NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

Moulding True Citizen
Since 1968

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COURSE MATERIALS

EC 206: COMPUTER ORGANISATION

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Electronics and Communication Engineering

  M.Tech in VLSI
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Provide well versed, communicative Electronics Engineers with skills in Communication systems with corporate and social relevance towards sustainable developments through quality education.

DEPARTMENT MISSION

1) Imparting Quality education by providing excellent teaching, learning environment.

2) Transforming and adopting students in this knowledgeable era, where the electronic gadgets (things) are getting obsolete in short span.

3) To initiate multi-disciplinary activities to students at earliest and apply in their respective fields of interest later.

4) Promoting leading edge Research & Development through collaboration with academia & industry.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1. To prepare students to excel in postgraduate programmes or to succeed in industry / technical profession through global, rigorous education and prepare the students to practice and innovate recent fields in the specified program/ industry environment.

PEO2. To provide students with a solid foundation in mathematical, Scientific and engineering fundamentals required to solve engineering problems and to have strong practical knowledge required to design and test the system.

PEO3. To train students with good scientific and engineering breadth so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

PEO4. To provide student with an academic environment aware of excellence, effective communication skills, leadership, multidisciplinary approach, written ethical codes and the life-long learning needed for a successful professional career.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Facility to apply the concepts of Electronics, Communications, Signal processing, VLSI, Control systems etc., in the design and implementation of engineering systems.

PSO2: Facility to solve complex Electronics and communication Engineering problems, using latest hardware and software tools, either independently or in team.optimization.

COURSE OUTCOMES
EC 206

SUBJECT CODE: EC 206	
COURSE OUTCOMES	
C206.1	Ability to illustrate the understanding of the functional units of a computer.
C206.2	Ability to Identify the different types of instructions.
C206.3	Ability to illustrate signal space representation of signal using Gram Schmidt orthonormalisation procedure.
C206.4	Ability to understand the various addressing modes.
C206.5	Ability to understand the I/O addressing system.
C206.6	Ability to Categorize the different types of memories.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C206.1			3									1
C206.2	2	3	3		2			2				1
C206.3	2	3	3	3	2	3	2				2	1
C206.4		3	3	3		3						1
C206.5			3		2							1
2066			3	3			2					1
C206	2	3	3	3	2	3	2	2			2	1

CO'S	PSO1	PSO2	PSO3
C206.1			
C206.2	3		
C206.3	3	3	2
C206.4	3	3	2
C206.5			
C206.6			
C206	3	3	2

SYLLABUS

Course code	Course Name	L-T-P - Credits	Year of Introduction
EC206	COMPUTER ORGANISATION	3-0-0-3	2016

Prerequisite: EC207 Logic Circuit Design

Course Objectives

- To impart knowledge in computer architecture.
- To impart knowledge in machine language programming.

  To develop understanding on I/O accessing techniques and memory structures.

Syllabus

Functional units of a computer, Arithmetic circuits, Processor architecture, Instructions and addressing modes, Execution of program, Micro architecture design process, Design of data path and control units, I/O accessing techniques, Memory concepts, Memory interface, Cache and Virtual memory concepts.

Expected outcome .

The students will be able to:

i. Understand the functional units of a computer
ii. Identify the different types of instructions
iii. Understand the various addressing modes
iv. Understand the I/O addressing system
v. Categorize the different types of memories

Text Books:

1. David A. Patterson and John L. Hennessey, Computer Organisation and Design, Fourth Edition, Morgan Kaufmann
2. David Money Harris, Sarah L Harris, Digital Design and Computer Architecture,M Kaufmann – Elsevier, 2009

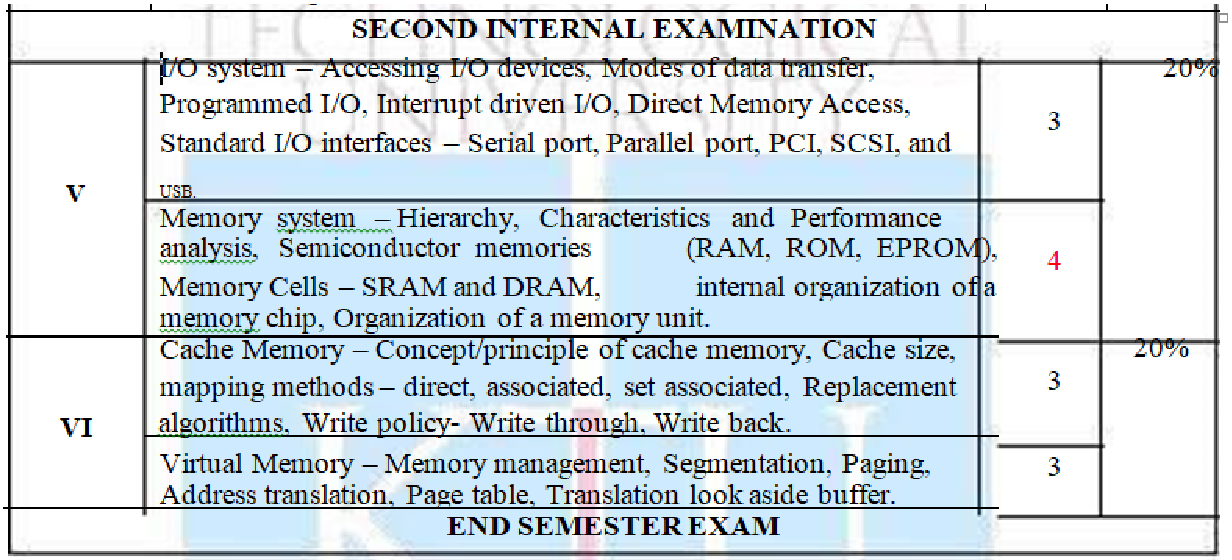
References

1. Carl Hamacher : "Computer Organization ", Fifth Edition, Mc Graw Hill
2. John P Hayes: "Computer Architecture and Organisation", Mc Graw Hill
3. William Stallings: "Computer Organisation and Architecture", Pearson Education
4. Andrew S Tanenbaum: "Structured Computer Organisation", Pearson Education
5. Craig Zacker: "PC Hardware : The Complete Reference", TMH

Course Plan

Module	Contents	Hours	Sem. Exam Marks
I	Functional units of a computer Arithmetic Circuits: Adder-carry propagate adder, Ripple carry adder, Basics of carry look ahead and prefix adder, Subtractor, Comparator, ALU	4	15%
	Shifters and rotators, Multiplication, Division	3	
	Number System: Review of Fixed point & Floating point number system	1	
II	Architecture : Assembly Language, Instructions, Operands, Registers, Register set, Memory, Constants	2	15%
	Machine Language: R-Type, I-Type, J-Type Instructions, Interpreting machine language code	3	
FIRST INTERNAL EXAMINATION			
III	MIPS Addressing modes – Register only, Immediate, Base, PC-relative, Pseudo - direct	3	15%

	MIPS memory map, Steps for executing a program - Compilation, Assembling, Linking, Loading	3	
	Pseudo instructions, Exceptions, Signed and Unsigned instructions, Floating point instructions	3	
IV	MIPS Microarchitectures – State elements of MIPS processor	1	15%
	Design process and performance analysis of Single cycle processor, Single cycle data path, Single cycle control for R – type arithmetic/logical instructions.	3	
	Design process and performance analysis of multi cycle processor, Multi cycle data path, Multi cycle control for R – type arithmetic/logical instructions.	3	



	SECOND INTERNAL EXAMINATION		
V	I/O system – Accessing I/O devices, Modes of data transfer, Programmed I/O, Interrupt driven I/O, Direct Memory Access, Standard I/O interfaces – Serial port, Parallel port, PCI, SCSI, and USB.	3	20%
	Memory system – Hierarchy, Characteristics and Performance analysis, Semiconductor memories (RAM, ROM, EPROM), Memory Cells – SRAM and DRAM, internal organization of a memory chip, Organization of a memory unit.	4	
VI	Cache Memory – Concept/principle of cache memory, Cache size, mapping methods – direct, associated, set associated, Replacement algorithms, Write policy- Write through, Write back.	3	20%
	Virtual Memory – Memory management, Segmentation, Paging, Address translation, Page table, Translation look aside buffer.	3	
	END SEMESTER EXAM		

Question Paper Pattern (End Sem Exam)

Maximum Marks: 100 **Time : 3 hours**

The question paper shall consist of three parts. Part A covers modules I and II, Part B covers modules III and IV, and Part C covers modules V and VI. Each part has three questions uniformly covering the two modules and each question can have maximum four subdivisions. In each part, any two questions are to be answered. Mark patterns are as per the syllabus with maximum80 % for theory and 20% for logical/numerical problems, derivation and proof.

