced instruction set were its only characteristics. At least two problems are evident:

- RISC architectures lack the more powerful instructions of CISC architectures, and therefore they must execute more instructions to do the work of CISC architecture.
- In executing more instructions to do equal work, at best data traffic may be the same for CISC and pure RISC machines, but instructions traffic must be higher so RISC machine require higher instruction bandwidth to do equal work of CISC architecture.

Because of these problems, the benefits of the fast cycle of a RISC machines are partially lost. If a RISC machines lacks crucial complex instructions, such as integer divide and multiply and the full spectrum of floating-point instructions, then a RISC machine performance would almost surely fall below that of a CISC machine on workloads with a hefty percentage of numerical operations.

There is then very little point in building a pure RISC machine with only a fast clock cycle. It is necessary to address the negatives of the pure architecture and embellish the RISC architecture so that it has higher performance than a pure RISC machine, with a little compromise on the cycle time as possible.

The integer divide is one of many complex instructions that are not directly compatible with the RISC philosophy. An architect must investigate how to add such instructions to RISC architecture in a way that will improve net performance at low cost.

The second problem mentioned above, high instruction fetch traffic; can easily be resolved by incorporating an instruction cache into the VLSI implementation of a RISC machine where a CISC machine might not have such a cache. The point here is that RISC design has eliminated the logic for decoding complex instructions; that logic can be utilized instead in an instruction cache.

The Good idea of CISC machines can show up in RISC machines, and conversely, good idea developed for RISC machines can be used in CISC machines. If an idea is good, and by using it performance improves, then the architect should use the idea. This means that the delay-branch instructions, originally proposed for microcode and the used in RISC machines, can easily find their way to CISC machines, just for floating-point instructions have moved from CISC machines to RISC machines. Here is a study to the effect of register windows on RISC and CISC machines. The idea here is to view register windows as an independent feature of an instruction set and determine its net benefits in performance. RISC architecture was studied with and without register windows and two CISC architectures with and without register windows. The technique used for the study was to run a common set of benchmarks on the architectures and evaluate performance in the terms of relative traffic between processor and memory. The benchmarks chosen were rather unusual because they use procedure calls intensively and are probably not representative of most workloads. The reason for this selection of benchmarks is that differences attributable to register windows are most likely to be visible in such benchmarks, so they represent an upper bound on the effect of register windows. Register windows are likely to affect performance much less when realistic workloads are used.

The main difference between RISC and CISC is that the instruction set of the first kind of processors was explicitly designed to allow the sustained execution of instructions in one cycle as average. CISC processors can also approach this objective, but only at the expense of much more hardware logic capable of reproducing what RISC processors achieve through a streamlined design. Some RISC processors, like the SPARC, achieve a sustained speedup of 2.8 running real

applications. This means that the SPARC is a parallel engine capable of working on about three instructions simultaneously. Other RISC processors offer similar performance.

RISC processors are processors with an instruction set whose individual instructions can be executed in one cycle exploiting pipelining. Pipelined supercomputers and large mainframes have used pipelining intensively for years, but in a radically different way as RISC processors. In IBM mainframes, for example, the instruction set was given by "tradition" and pipelining was implemented in spite of an instruction set which was not designed for it. Of course there are ways to accommodate pipelining, but at a much higher cost.

In summary: taking pipelining as the starting point, it is easy to deduct all other features of RISC processors. The fundamental question is: what is needed in order to maintain a regular pipeline flow in the processor? The following RISC features constitute the answer:

a. **Regular pipeline phases and deep pipelines**

   First of all the logical levels of the processing pipeline must be defined and each one must be balanced against each other. Going through each pipeline stage must take the same time and all the work done in the execution path should be distributed in the most uniform way. Each pipeline stage takes a complete clock cycle. Typical processors use a clock cycle time at least so large as the time it takes to perform one typical ALU operation. It is clear that this restriction imposes a heavy burden on the designer of microprocessors. In each stage of the pipeline a maximum of 10 logic levels can be traversed. The computer architect must try to parallelize each one of the phases internally in order to use a minimum of logic levels. This is easier if the pipeline phases are correctly balanced and if they are as independent from each other as possible, so as not to have to handle signals running from one stage to the other. Typical RISC processors go beyond the classical three level pipelines and use pipelines with four, five or six levels. A deeper pipeline means more potential parallelism but also more coordination problems.

b. Fixed Instruction Length

In CISC processors, instructions are of variable length and several words have to be fetched until the whole instruction can be completely decoded. This introduces a variable element in the duration of the fetch stage which can stall the pipeline if the decoding stage is waiting for an instruction. Large processors avoid this problem with a prefetch buffer which can store many instructions of the sequential stream. CISC microprocessors use also small prefetch buffers or several words of instruction cache.

The simplest technique for avoiding a variable fetch time is to encode each instruction using a fixed one word format. The fetch stage has in this way a fixed duration and one instruction can be issued each cycle to the decoding stage under normal pipeline flow. The decoding stage does not need to request additional instruction bytes according to the encoding of the instruction and there is no need for any additional control lines between the fetch and decode stages.

c. Hardwired decoding

Typical RISC processors reserve 6 bits out of 32 for the opcode of the instruction. The operands and the result are typically held in registers. Each argument is encoded, using for example 5 bits. Thirty-two registers can be referenced in this way. Decoding of the opcode and access to the register operands can be done simultaneously, which is a very important feature if the operands are to be ready for execution in the next cycle.