QUESTION BANK

Module-1:

1. Perform the following shift operations on 8-bit binary representations of the decimal numbers and hence verify the shift rules (wherever applicable):
(a) 4 << 4 (b) 16 << 2 (c) -32 >>> 4 (d) 64 >> 4 (e) 64 O>> 2 (f) 32 <<O 5

2. Implement a two Operand *8-bit Equality Comparator* for the numbers A and B represented in 8-bit binary. State the rule for the operations.

3. Implement a 32-bit *Carrylookahead Adder*, starting from the 4-bit Adder. Clearly depict the logic diagrams with the supporting analyses. Compute the overall delay of the Adder.

4. From first principles, construct a *Carry Prefix Adder* with logic diagram and supporting analyses. Estimate the delay of the Adder.

5. Implement a *32-bit Adder* using *Ripple Carry Adder*. Explain operation. Estimate the total delay.

6. Construct an *8-bit Array Multiplier* to handle 8-bit Multiplicand and 8-bit Multiplier, with the aid of a supporting binary example of the multiplication process. Estimate the delay of the Multiplier.

7. Construct a *non-restoring Divider* as a Sequential circuit with the aid of a supporting example of division and the control flow process for the operation.

8. Construct a *Multiplier* as a Sequential circuit with the aid of a supporting example of multiplication and the control flow process for the operation.

9. Perform the implementation of 4-*bit Rotators* for ROL and ROR operations on an 4-bit operands A to yield Y, with proper use of the shamt bits.

10. Represent the two decimal numbers 7.875 and 0.1875 in the *IEEE 754 standard single precision floating point number format* and perform the Addition of the numbers with final representation of the result in the same format.

11. Perform the implementation of 4-*bit Shifters* for LSL, LSR and ASR operations on an 4-bit operands A to yield Y, with proper use of the shamt bits.

12. Implement a two Operand *4-bit Magnitude Comparator* for the numbers A and B represented in 4-bit binary. State the rules for the operation.

13. Examine the operation of a modern Subtractor with the aid of a sketch and hardware details.

14. Construct an eight-bit Equality Comparator from first principles and list the hardware requirements. Does it validate the three Y's?

15. Investigate the operation of a 4-bit Shifter with the aid of hardware representation diagram with wirings for left shift, logical right shift and arithmetic right shift. How the wiring would be tweaked to support Rotator?

16. Examine the Functional Units of a Computer with the aid of a sketch and hardware details.

17. Compare and Contrast RISC and CISC in the context of Computer Architectures, with the help of examples.

18. Investigate the hardware of the ALU Implementation with symbol, functional table and internal structure. Verify its adaptability in the context of modern computing.
19. Investigate the operation of an 8-bit Shifter with the aid of hardware representation diagram for left shift, logical right shift and arithmetic right shift. How does it support Rotation operation?
20. Give definitions of Computer Organization, Computer Architecture and Computer Hardware. State examples of Computers.
21. Perform the following operations using 8-bit binary representations and justify the shift rules: 16 << 2; -32>>>4;
22. Investigate the operation of the Carry Look Ahead Adder with the help of logic diagrams. Calculate the delay for this Adder and compare with that of Ripple Carry Adder.
23. Compare the delays of a 32-bit ripple carry adder and a 32-bi5 carry look ahead adder with 4-bit blocks. Assume each 2-i/p gate delay is 200ps and the FA delay is 400ps.

Module 1 and 2

1. With the help of a block diagram, describe the structure and functional operation of an Digital Computer.
2. Write short notes on:
   a. Differences between RAM and ROM
   b. Computer Hardware
   c. Computer Architecture
   d. Assembly language and its relevance in the context of Computer Architecture
   e. Logical Implementation of a Full Adder from Truth Table, K-Map Minimizations and diagram.
   f. Internal Registers of the Processor and their functions

3. Describe the principle and operation of Ripple Carry Adder with the aid of circuit diagram and proper design. Estimate the delay in this circuit.
4. Perform the following shift operations on 8-bit binary representations of the decimal numbers and hence verify the shift rules (wherever applicable):
   (b) 4 << 4 (b) 16<<2 (c) -32 >>> 4 (d) 64 >> 4 (e) 64 O>> 2 (f) 32 <<O 5

5. Describe the principle and operation of Carry Lookahead Adder with the aid of circuit diagram and proper design. Estimate the delay in this circuit.
6. Compare and contrast Ripple Carry Adder with Carry Lookahead Adder.
7. Describe the implementation of a 4-bit Multiplier using Array Multiplier with the aid of all appropriate diagrams and supporting analyses. State the hardware requirements for n-bit multiplication.

8. Describe the implementation of a 4-bit Multiplier using Sequential Circuit Multiplier with the aid of all appropriate diagrams and supporting analyses. State the hardware requirements for n-bit multiplication.

9. Describe the implementation of a 4-bit Divider using Sequential Circuit Divider with the aid of all appropriate diagrams and supporting analyses.

10. Explain the Shifters and Rotators by simple examples. Discuss the hardware requirements. Implement 4-bit Shifters for logical and arithmetic operations.

11. Provide the logical implementation of the ALU with the aid of supporting analyses, functional table and internal representation. Describe its operations. Suggest how Flags could be made available.

12. Explain the Comparator circuit used in the Processor with implementation for the different types. Choose 4-bit operands for your answer.

13. Describe how a Subtractor can be implemented in modern fast arithmetic circuits. Use appropriate logic diagrams and other representations.

14. Estimate the delay of a 64-bit carry prefix adder assuming that each 2-input Gate has a delay of 400 ps.

15. With the help of suitable examples, differentiate between the R-type and I-type Instructions in MIPS machine language.

16. Illustrate the IEEE standard for single precision and double precision floating point numbers.

17. Write short notes on:

a. MIPS Register Set.

b. Byte Addressable Memory.

c. Format of J-Instructions in MIPS machine language

18. Assume that opcode 'addi' is represented by 8_{10}, register 'add' function is represented by the function code 32_{10}, and register s0 to s7 are represented by 16_{10} to 23_{10}, in MIPS Machine language.

a. Translate the following machine language code into MIPS assembly language: 0x2237FFF3

b. Translate the following MIPS assembly code into MIPS machine code in hex format: add $s0, $s4, $s5

19. Describe the structure of the N-bit Non-restoring Divider as a sequential circuit with the help of appropriate logic diagram and illustrations with analyses.

20. Translate the following high-level code into assembly language. Assume variables a `to` `c` are held in registers `$s0 to $s2` and `f to j` are in `$s3 to $s7`.

```
a = b - c;
f = (g + h) - (i + j);
```

21. Describe the types of Digital Computers and the factors involved in comparing their performance.

22. Investigate how the arithmetic operations of Addition/Subtraction and Multiplication are performed on floating point operands within the Processor.

Module 3 and 4

23. Describe the Addressing Modes of the MIPS with the aid of examples of Instructions for each type.

24. What is meant by Microarchitecture? Explain the relevance with regards to MIPS architecture.

25. Explain the Performance Analysis of Computer systems.

26. Explain the MIPS Memory Map with the help of a diagram, stating lucidly the various sections or segments and their properties and functions.

27. With the aid of a diagram, describe the steps involved in translating and executing a high level language program.

28. Describe the Pseudo-instructions used in MIPS architecture with the help of four examples. Decompose the same into legitimate MIPS instructions.

29. Describe the concept of Exception Processing and its implementation in MIPS architecture.

30. Explain the signed and unsigned instructions of the MIPS for different categories of Instructions with the aid of lucid examples.

31. Describe the Floating Point Instructions of the MIPS architecture with the help of examples of instructions and their usage.

32. Compare and contrast the three microarchitectures used for MIPS architecture.

33. Derive the expression for Cycle Time in a Single Cycle MIPS processor. If the Cycle time for a single cycle MIPS processor is 1000 pS, calculate the total execution time for a program with 10 lakh instructions.

34. List the main drawbacks of Single Cycle Microarchitecture. How are they eliminated in Multi Cycle Microarchitecture?

35. With the help of suitable examples, differentiate between the R-type and I-type Instructions in MIPS machine language.

36. Illustrate the IEEE standard for single precision and double precision floating point numbers.

37. Write short notes on:

d. MIPS Register Set.

e. Byte Addressable Memory.

f. Format of J-Instructions in MIPS machine language

38. Assume that opcode ‘addi’ is represented by 8_{10}, register ‘add’ function is represented by the function code 32_{10}, and register s0 to s7 are represented by 16_{10} to 23_{10}, in MIPS Machine language.

c. Translate the following machine language code into MIPS assembly language: 0x2237FFF3

d. Translate the following MIPS assembly code into MIPS machine code in hex format: add $s0, $s4, $s5

39. Describe the organization of the Datapath of a Single Cycle Microarchitecture for an `lw` instruction. Support your answer with the aid of diagrams of the interconnections between the State Elements.

40. Describe the organization of the enhanced or extended Datapath of a Single Cycle Microarchitecture for inclusion of `sw` instruction, `R-type` instructions and `beq` instructions. Support your answer with the aid of diagrams of the interconnections between the State Elements.

41. Describe the organization of the Control unit of a Single Cycle Microarchitecture for an `lw` instruction and extend the same for the other categories of Instructions.

42. Describe the organization of the Datapath of a Multi Cycle Microarchitecture for an `lw` instruction and other categories of Instructions. Support your answer with the aid of diagrams of the interconnections between the State Elements.

43. Describe the organization of the Control Unit of a Multi Cycle Microarchitecture for an `lw` instruction and other categories of Instructions. Support your answer with the aid of diagrams of the interconnections between the State Elements.

44. Derive the expression for CPI (M) and Cycle Time in a Multi Cycle MIPS processor. If the Cycle time for a multi cycle MIPS processor is 1000 pS, calculate the total execution time for a program with 10 lakh instructions. Compare and contrast this result with that of Single Cycle Microarchitecture.

MODULE 5

1. State the various types of Input and Output devices that need to be used in a modern General purpose or Embedded Computer, and provide details how I/O Capability is provided?

2. Explain how accessing I/O devices is made possible in modern Computer, with the aid of a generic sketch.

3. Explain the relevance of Memory Mapped I/O.

4. State the need for I/O Device Interface and illustrate with the aid of a diagram the connections between CPU and I/O devices.

5. State the modes of I/O Data Transfer.

6. Explain Programmed controlled I/O mechanism.

7. Explain Interrupt Driven I/O mechanism with the aid of a diagram.

8. What is DMA? How does it use Interrupts?

9. Explain the DMA Controller Registers with the aid of diagrams.

10. Describe the DMA transfer with the aid of diagram and give details of the role of DMA Controller.

11. What is a Port and what are the types? Explain.

12. Explain functions of I/O Interface.

13. What is the need for a standard for I/O device connections?

14. Describe the objectives of USB and provide technical details of the standard.

15. Describe PCI Bus with the aid of diagram and give its advantages and benefits.

16. Investigate the SCSI Bus and how its serves the purpose of efficient data transfer, with details of a typical read operation.

17. Investigate Memory Hierarchy with the aid of a diagram and provide details of the different layers.

18. Explain the characteristics of Memory.

19. Describe Static Memories with the aid of diagrams.

20. Describe Dynamic RAM with the aid of a diagram.

21. Provide the internal organization of a Dynamic Memory Chip with a diagram.

22. Compare and Contrast Asynchronous and Synchronous DRAMs.

23. Explain the relevance of SIMMs and DIMMs in modern Computer Systems.

24. Compare and Contrast RAM and ROM by providing their inherent characteristics and differences.

25. With the aid of a diagram, explain the ROM Cell.

26. Explain PROM, EPROM and EEPROM.

27. Describe Flash memory and Flash Cards.

28. Compare and Contrast Flash Drives and Hard Disk Drives.

MODULE 6

1. State and explain the Principle of Locality.
2. State and explain the Locality of Reference.
3. Describe the Role of Cache memory w.r.t Memory Hierarchy.
4. With a diagram explain the Cache Memory Organization.
5. Define Cache Hit and Miss.
6. Explain the Cache Read process.
7. Compare the Cache Write- Write Through and Write Back a.k.a Copy back mechanism.
8. Define Cache Hit Rate ,Miss Rate, Ave Mem Access Time (AMAT)
9. Perform Cache Hit Rate ,Miss Rate, AMAT Calculations (Refer worked out questions in notes)
10. What is meant by Mapping and Replacement?
11. Perform an Analysis of Direct Mapped Cache or Direct Mapping with the aid of diagram.
12. Perform an Analysis of Associative Cache Mapping with the aid of diagram.
13. Perform an Analysis of Set Associative Cache Mapping with the aid of diagram .
14. Explain the use of Write Buffer.
15. Perform an Analysis of Replacement Algorithms.
16. Compare the complexity of implementation of the replacement algorithms.
17. Explain the need for Virtual Memory in Computers.
18. Analyze the Virtual Memory Organization with the help of a diagram.
19. Explain the Virtual Address or Logical Address.
20. Describe the Memory Management Unit (MMU) with a diagram.
21. Describe the Address Translation mechanism.
22. Explain the Virtual Memory Address Translation

23. Describe Paging with diagram.

24. Describe Translational Look aside Buffer (TLB) with diagram.

25. Explain Page Faults.

26. Describe the Segmentation with the help of a diagram.

27. Compare and contrast Paging and Segmentation.

Introduction to Computer Organization

Computer Organization is concerned with the function and design of the various units or sections of Digital Computers, that store and process information, receive information from external sources and send computed results to external destinations.

Computer Architecture encompasses the specification of an Instruction Set and the hardware units that implement Instructions.

Computer Hardware consists of Electronic circuits, Displays, Electronic, magnetic and optical storage media, Electromechanical equipment and Communication facilities.

Computer Software is concerned with the System Software or Operating System that manages the Hardware as well as the Application Software as well as Programming languages and Utilities.

Digital Computer is a fast Electronic Calculating Machine that accepts digitize input information, processes it according to a list of internally stored Instructions, and produces resulting output information. The list of instructions is called computer program and the internal storage is computer memory.

Computer Types

Since their introduction in the 1940s, digital computers have evolved into many different types that vary widely in size, cost, computational power, and intended use. Modern computers can be divided roughly into four general categories:

- *Embedded computers* are integrated into a larger device or system in order to automatically monitor and control a physical process or environment. They are used for a specific purpose rather than for general processing tasks. Typical applications include industrial and home automation, appliances, telecommunication products, and vehicles. Users may not even be aware of the role that computers play in such systems.

- *Personal computers* have achieved widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use. They support a variety of applications such as general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing. A number of classifications are used for personal computers.