In case of the operands is a constant (that must be stored or added to in a register) it is encoded using an overlapped format. This poses no problem for the decoder, because this constant can be decoded simultaneously with the access to the argument registers. One register too much will be read, but this intermediate read can be discarded without losing any cycles. So, decoding of a fixed instruction format can be done in parallel in a clock cycle.

d. Register to Register Operations

The execution phase of an instruction should also take one clock cycle as a maximum whenever possible. Arithmetical instructions which access operands in memory do not fulfill this condition because the long latency of memory accesses keeps the ALU waiting several cycles. Register to register operations avoid this inconvenience. Instructions like integer multiply or divide can be directly implemented in the ALU, but they take several cycles to complete and they stall the pipeline. Some RISC processors, like the SPARC, do not directly implement multiply and divide.

e. Load/store Architecture

If all operands for arithmetic and logical operations are located in registers, it is obvious that these registers have to be loaded first with the necessary data. This is done in RISC processors using a "load" instruction, which can access bytes, halfwords or complete words. A "store" instruction transfers the contents of registers to memory.

Without special measures the processor must wait after each load instruction for the memory to deliver the wished data "pipeline stalls". RISC processors avoid this problem using a "delayed" load. The load instruction is executed in one cycle but the result of the load is made available only one or more cycles later. This means that the instruction following the load must avoid using the register being loaded as one of its arguments. In most cases this condition can be enforced by the compiler, which tries to reschedule the instructions so that the load does not have to stop the pipeline. When this rescheduling is not possible, the load stalls the pipeline for as many cycles as the main memory or cache takes to respond.

f. Delayed Branching

Instructions are fetched sequentially but a taken branch can alter the sequential flow of instructions. After a taken branch a new instruction located at the branch target has to be fetched and the pipeline has to be active of now irrelevant instructions.

RISC processors use other strategies. First of all, the branching decision is made very early in the execution path - possibly already in the decode stage. This can be done only if the branching condition tests are very simple, like a register compare with zero or a condition flag test. At the end of the decode phase the processor can start fetching instructions from the new target. But in this decode cycle the next instruction after the branch has already been fetched. In order to avoid stall cycles this instruction can be executed. In this case the branch is a delayed branch. From the programmers point of view the branch is postponed until after the next instruction is executed. The compiler tries to schedule a useful instruction in the location after the branch, which is called the "delay slot." Some RISC processors with very deep pipelines schedule up to two delay slots. More delay slots make the scheduling of useful instructions increasingly complicated and in many cases the compiler ends writing NOPs in them.

Delayed branching is not strictly a RISC innovation. This kind of branching was used before in microprograms but certainly not in macroinstruction sets. Another technique borrowed from mainframes is the so called "zero cycle" branching. After each prefetch of a branch special hardware tries to predict if the branch will be taken or not. The next instruction is then prefetched from the predicted target address. In this case no delay slots are needed. If a special branching processor is included branches can be preprocessed and filtered out so that the arithmetical processor receives only a sequential instruction stream.

g. Software scheduling and Optimizing Compilers

The whole benefit of a RISC architecture can be reaped only if the compiler is sophisticated enough to rearrange instructions in the optimal order. RISC architectures try to maximize the synergy between hardware and software. Optimizing compilers are not an optional feature of RISC systems but one of their essential components.

h. High Memory Bandwidth

If instructions are to be fetched, decoded and executed in one cycle steps, a huge memory bandwidth is required. Using a 20 MHz processor and dynamic RAM chips with 100 ns cycle time some form of intermediate cache is needed, capable of delivering at least one word per cycle. RISC processors depend on a complex memory hierarchy in order to work at full speed. In most of them, separate data and instruction caches try to avoid contention for the system bus when a fetch is overlapped with a register load or store. For this reason most RISC processors include memory management components. A RISC processor without management of a memory hierarchy could hardly outperform a CISC processor because the latter encode much more semantic information in each instruction.

From the above discussion it should be clear that all of the discussed RISC features are part of a common strategy to guarantee an uninterrupted pipeline flow, and in this way, a high level of parallel execution of sequentially coded programs. Fixed word encoding, hardwired decoding, delayed loads, delayed branches, etc., are just ways to achieve a regular pipeline flow. Some of these features could disappear in future RISC designs or not be used in others. The essential point will remain being the exploitation of instruction level parallelism.

4.5 The Features of RISC Processors

In this section we review some of the most important and popular RISC processors. A drawn for each processor the corresponding Kiviat graph. This type of graphical representation has been used in other architectural studies and in many fields in which the representation of several dimensions of data must be handled in just two dimensions. In doing this we tried to make the design of the Kiviat graph as expressive as possible in order to facilitate the comparison of different kinds of processors.

The variables considered in the comparison of processors are the following: number of pipeline stages, number of addressing modes, number of instructions, method of branch handling, average CPI according to some authors, number of registers, instruction length (fixed or variable) and levels of decoding (one level for hardware decoding, two for microcode, and three for micro plus nanocode). The circle meets the points in the different data axis that could be considered as "typical" RISC values. A pipelining depth of four stages, for example, could be considered as a normal feature of RISC technology. More pipelining makes the processor potentially faster if the other associated features have the adequate values. One single addressing mode is normally associated with load/store architecture. Several RISC processors use just 6 bits for the encoding of instructions: this means that only 64 instructions can be encoded. One delayed branch slot could be considered normal in most RISC designs, but there are other alternatives. With this information in mind we can look now at several commercial RISC processors.

- **Intel 860**

Intel developed the 80860 processor with embedded applications in mind. It was the first RISC chip of the semiconductor manufacturer and silicon area was not spared - more than one million transistors were used in the final design. The chip has not been a great market success.