Desktop computers serve general needs and fit within a typical personal workspace.

Workstation computers offer higher computational capacity and more powerful graphical display capabilities for engineering and scientific work.

Finally, *Portable* and *Notebook computers* provide the basic features of a personal computer in a smaller lightweight package. They can operate on batteries to provide mobility.

• *Servers* and *Enterprise systems* are large computers that are meant to be shared by a potentially large number of users who access them from some form of personal computer over a public or private network. Such computers may host large databases and provide information processing for a government agency or a commercial organization.

• *Supercomputers* and *Grid computers* normally offer the highest performance. They are the most expensive and physically the largest category of computers. Supercomputers are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work. They have a high cost. Grid computers provide a more cost-effective alternative. They combine a large number of personal computers and disk storage units in a physically distributed high-speed network, called a grid, which is managed as a coordinated computing resource. By evenly distributing the computational workload across the grid, it is possible to achieve high performance on large applications ranging from numerical computation to information searching.

There is an emerging trend in access to computing facilities, known as *cloud computing*. Personal computer users access widely distributed computing and storage server resources for individual, independent, computing needs. The Internet provides the necessary communication facility. Cloud hardware and software service providers operate as a utility, charging on a pay-as-you-use basis.

FUNCTIONAL UNITS OF A COMPUTER

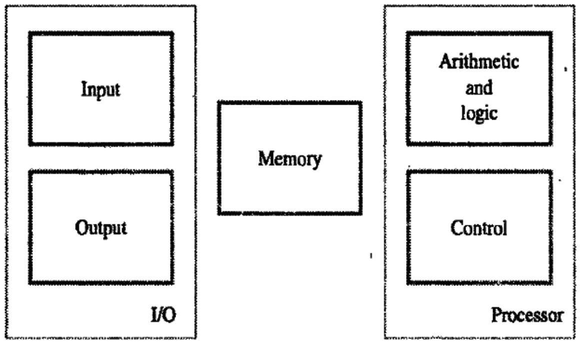


Figure 1.1 Basic functional units of a computer.

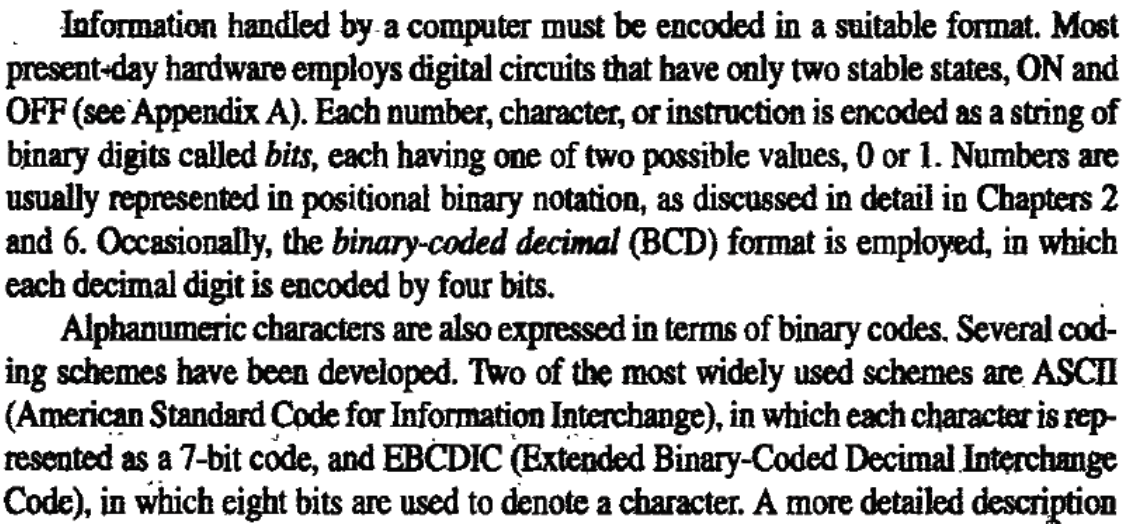
A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1. The input unit accepts coded information from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines. The information received is either stored in the computer's memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations. The processing steps are determined by a program stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. Figure 1.1 does not show the connections among the functional units. These connections, which can be made in several ways, are discussed throughout this book. We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the *processor,* and input and output equipment is often collectively referred to as the *input-output* (I/O) unit.

We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. *Instructions,* or *machine instructions,* are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed

A list of instructions that performs a task is called a *program.* Usually the program is stored in the memory. The processor then fetches the instructions that make up the program from the memory, one after another, and performs the desired operations. The computer is completely controlled by the *stored program,* except for possible external interruption by an operator or by I/O devices connected to the machine.

Data are numbers and encoded characters that are used as operands by the instructions. The term data, however, is often used to mean any digital information. Within this definition of data, an entire program (that is, a list of instructions) may be considered as data if it is to be processed by another program. An example of this is the task of *compiling* a high-level language *source program* into a list of machine instructions constituting a machine language program, called the *object program.* The source program is the input data to the *compiler* program which translates the source program into a machine language program.



Information handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states, ON and OFF (see Appendix A). Each number, character, or instruction is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1. Numbers are usually represented in positional binary notation, as discussed in detail in Chapters 2 and 6. Occasionally, the *binary-coded decimal* (BCD) format is employed, in which each decimal digit is encoded by four bits.

Alphanumeric characters are also expressed in terms of binary codes. Several coding schemes have been developed. Two of the most widely used schemes are ASCII (American Standard Code for Information Interchange), in which each character is represented as a 7-bit code, and EBCDIC (Extended Binary-Coded Decimal Interchange Code), in which eight bits are used to denote a character. A more detailed description

Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input. Digital communication facilities, via the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called *main memory*, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called *words*. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the *word length* of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct *address* is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a *random-access memory* (RAM). The time required to access one word is called the *memory access time*. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a *cache*, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent *secondary storage* is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including *magnetic disks*, *optical disks* (DVD and CD), and *flash memory devices*.

Arithmetic and Logic Unit

Most computer operations are executed in the *arithmetic and logic unit* (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elements called *registers*. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a *printer*. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name *input/output* (I/O) unit in many cases.

Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the *timing signals* that govern the transfers and determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.

- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.

- Processed information leaves the computer through an output unit.

- All activities in the computer are directed by the control unit.

ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division.

The Three-Y's

Designers use the three "-y's" to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

Hierarchy involves dividing a system into modules, then further sub-dividing each of these modules until the pieces are easy to understand.

Modularity states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.

Regularity seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

These building blocks are not only useful in their own right, but they also demonstrate the principles of Hierarchy, Modularity, and Regularity. The building blocks are hierarchically assembled from simpler components such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes.

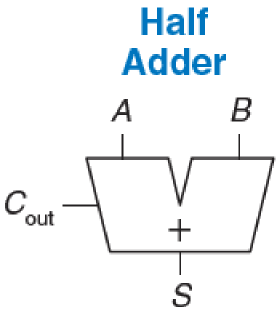
Addition

Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to N-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

Half Adder

We begin by building a 1-bit half Adder. As shown, the half Adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half Adder can be built from an XOR gate and an AND gate.

In a multi-bit Adder, C_{out} is added or carried in to the next most significant bit. For example, the carry bit is the output C_{out} of the first column of 1-bit addition and the input C_{in} to the second column of addition. However, the half Adder lacks a C_{in} input to accept C_{out} of the previous column.



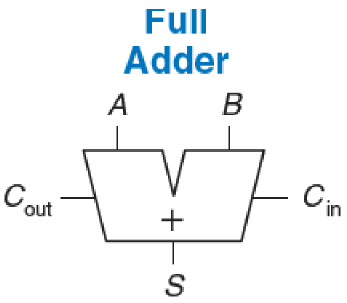
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

$$\begin{array}{r} \scriptstyle 1 \\ 0001 \\ +0101 \\ \hline 0110 \end{array}$$

Full Adder

A full Adder accepts the carry in C_{in} as shown. The figure also shows the output equations for S and C_{out}.



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

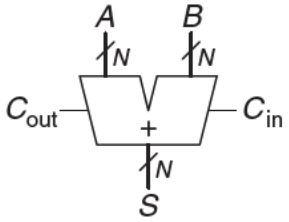
$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Carry Propagate Adder

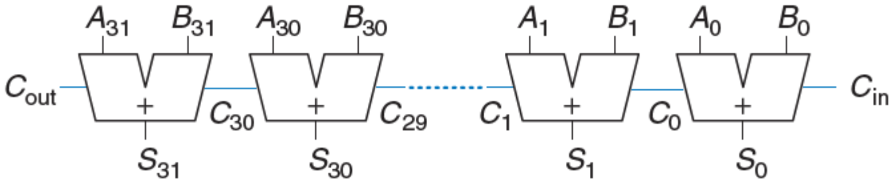
An N-bit Adder sums two N-bit inputs, A and B, and a carry in Cin to produce an N-bit result S and a carry out Cout. It is commonly called a carry propagate Adder (CPA) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full Adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry Adders, carry-look ahead Adders, and Prefix Adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate Adder is to chain together N full Adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown for 32-bit addition. This is called a ripple carry Adder. It is a good application of modularity and regularity: the full Adder module is reused many times to form a larger system. The ripple carry Adder has the disadvantage of being slow when N is large. S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}, as shown. We say that the carry ripples through the carry chain. The delay of the Adder, t_{ripple}, grows directly with the number of bits, where t_{FA} is the delay of a full Adder.



Carry propagate adder



32-bit ripple-carry adder

$$t_{\text{ripple}} = N t_{FA}$$

Carry-Look Ahead Adder

The fundamental reason that large ripple-carry Adders are slow is that the carry signals must propagate through every bit in the Adder. A carry look ahead Adder (CLA) is another type of carry propagate Adder that solves this problem by dividing the Adder into blocks and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to look ahead across the blocks rather than waiting to ripple through all the full Adders inside a block. For example, a 32-bit Adder may be divided into eight 4-bit blocks.

CLAs use generates (G) and propagate (P) signals that describe how a column or block determines the carry out. The ith column of an Adder is said to generate a carry if it produces a carry out independent of the carry in. The ith column of an Adder is guaranteed to generate a carry C_i if A_i and B_i are both 1. Hence G_i, the generate signal for column i, is calculated as $G_i = A_i.B_i$. The column is said to propagate a carry if it produces a carry out whenever there is a carry in. The ith column will propagate a carry in, C_{i-1}, if either A_i or B_i is 1. Thus, $P_i = A_i + B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the Adder. The ith column of an Adder will generate a carry out C_i if it either generates a carry, G_i, or propagates a carry in, $P_i.C_{i-1}$. In equation form,

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j.

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is :

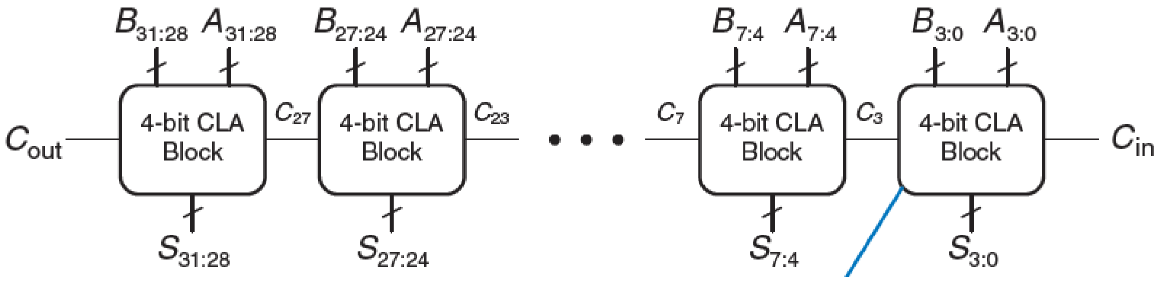
$$G_{3:0} = G_3 + P_3\big(G_2 + P_2(G_1 + P_1G_0)\big)$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is:

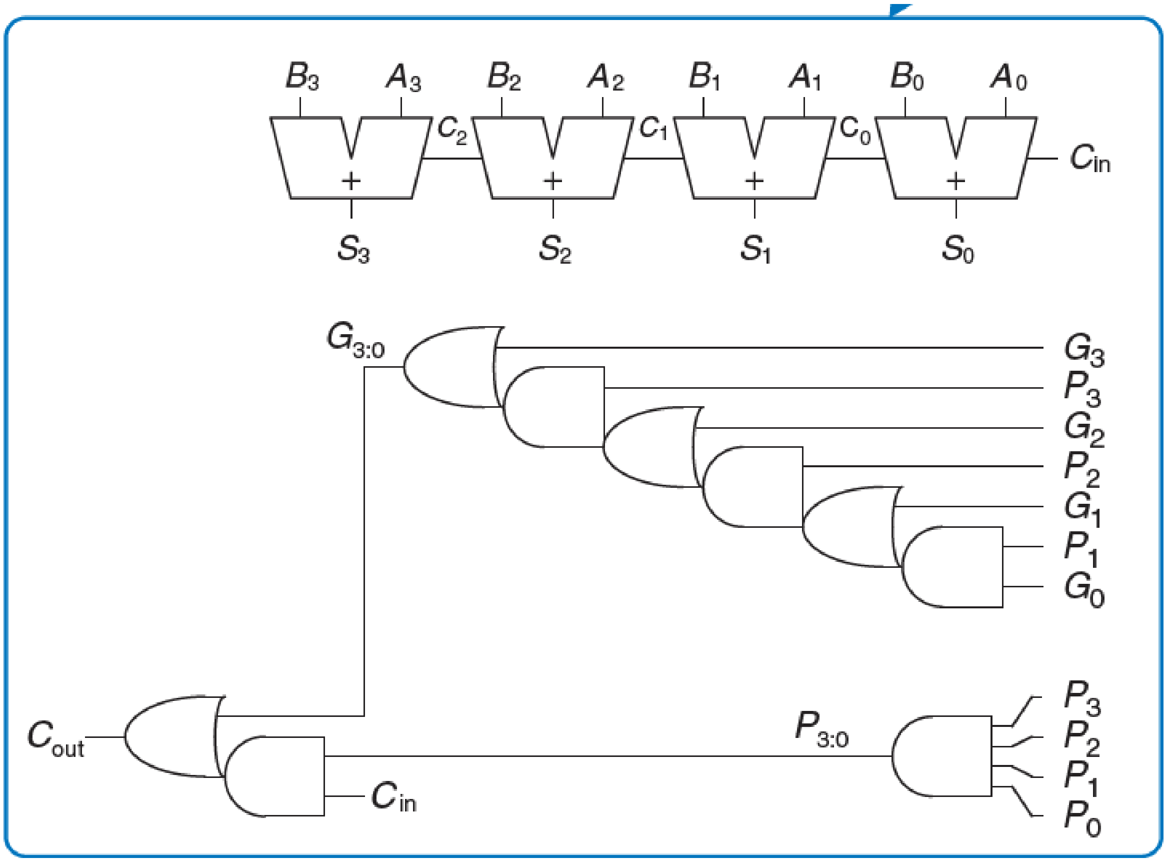
$$P_{3:0} = P_3P_2P_1P_0$$

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i, using the carry in to the block, C_j.

$$C_i = G_{i:j} + P_{i:j}C_j$$



A 32-bit carry-look ahead Adder is composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry Adder and some look ahead logic to compute the carry out of the block given the carry in, as shown.



The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i, from A_i and B_i are left out for brevity. Again, the carry-look ahead Adder demonstrates modularity and regularity.