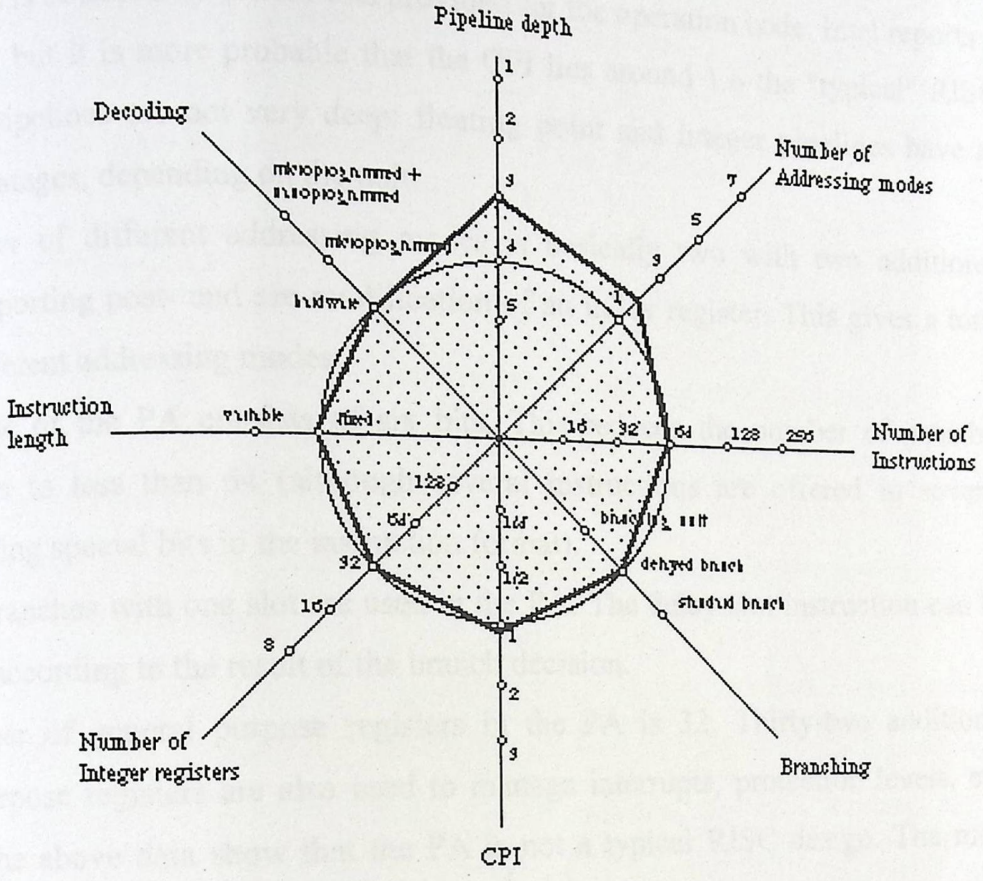

Figure4.2: Kiviat graph

The I860 is a 32 bit processor built with Harvard architecture. The bus to the instruction cache is 32 bits wide, and the bus to the data cache is 128 bits wide, making possible to access four words in parallel. The caches are located on-chip. The chip follows the multiple units' paradigm and provides one floating point adder, one floating point multiplier and one special graphics unit. The "RISC core" contains 32 bit registers and one ALU. A scoreboard controls the allocation of general purpose registers.

The floating point register file contains 30 registers 32 bits wide, which can be used as 15 64 bit registers. The adding and multiplying units can be chained to speed-up the multiply and add combination needed in linear algebra and graphics. The processor uses a fixed instruction format very similar to the MIPS format,

decoding is hardwired, and only two addressing modes are provided. The number of instructions is bounded by the six bits provided for the operation code. Intel reports a CPI of 1.1, but it is more probable that the CPI lies around 1.6 the "typical" RISC CPI. The pipelines are not very deep: floating point and integer pipelines have at most three stages, depending on the unit.

The number of different addressing modes is basically two with two additional modes supporting post- and pre-modification of an index register. This gives a total of four different addressing modes.

The opcode of the PA consists of six bits. This reduces the number of possible instructions to less than 64 (although several instructions are offered in several variants using special bits in the instruction format).

Delayed branches with one slot are used in the PA. The delay slot instruction can be cancelled according to the result of the branch decision.

The number of general purpose registers in the PA is 32. Thirty-two additional special purpose registers are also used to manage interrupts, protection levels, etc. some of the above data show that the PA is not a typical RISC design. The most atypical feature, however, is the low level of pipelining of the first processors offered. Just three pipeline stages are used, although newer designs can employ a deeper pipeline. The pipeline implements interlocks in hardware. The optimal pipeline flow requires software scheduling.

The PA achieves a low CPI through simultaneous execution of scalar and floating point operations. The number of floating point units can vary from one PA machine to another. The PA tries to achieve low CPI using superscalar techniques.

4.6 CISC Architectures

CISC is an acronym for Complex Instruction Set Computer and are chips that are easy to program and which make efficient use of memory. Since the earliest machines were programmed in assembly language and memory was slow and expensive, the CISC philosophy made sense, and was commonly implemented in

such large computers as the PDP-11 and the DEC system 10 and 20 machines Most common microprocessor designs such as the Intel 80x86 and Motorola 68K series followed the CISC philosophy. But recent changes in software and hardware technology have forced a re-examination of CISC and many modern CISC processors are hybrids, implementing many RISC Principles.

4.6.1 Instruction Set

The minimal CISC instruction set presented here is stack-based and composed entirely of zero-address instructions. There are only 8 instructions, so each can be coded as a 3-bit syllable. Assuming a 16 bit word, 5 instruction syllables can be packed in each word (the least significant syllable is the first). The instructions are:

(000) NOP: No operation.

(001) DUP: Duplicate the stack top. This is the only way to allocate stack space.

(010) ONE: Shift the stack top left one bit, shifting one into the least significant bit.

(011) ZERO: Shift the stack top left one bit, shifting zero into the least significant bit.