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block.

Cin then advances directly to C_{out} through the AND/OR gate in each block until the last.

For a large Adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the Adder. Finally, the critical path through the last block contains a short ripple-carry Adder. Thus, an N-bit Adder divided into k-bit blocks has a delay:

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + k t_{FA}$$

where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate Pi and Gi, t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and t_{AND_OR} is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block. For N >16, the carry-look ahead Adder is generally much faster than the ripple-carry Adder. However, the Adder delay still increases linearly with N.

Example 1: RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry Adder and a 32-bit carry-look ahead Adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full Adder delay is 300 ps.

Sol:

The propagation delay of the 32-bit ripple carry Adder is 32 × 300 ps = 9.6 ns.

The CLA has t_{pg} = 100 ps,

t_{pg_block} = 6 × 100 ps = 600 ps, and

t_{AND_OR} = 2 × 100 ps =200 ps.

The propagation delay of the 32-bit carry-look ahead Adder with 4-bit blocks is :

100 ps+ 600 ps + (32/4 − 1) × 200 ps + (4 × 300 ps)

= 3.3 ns,

Almost three times faster than the ripple-carry Adder!

Prefix Adder

Early computers used ripple carry Adders, because components were expensive and ripple-carry Adders used the least hardware. Virtually all modern PCs use Prefix Adders on critical paths, because transistors are now cheap and speed is of great importance.

Prefix Adders extend the generate and propagate logic of the carry look ahead Adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a Prefix Adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

Define column i = −1 to hold C_{in}, so $G_{-1} = C_{in}$ and $P_{-1} = 0$. Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column i−1 if the block spanning columns i−1 through −1 generates a carry. The generated carry is either generated in column i−1 or generated in a previous column and propagated.

Thus, we rewrite Equation as:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

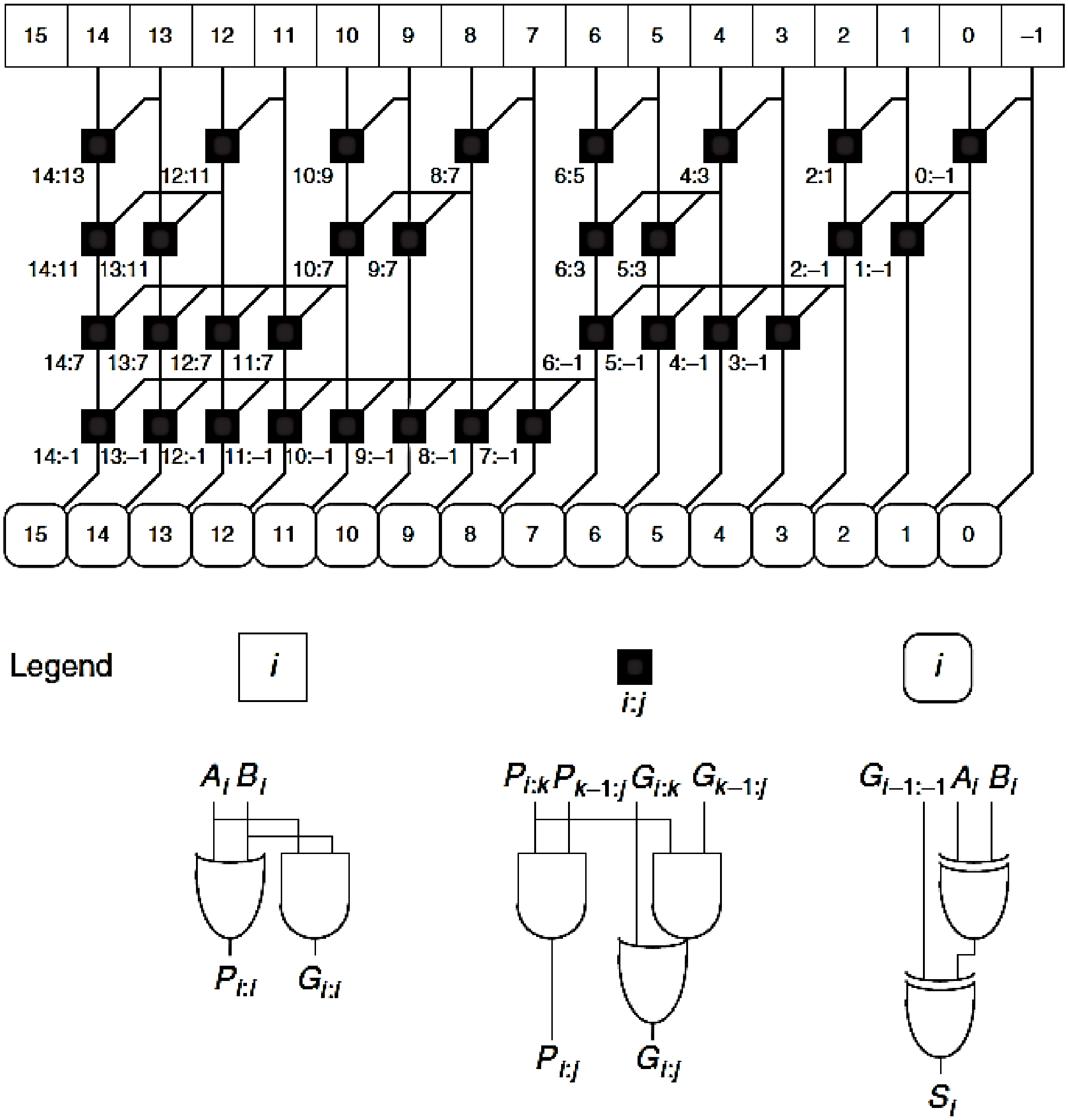
Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, . . . , $G_{N-2:-1}$. These signals, along with $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, . . . , $P_{N-2:-1}$, are called Prefixes.

The diagram shows an N= 16-bit Prefix Adder. The Adder begins with a pre-computation to form P_i and G_i for each column from A_i and B_i using AND and OR gates.

It then uses $\log_2 N = 4$ levels of black cells to form the Prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits i:k and from the lower part spanning bits k−1:j. It combines these parts to form generate and propagate signals for the entire block spanning bits i: j using the equations:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$



A block spanning bits i:j will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the upper and lower parts propagate the carry. Finally, the Prefix Adder computes the sums.

In summary, the Prefix Adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the Adder. This speedup is significant, especially for Adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-look ahead Adder.

The network of black cells is called a Prefix Tree. The general principle of using Prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique.

The critical path for an N-bit Prefix Adder involves the pre-computation of P_i and G_i followed by $\log_2 N$ stages of black Prefix cells to obtain all the Prefixes.

$G_{i-1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i.

The delay of an N-bit Prefix Adder is:

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$

where t_{pg_Prefix} is the delay of a black Prefix cell.

Example-2 **PREFIX ADDER DELAY**

Compute the delay of a 32-bit Prefix Adder. Assume that each two-input gate delay is 100 ps.

Sol:

The propagation delay of each black Prefix cell t_{pg_Prefix} is 200 ps (i.e., two gate delays).

Thus, the propagation delay of the 32-bit Prefix Adder is:

100 ps + $\log_2(32)$ × 200 ps + 100 ps = 1.2 ns,

which is about three times faster than the carry-look ahead Adder and eight times faster than the ripple-carry Adder from Example 1.

In practice, the benefits are not quite this great, but Prefix Adders are still much faster than the others.

Putting It All Together …

This section introduced the half Adder, full Adder, and three types of carry propagate Adders: ripple-carry, carry-look ahead, and Prefix Adders. Faster Adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate Adder for a design.

Subtraction

Adders can add positive and negative numbers using two’s complement number representation. Subtraction is almost as easy: flip the sign of the second number, then add. Flipping the sign of a two’s complement number is done by inverting the bits and adding 1.