(100) LOAD: Use the value on the stack top as a memory address; replace it with the contents of the referenced location.

(101) POP: Store the value from the top of the stack in the memory location referenced by the second word on the stack; pop both.

(110) SUB: Subtract the top value on the stack from the value below it, pop both and push the result.

(111) JPOS: If the word below the stack top is positive, jump to the word pointed to by the stack top. In any case, pop both. This instruction set is not particularly convenient, but that is not the point of this exercise. The important thing is that it is very simple yet it is sufficient to write any program, given enough memory, and given memory mapped input-output devices.

4.6.2 Characteristics of CISC

The design constraints that led to the development of CISC (small amounts of slow memory and fact that most early machines were programmed in assembly language) give CISC instructions sets some common characteristics:

- A 2-operand format, where instructions have a source and a destination. Register to register, register to memory, and memory to register commands. Multiple addressing modes for memory, including specialized modes for indexing through arrays.
- Variable length instructions where the length often varies according to the addressing mode.
- Instructions which require multiple clock cycles to execute.
- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.
- A small number of general purpose registers. This is the direct result of having instructions which can operate directly on memory and the limited amount of chip space not dedicated to instruction decoding, execution, and microcode storage.
- Several special purposes registers. Many CTSC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design somewhat, at the expense of making the instruction set more complex.
- A 'Condition code" register which is set as a side-effect of most instructions. This register reflects whether the result of the last operation is less than, equal to, or greater than zero and record if certain error conditions occur.
- At the time of their initial development, CISC machines used available technologies to optimize computer performance. Microprogramming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit. The ease of micro coding new instructions allowed designers to make CISC machines upwardly compatible: a new computer

could run the same programs as earlier computers because the new computer would contain a superset of the instructions of the earlier computers. As each instruction became more capable, fewer instructions could be used to implement a given task this made more efficient use of the relatively slow main memory. Because micro program instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated.

4.6.3 CISC Problems

The CISC philosophy had its own problems, including:

- Earlier generations of a processor family generally were contained as a subset in every new version, so instruction set & chip hardware become more complex with each generation of computers. So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length. this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.
- As memory speed increased, and high-level languages displaced assembly language, the major reasons for CISC began to disappear, and computer designers began to look at ways computer performance could be optimized beyond just making faster hardware.

CISC and RISC implementations are becoming more and more alike. Many of today's RISC chips support as many instructions as yesterday's CISC chips. And today's CISC chips use many techniques formerly associated with RISC chips.

4.6.4 CISC Philosophy

- **Use Microcode:** The earliest processor designs used dedicated (hardwire) logic to decode and execute each instruction in the processor's instruction set. This worked well for simple designs with few registers, but made more complex architectures hard to build, as control path logic can be hard to implement. So, designers switched tactics they built some simple logic to control the data paths between the various elements of the processor, and used a simplified microcode instruction set to control the data path logic. This type of implementation is known as a micro programmed implementation. In a micro programmed system, the main processor has some built-in memory (typically ROM) which contains groups of microcode instructions which correspond with each machine-language instruction. When a machine language instruction arrives at the central processor, the processor executes the corresponding series of microcode instructions. Because instructions could be retrieved up to 10 times faster from a local ROM than from main memory, designers began to put as many instructions as possible into microcode. In fact, some processors could be ordered with custom microcode which would replace frequently used but slow routines in certain application.

There are some real advantages to a microcoded implementation:

- The microcode memory can be much faster than main memory.
- An instruction set can be implemented in microcode without losing much speed over a purely hard-wired implementation.
- Using microcoded instruction sets, the IBM 360 series was able to offer the same programming model across a range of different hardware configurations.
- **Build "rich" instruction sets**: Designers were enhancing their instruction sets with instructions aimed specifically at the assembly language programmer. Such enhancements included string manipulation operations,

- **Build high-level instruction sets**: Once designers started building programmer-friendly instruction sets, the logical next step was to build instruction sets which map directly from high-level languages. Not only does this simplify the compiler writer's task, but it also allows compilers to emit fewer instructions per line of source code. Modern CISC microprocessors, such as the 68000, implement several such instructions, including routines for creating and removing stack frames with a single call.

4.7 Comparing RISC with CISC

There has been much discussion about the relative advantage of CISC and RISC architectures. Some difference that many of the techniques used in RISC processors can be translated also to CISC designs. It is possible for example to rewire the processor in order to execute most of the simple instructions in one cycle. Or it is possible to use a pipelined microengine, like in the Vax, in order to speed up execution.

RISC features can be introduced in CISC processors only at the expense of much more hardware. It is possible, for example, to program the pipeline of a CISC processor to use the dead time between the load and store of one instruction argument in memory. The microengine works in this case following a load/store model, and it dynamically reschedules the operations needed by the macrocode. This dynamical rescheduling is too expensive compared to the software scheduling used in RISC processors. Software scheduling must be done only once and then it runs without complex hardware. Dynamical scheduling needs increasing amounts of logic.

CISC processors can still be made competitive to RISC processors if the cycle time is reduced. There are already prototypes of Intel 80386 microprocessors running at clock frequencies as high as 50 MHz. Such processors can outperform RISC designs

running at a slower clock rate. But RISC processors are better positioned to achieve greater reductions in the clock cycle time in the long run. The cycle time is determined by the following factors: pipelining depth, amount of logic in each stage and the VLSI technology used. If the first and third factors are fixed, it is the amount of logic, i.e., the number of logic levels in each pipeline stage, the factor which determines the clock cycle time. It is much more difficult to reduce the number of logic levels in a complex design as in a simple one. RISC processors can achieve larger reductions in the clock cycle time with a lower investment in design time. Reducing the clock cycle time of CISC processors is not impossible, but much more difficult.