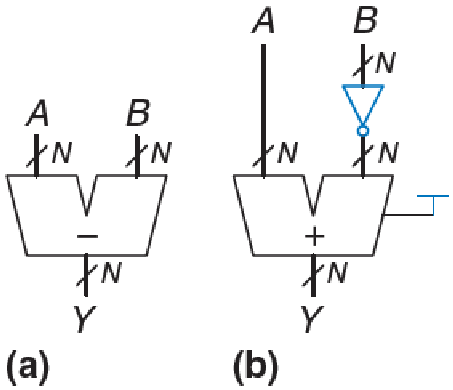
To compute Y = A − B, first create the two’s complement of B:

Invert the bits of B to obtain B and add 1 to get −B = B + 1.

Add this quantity to A to get Y = A + B + 1 = A − B.

This sum can be performed with a single CPA by adding A + B with C_{in} = 1.

The symbol for a Subtractor and the underlying hardware for performing Y = A – B are shown.



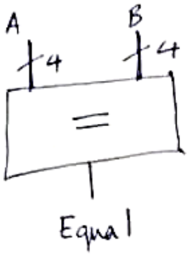
Comparators :-

A Comparator determines whether two binary numbers are equal or if one is greater or less than the other.

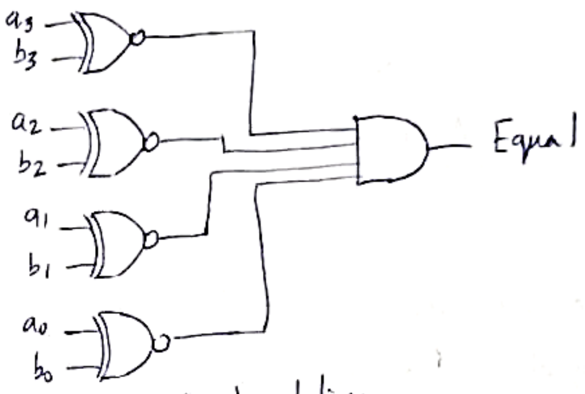
A Comparatre receives two N-bit binary numbers, A and B.

There are two common types of Comparators.

Equality Comparator :-



(a) Symbol
4-bit equality Comparator

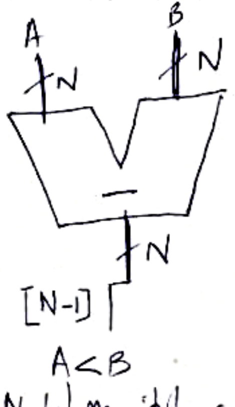


(b) Implementation

An equality comparator produces a single output to indicate whether A is equal to B (A==B). The implementation first checks to find if corresponding bits of A and B are equal, using XNOR gates. The numbers are equal, if all of the corresponding bits are equal.

Magnitude Comparator:

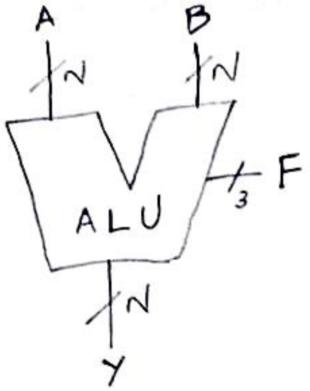
A magnitude comparator produces one or more outputs to indicate the relative values of A and B.



N-bit magnitude comparator

Magnitude comparison is done by Computing A-B and looking at the sign (MSB) bit of result as shown. If the result is negative (i.e, sign bit is 1), then A is less than B. Else A is greater than or equal to B.