It is also very difficult to increase the pipelining depth in CISC processors. Using RISC technology, it is possible to think about superpipelined processors capable of working with a pipeline of eight or nine stages. This is something being investigated by the designers of the MIPS series. In summary: the controversy surrounding CISC versus RISC designs can not be settled just by looking at the present performance differences of the two technologies. If this were the case, then it should be admitted that CISC microprocessors have come nearer to the performance of RISC designs in the last two years. But the question is which design philosophy will be capable of climbing the performance ladder faster in the next few years. Here RISC designs appear as potentially much faster than CISC processors, which have already come close to their "physiological" limits whereas RISC is still in its infancy.

RISC processors can be distinguished from CISC designs mainly on one count: their efficient utilization of instruction pipelining. RISC processors have been defined explicitly with the aim to exhaust the available instruction level parallelism available in typical programs. All other RISC features can be logically derived from this initial purpose. RISC has also been called a "scalable architecture" because it is possible to go from one technology to another with practically the same design (from CMOS to ECL, for example). The first mainframes with a Reduced Instruction Set should

appear in the next years. Although the future belongs to the superpipelined/superscalar processors, CISC designs will not disappear so fast, just because of the enormous installed base of such computers. RISC and CISC will peacefully coexist until CISC adopts so many features of RISC that it will be hard to tell the difference.

4.7.1 Control Unit in RISC and CISC

The control unit is a finite state machine that takes as its inputs the IR, the status register (which is partly filled by the status output from the ALU), and the current major state of the cycle. Its rules are encoded either in random logic, a Programmable Logic Array (PLA), or Read-Only Memory (ROM), and its outputs are sent across the processor to each point requiring coordination or direction for the control unit. The ALU function select takes the instruction op code and translates it into a given function of the ALU (either one line per ALU function or a compact binary code for the function).

The Jump/Branch/PC depends on the instruction type and in a RISC architecture these may be directly coded in the op code. Read control Occurs at the start of an instruction cycle. IR latch and occurs at the end of the fetch state. Load control happens at the end of the data fetch state of a load instruction. Load Reg/Reg again depends on the op-code. Register R/W is in the start of the data Fetch stage and at the write back stage of an operation. It thus depends on the major state and the instruction. CISC architecture typically uses a more complex control unit. As we've noted before, the IR is often multiple words, and the control unit has to look at Different parts of the IR at different stages of execution. In fact, the entire IR may not be available at once; requiring interlocks with fetch logic to ensure the contents of the IR are valid. There are many more control signals coming out of a CISC control unit, partly to control the more complex addressing logic, but also to directly connect to the many special purpose registers. In RISC architecture, the registers are accessed uniformly in a block so a simple decoder in the register file can select the

particular register. In CISC architecture, there are restrictions on the particular registers that can be used by a given instruction and these are enforced by the control unit.

To begin the design of a control unit, we start by listing every control signal in the instruction/data path of the processor. This becomes a list of the control unit's outputs. As input, it has the instruction register, any status information (such as branch flags, interrupts, etc.) from the processor, and a "major state" which simply keeps track of where we are in the execution of an instruction. We always begin an instruction with state 0, which corresponds to fetch. During that state, the control unit outputs the necessary signals to route the contents of the PC to the memory address port, to select and clock the memory until it responds with data from that location, and then cause this data to be latched into the IR.

In CISC architecture, the Fetch may only retrieve the first part of an instruction, and (depending on bits in the IR that are then decoded by the control unit) more words may need to be fetched.

In RISC Architecture, a single Fetch retrieves a complete instruction, so we may proceed to the next major state, which is usually to begin fetching data from the registers, while we decode the instruction. In RISC architecture, "decoding an instruction" mainly means that the instruction type field determines what the control unit will do for the remainder of the instructions. If you think of the CU as a finite state machine, the bits in the type field select the next state following the decode. In terms of a program's logic, this is like selecting a branch in a Switch statement each branch of the Switch contains the series of steps to be performed for one type of instruction. For example, after decoding a Jump instruction, the control unit outputs the signals required to combine the address portion of the instruction with the upper bits of the PC and load the result back into the PC. The CU then returns to the Fetch step. Thus, a Jump has three major states (Fetch, Decode, and Complete). For a memory Load instruction, the CU first sends one of the selected register values (the address) to the address port of the memory (via a multiplexer) and signal the memory to fetch this location. When the memory returns the value, then the CU sends signals

to the necessary multiplexer(s) and the register file so that the memory data goes over the Destination bus and is stored in the designated register. Thus, a Load has four major states (Fetch, Decode, Memory, and Write Back). So, for each type of instruction, and for each major state in each type of instruction, we look at the list of control signals and decide what value each signal must have. In some cases, the value doesn't matter (e.g., if memory isn't selected, it doesn't matter whether it is set to read or write, because it simply won't do anything in either case).

You can think of this as a large 2-dimensional table indexed by instruction type and major state. Within each cell of the table is a list of the control signal and their values. One last bit of control output that we've neglected is the control of the major state itself. This is usually a register, as shown above, that is input to the CU. But it also receives its next value on each clock from the CU. In the above example, the Jump proceeds from State 0 (Fetch) to State 1 (Decode) to State 3 (Complete) and then goes back to State 0. While a Load adds a State 4. In some designs, the state register also encodes the instruction type. Thus, it is really referring to the different states of the finite state

Machine (FSM) rather than the major steps of the instructions. So, for example, the FSM states for a Load might be the sequence 0, 1, 12, 13. The latter two distinguish Memory and Write Back from the complete stage of the Jump. In other designs, we might see Jump going through states 0, 1, 2, and Load going through 0, 1, 2, 3, with the type field used to distinguish the different behavior of the latter states. This is all just a matter of using somewhat different ways of naming the same things. The important point is just that the CU has the inputs it needs to know what it is supposed to be doing on the present clock and what it will do next. In the CU design process, this translate to ensuring that one of the control signals on the list is the "next state" Signal and those we always specify this in every cell of our table.

5

Superscalar Architecture

Chapter 5

Superscalar Architecture

5.1 Introduction

Superscalar Architecture with separate execution units several instructions can be executed simultaneously. In a superscalar architecture, there might be one or more separate Integer Units (IUs), Floating Point Units (FPUs), and Branch Processing Units (BPUs). This implies that instructions need to be scheduled into the various execution units and, further that instructions might be executed out of order. Out-of-order execution means that instructions need to be examined prior to dispatching them to an execution unit, not only to determine which unit should execute them but also to determine whether executing them out of order would result in an incorrect program because of dependencies between the instructions. This in turn implies an instruction unit, IU, which can prefetch instructions into an instruction queue, determine the kinds of instructions and the dependence relations among them, and schedule them into the various execution units.

In superscalar era developed dynamic scheduling hardware and a new stage of computer architecture history has been started.

Each individual superscalar computer, in its hardware, using the same distributed binary of a program code, during the program execution in real time dynamically appoint specific resources of this computer (execution units, register file locations) to each algorithm entity (operations, register locations, buses).

In pre-superscalar computers each code instruction represents a real physical time step of an executable program; a reference to register location implies a real physical register location; a reference to operations (opcode) implies a real physical execution unit (though a single one in a pre-superscalar case).

Superscalar processors are hardware organizations capable of executing several instructions per clock cycle. They may be considered as the most promising uniprocessor architectures of the post RISC era. Another notable architecture has been implemented. Although they can be viewed as an evolution of RISC architectures superscalar architectures are subject to many more trade-offs than simply widening the pipeline. The term superscalar implies a variety of mechanisms and designs and finding a standardized description of superscalar architectures is not easy.

5.2 Instruction Issue Policies

In a superscalar machine, pipeline stages are concurrent processes that may take care of a variable number of instructions per cycle The overall coordination between processes have a significant impact on the processor ability to exploit the instruction parallelism available in a program. Instruction issue, which is the process of letting an instruction move from the decode stage to the execute stage, determines the processor capacity to discover instructions that can be executed concurrently.
There are three policies for instruction issue as detailed below:

1. In-order issue in-order completion. This policy issues instructions in strict program order and also writes their results in program order. This is the simplest policy to implement and the safest: the saved process state (PC registers and memory) is always consistent with the sequential architectural model) so precise interrupts are easier to implement. However, it is not very efficient: not only does the pipeline stall as soon as a true dependency or a functional unit conflict arise, but also, the instructions are held at the issue stage after slower instructions, even though they have no dependency and would otherwise issue.
2. In-order issue out-of-order completion
   The instructions pass through the issue stage in exact program order, but can bypass each other in the execute stage and may finish execution in different

order. So instruction issuing is continuous unless there are structural hazards or true dependencies. Instruction issuing is not stalled after a long latency operation, which improves performance for floating point operations.
For example, there are notable variations of this scheme: in the IBM RS/6000 instructions issued in program sequence can complete out-of-order between the units (fixed point unit or floating point unit) but have to complete in-order within the same functional unit. In fact, the floating point pipeline is extremely fast, with a two cycle multiply-add operation and so out of order completion of floating point operations is not that critical. Floating point memory reads can overlap floating point operations and bypass them.

3. Out-of-order issue out-of-order completion
   There are severe limitations of the techniques used so far. If an instruction is stalled in the pipeline, no further instruction can proceed. Instead, with out-of- order issue capability the processor is provided with a larger set of instructions selectable for issue. Some instructions must still interlock because of data hazards and structural conflicts, but the processor can look beyond the stalled instructions to discover others which are ready to issue. For this to be possible, the decode and execute stage have to be "decoupled": some kind of instruction pool gathers instructions after decode and before issue. This instruction pool can be a central instruction window which buffers all the instructions or reservation stations in front of each functional unit.

5.3 Data Hazards

When instructions are issued in-order and compete in-order, there is a one to one correspondence between registers and values When instructions complete out-of-order, and issue out-of-order, this correspondence is broken because of storage conflicts between values.

Several levels of complexity and performance can be used in the hardware mechanisms that enforce data dependencies; they have a significant impact on the efficiency of the instruction issue/completion policy.

Out-of-order completion creates output dependencies because instructions write their results in different order than they appear. There exist different techniques to alleviate those dependencies:-

- Scoreboarding

  This simple mechanism uses a reservation table where a picture of the data dependencies is constructed and an additional bit is assigned to each register in the register. This bit is used to indicate if the register is in pending Update, and to interlock an instruction on an output dependence.

- Register Renaming

  Scoreboarding only allows one pending update of a register. Instead, with register renaming, a new instance of a register can be generated for every assignment to a register. Tomasulo's algorithm implements register renaming by associating a tag to each register instance. This tag is then used in place of the register identifier. Appropriate tag management ensures in-order writing in each register. The IBM RS/6000 uses a variant of Tomasulo's algorithm to rename load destination floating point registers.

  Other more complex -but more powerful- implementations use some type of buffering device which provides extra storage for instructions results. An example of such device is the reorder buffer, a FIFO queue where issued instructions place their results. While instructions complete out-of-order, the buffer reorders the results, so that they can be written in strict program order to the register file. There are variants of the reorder buffer: for example, the register update unit; which integrates both reorder buffer and instruction window features within the same device.

Out-of-order issue creates anti-dependencies because register values are not accessed in program order. Anti-dependencies are enforced by identifying the semantic

instances of source registers at decode stage, and associate them with the instructions in the instruction pool. Operands are thus copied with the instructions in the instruction pool in the form of values, register number (scoreboarding case), or tags (register renaming case). The mechanism must ensures that the operands of an instruction will not be overwritten, and that the instruction will eventually issue when the proper operand values are ready and forwarded

5.4 Control Hazards

We now address techniques for avoiding stalls due to control dependencies Control hazards cause a severe performance loss in scalar pipelined processors when the fetcher is stalled until a branch outcome is known. In superscalar machines, the effect of branches is even more critical as instructions need to be consumed at a prodigious rate. Instead of stalling the instruction fetch stage in the presence of branches, most superscalar processors use branch prediction, a mechanism by which they guess which direction the branch is likely to go. There are two types of branch prediction techniques:-

1. Static branch prediction

   This technique can be performed by the compiler, in which case the prediction is associated with the instruction and the processor always predicts that branch taken or not taken. Some form of static branch prediction can also be performed by the hardware, which simply predicts that all branches will be either taken or not taken. The IBM RS/6000 predicts that unresolved conditional branches will not be taken and dispatches sequential instructions past the branch to the execution units, it also prefetches the branch-taken path in case the prediction is wrong. Hardware can also predict a branch outcome based on other static criterias, such as the direction of the branch
2. Dynamic branch prediction

   This technique is performed at run time by the hardware, the results of past executions of a branch are used to a predict future outcome.

Unfortunately, branch prediction is not guaranteed to be correct; when execution is allowed on a predicted path_ there must be some ways to undo the effects of all instructions issued after an incorrectly predicted branch. Again there are different techniques for branch repairs, which rely on post-issue instruction invalidation. Such invalidation is possible when the instruction has not modified the processor state. When instructions can be undone, branch prediction is well integrated in the out-of-order issue scheme, as it extends the processor look-ahead capability. However, when there is no instruction undoing capability, the predicted branch path cannot not pass the decode stage.

6

Simulator

Chapter 6

Simulation

6.1 Introduction

The objective of this document is to present superDLX: a cycle by cycle superscalar simulator using the DLX instruction set the functions that simulate the pipeline stages, and the user interface finally. Superscalar processors are hardware organizations capable of executing several instructions per clock cycle. They may be considered as the most promising uniprocessor architectures of the post RISC era. Some commercial superscalar architecture has appeared on the marke: for example, the IBM RS/6000.

Although they can be viewed as an evolution of RISC architectures, superscalar architectures are subject to many more trade-offs than simply widening the pipeline. The term superscalar encompasses a variety of mechanisms and designs and finding a standardized description of superscalar architectures is not easy.

6.2 SuperDLX

A Superscalar Simulator There is different ways to evaluate the benefits of design ideas for architecture in terms of hardware cost versus performance The most accurate way is certainly to build a prototype, but it is also too time consuming and expensive A more efficient way is to build a trace driven simulator which uses an instruction trace generated by a trace generator of a machine. Trace driven simulation is fast in a run-time sense because the simulation is only concerned by modeling the processor features that affect performance, it does not record values in the register file or memory because it only needs to keep track of dependencies between instructions also it does not compute

any results However because the simulated program is not executed, correctness of simulation is not ensured one might argue that. this is not necessarily a negative point since strict correctness is not indispensable to assess the performance of an architecture.

SuperDLX "the superscalar simulator "is not trace driven. We took this "execution" oriented approach for different reasons":

- Accuracy: We wanted the results of simulated assembly instructions to be effectively computed and the contents of the different hardware elements to be recorded on a cycle basis. Correctness of simulated program output was very important for us to assess the proper coordination of all the different simulated hardware components.
- Portability: We wanted a simulator executing assembly programs generated by a cross compiler so, that it be portable to different machines.
- Usability: Not only did we want an architecture evaluation tool, but also a pedagogical tool aimed at students to learn superscalar mechanisms. The user interface permits one to run the simulator on a cycle basis, and examine the contents of the various hardware elements at a given cycle.

Our simulator implements the most sophisticated superscalar instruction processing policy multiple out of order issue, multiple out of order completion. To achieve this policy, efficient hardware mechanisms were selected for simulation: a central window, buffering instruction for issue, and a reorder buffer, supporting register renaming Other features were built around them to go even further in performance: branch prediction, load and store buffering Taken individually, these mechanisms were very attractive because they seemed rather powerful while conceptually simple Finding the way to

effectively orchestrate them was certainly the most decisive and challenging part of implementation.

6.2.1 The Processor Model

The basic processor model consists of two operational units: an integer and a floating point unit. These operational units are supplied with instructions coming from the instruction queue, where fetched instructions are buffered each operational unit contains a set of functional units where instructions are executed, and a register file where results are written. Those hardware elements are not different from those of a classical scalar processor.

But other elements have been added to support superscalar techniques:

- **Multiple out-of-order issue**: In each operational unit, the decoder places instructions in program order in a central instruction window that decouples instruction decoding from instruction execution to perform dynamic scheduling. The instruction issue logic examines the instructions in the window, selects some of them for issue not necessarily in program order and dispatches them to their appropriate functional units. Any number of instructions is allowed to be in execution in the functional units as long as resource conflicts are resolved.
- **Multiple out-of-order completion:** Functional units have various latencies, and instruction issuing is not stalled when a functional unit takes more than one cycle to complete its operation. So instructions can complete out of program order. Hardware mechanisms must insure that results are written in correct order in registers: storage conflicts are resolved with register renaming, using a reorder buffer where results of the instructions are placed once computed. Other mechanisms play their part to accelerate instruction processing and thus enhance the above superscalar features.

- **A branch target buffer:** between the instruction fetcher and the decoder. This branch target buffer enables branch prediction to be performed by the instruction fetcher. It allows the processor to execute instructions past conditional branches; the reorder buffer is then used to recover from any mispredicted branch.
- **A load and a store buffer:** common to both operational units. They permit the load/store address calculations to be decoupled from memory accesses, and allow loads to bypass stores, provided there are no memory address conflicts. By giving loads a higher priority, the overall instruction processing is accelerated.

6.2.2 User Interface Description

Until now, we have described the core of superDLX_ the simulated hardware mechanisms and hardware elements. The user interface needs to be introduced also, as it represents a non-negligible part of the simulator. Although less complex than the simulation core, it will likely receive more attention from the users. The interface presented here is an on-line interface with a DLXsim: when the program is run, the simulator prompt appears and the user can enter interactive commands.

User's input is of two kinds:

- A number of simulation parameters: they are gathered in the optional machine Configuration file.
- A number of commands: they allow the user to interact with the simulator within a simulation session.

6.3 Processing User Commands

1. **Load Command:** This command invokes **Asm_LoadCmd** function in asm.c. It has been entirely reused from DLXsim. It loads assembly code and its data in specific sections of memory.
2. **Go Command:** the go command invokes the **Sim_GoCmd** of sim.c. This function has been partly reused from DLXsim.
3. **Step Command:** The step command invokes the **Step_SimCmd** of sim.c. This function has been partly reused.
4. **Next Command:**It invokes the **Sim_NextCmd** function of sim.c. This function calls Simulate with directives that depend on the command arguments, if the argument specifies that a certain number of instructions should be executed, Simulate is called to simulate as many cycles as necessary for the execution of the specified number of instructions, if the argument specifies a number of cycles to be executed, Simulate is called to simulate as many cycles as specified.
5. **Print Command:** This command is intended for the user to inspect the various buffers and queues of the superscalar simulator, during simulation

   It invokes the **Sim_InspectCmd** function of sim.c. This function calls other procedures of inspect that format and output the content of various data structures, on a given clock cycle:

   - The instruction queue.
   - The instruction windows.
   - The reorder buffers.
   - The store and the load buffer.
   - The number, latency and usage of each functional units.

6.3 Processing User Commands

1. **Load Command:** This command invokes **Asm_LoadCmd** function in asm.c. It has been entirely reused from DLXsim. It loads assembly code and its data in specific sections of memory.
2. **Go Command:** the go command invokes the **Sim_GoCmd** of sim.c. This function has been partly reused from DLXsim.
3. **Step Command:** The step command invokes the **Step_SimCmd** of sim.c. This function has been partly reused.
4. **Next Command:**It invokes the **Sim_NextCmd** function of sim.c. This function calls Simulate with directives that depend on the command arguments, if the argument specifies that a certain number of instructions should be executed, Simulate is called to simulate as many cycles as necessary for the execution of the specified number of instructions, if the argument specifies a number of cycles to be executed, Simulate is called to simulate as many cycles as specified.
5. **Print Command:** This command is intended for the user to inspect the various buffers and queues of the superscalar simulator, during simulation

   It invokes the **Sim_InspectCmd** function of sim.c. This function calls other procedures of inspect that format and output the content of various data structures, on a given clock cycle:

   - The instruction queue.
   - The instruction windows.
   - The reorder buffers.
   - The store and the load buffer.
   - The number, latency and usage of each functional units.

6. **Stats Command:** It invokes the **Sim_DumpStats** function that formats and displays the various statistics that are gathered during simulation Again, not all the statistics are necessarily presented, the choice is made based on the command arguments Statistics gathered during simulation are counters which are transformed for display into percentages distribution tables.
7. **Reset Command: Sim_ResetCmd** reinitializes all the elements of the processor. But user's parameters entered via the Machine Configuration file at the beginning of a simulation session are still valid.

6.4 Simulation Speed

The simulation run time varies depending on the simulation configuration parameters. The sizes of the buffers, instruction queue, reorder buffers, instruction windows, have a significant impact on the simulation time.

6.5 Conclusion and Future Work

As a result of this research we learn more about RISC, CISC and superscalar architecture, and the main goal of the research did not satisfy from the simulator as it was suggested but the final result was the introduction to the founded simulator "superDLX", in addition to identify another simulator "PCSpim" that did not help us more since its task to show the register organization variation after each execution to a determined assembly program.

The goal of this section is to give some ideas for future enhancement on the research.

- Make hardware design for the control unit.
- Research in detail for a simulator and run to display the performance for the control unit.
- Improving the simulator user interface and usability.

References

[1] "Computer Organization And Architecture", William Stallings Prentice Hall Introduction Edition, Designing For Performance 1996 by prentice Hall, Inc. Simon &Schuster/A Viacom Company Upper Saddle Revering 07458 Fourth Edition.

[2] "Advanced Computer Architecture Kai Hwang".

[3] Internet : http:// WWW.research .digital.com\' Er1.

[4] Vikas Agarwal, StephenW. Keckler, And Doug Burger. Scaling of microarchitectureal structures in future process technologies. Technical Report TR2000-02, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, February 2000.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalape.no JVM. In ACM SIGPLAN Conference on Object-Oriented Programming, System< Languages< and Applications (OOPSLA'00), October 2000.

[6] T. Bell and J. Larus. Ef_cient path pro_ling. In Proceedings of the 29th International Symposium on microarchitecture, December 1996.

[7] Jim Basney, Miron Livny, and Todd Tannenbaum. High throughput computing with condor. HPCU News, 1(2), June 1997.

[8] Doug Burger and Todd M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[9] A.D.Kulkarni. Artifical netural networks for Image understanding. Van Nostrand Reinhold, 1993.