Arithmetic Logic Unit (ALU) :-

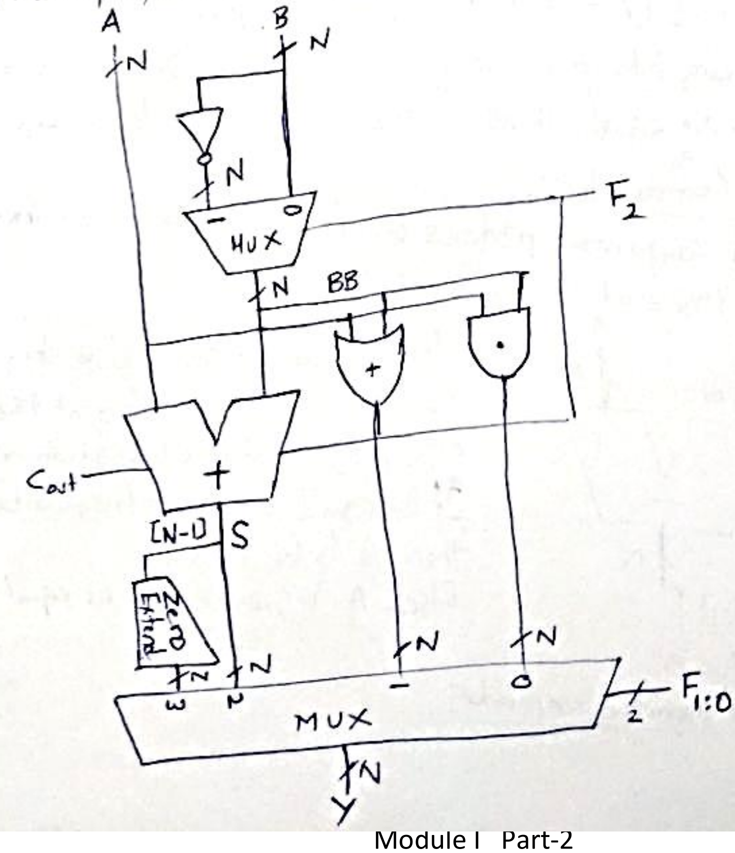


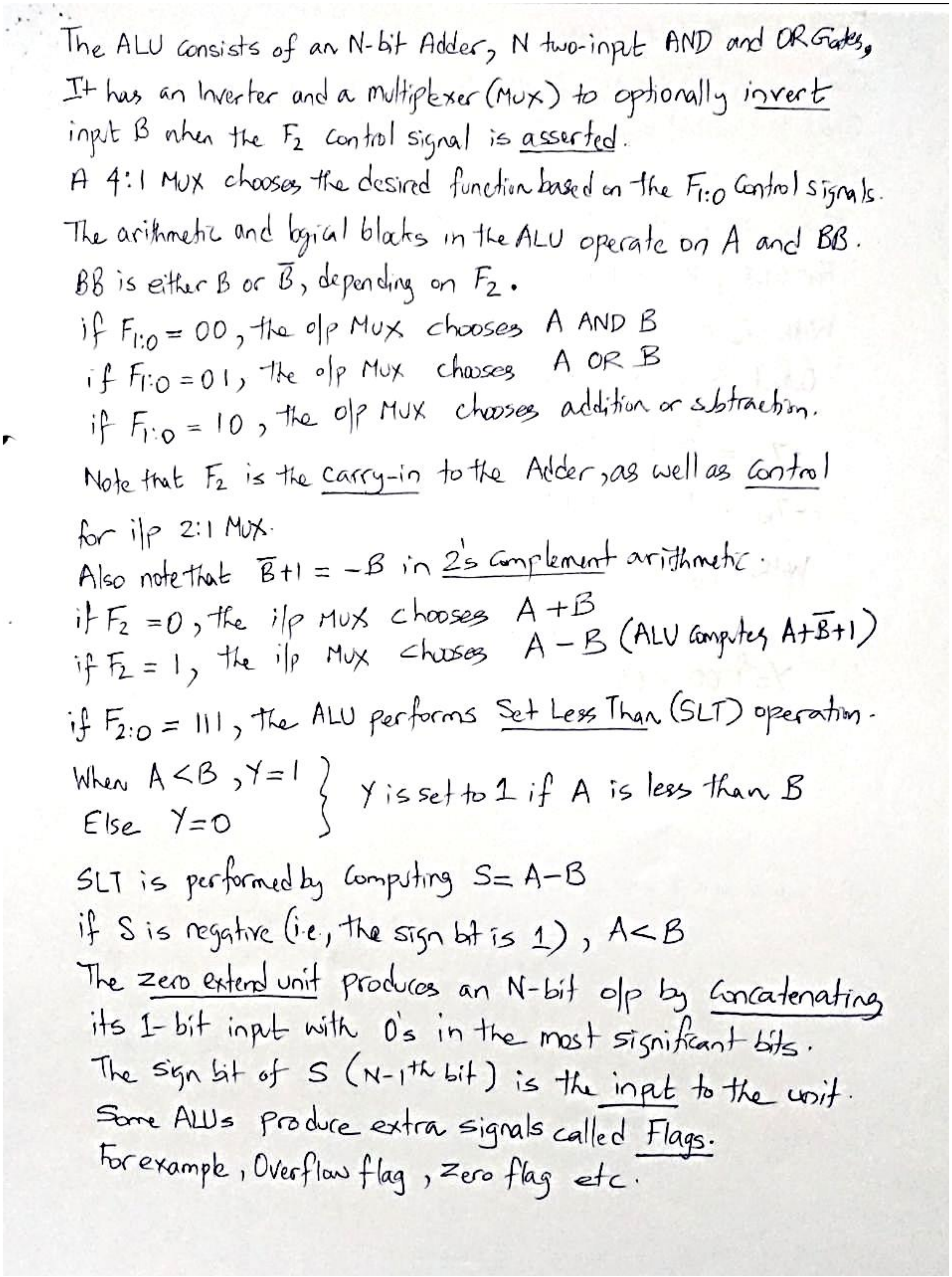
ALU Symbol

ALU Operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	Not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A − B
111	SLT

An Arithmetic Logic Unit (ALU) combines a variety of mathematical and logical operations into a single unit and forms the heart of the Computer System. The ALU accepts N-bit inputs and outputs. There is a Control Signal(s) F, that specifies which function to perform.





The ALU consists of an N-bit Adder, N two-input AND and OR Gates,
It has an Inverter and a multiplexer (MUX) to optionally invert input B when the F_2 control signal is asserted.

A 4:1 MUX chooses the desired function based on the $F_{1:0}$ Control signals.

The arithmetic and logical blocks in the ALU operate on A and BB.
BB is either B or $\overline{B}$, depending on F_2.

if $F_{1:0} = 00$, the o/p MUX chooses A AND B

if $F_{1:0} = 01$, the o/p MUX chooses A OR B

if $F_{1:0} = 10$, the o/p MUX chooses addition or subtraction.

Note that F_2 is the carry-in to the Adder, as well as control for i/p 2:1 MUX.

Also note that $\overline{B}+1 = -B$ in 2's complement arithmetic.

if $F_2 = 0$, the i/p MUX chooses $A+B$

if $F_2 = 1$, the i/p MUX chooses $A-B$ (ALU computes $A+\overline{B}+1$)

if $F_{2:0} = 111$, the ALU performs Set Less Than (SLT) operation.

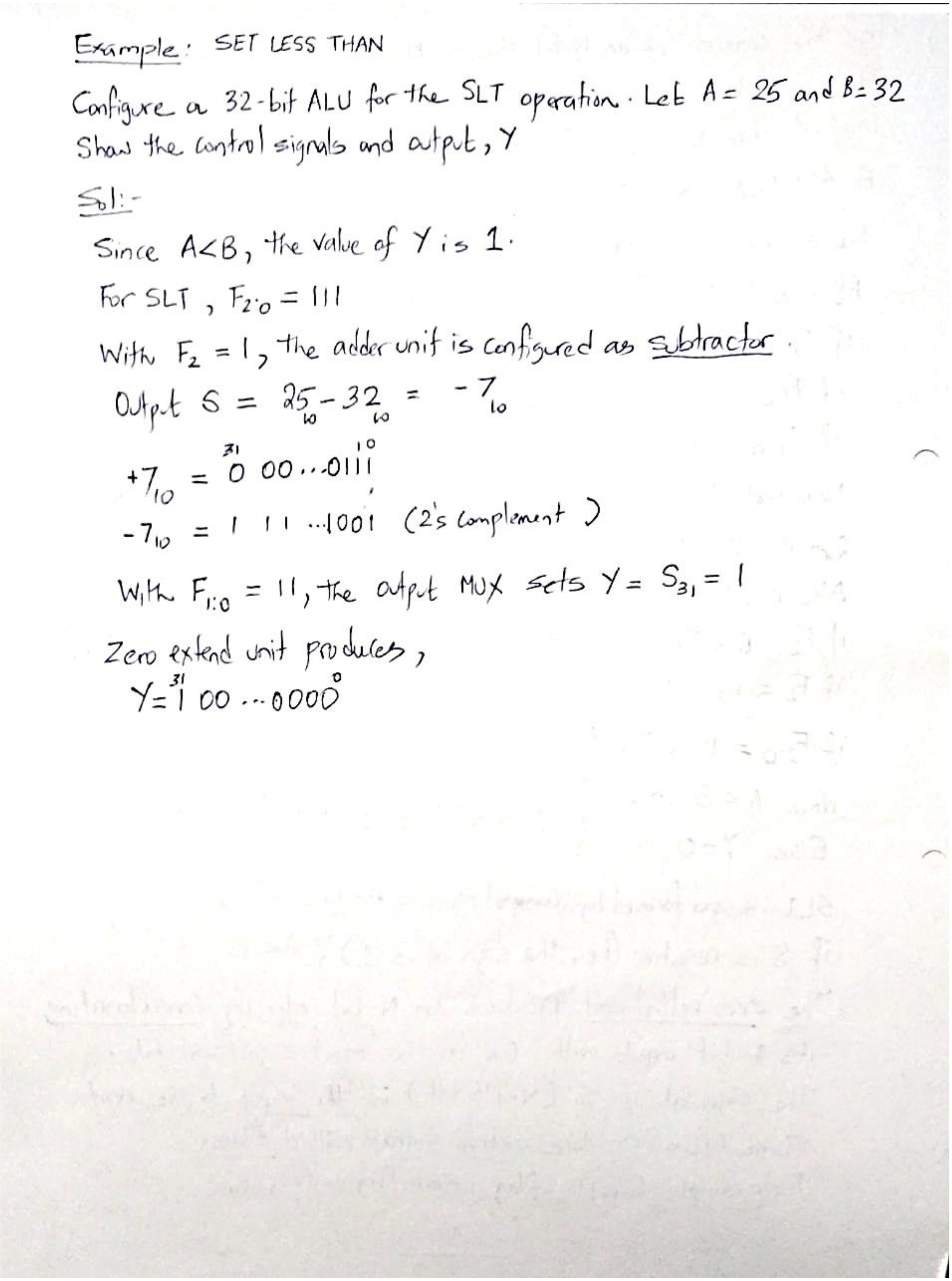
When $A<B$, $Y=1$ } Y is set to 1 if A is less than B
Else $Y=0$ }

SLT is performed by computing $S = A-B$

if S is negative (i.e, the sign bit is 1), $A<B$

The zero extend unit produces an N-bit o/p by concatenating its 1-bit input with 0's in the most significant bits.
The sign bit of S ($N-1^{th}$ bit) is the input to the unit.

Some ALUs produce extra signals called Flags.
For example, Overflow flag, Zero flag etc.



Example: SET LESS THAN

Configure a 32-bit ALU for the SLT operation. Let A = 25 and B = 32

Show the control signals and output, Y

Sol:-

Since A<B, the value of Y is 1.

For SLT, $F_{2:0} = 111$

With $F_2 = 1$, the adder unit is configured as subtractor.

Output $S = 25_{10} - 32_{10} = -7_{10}$

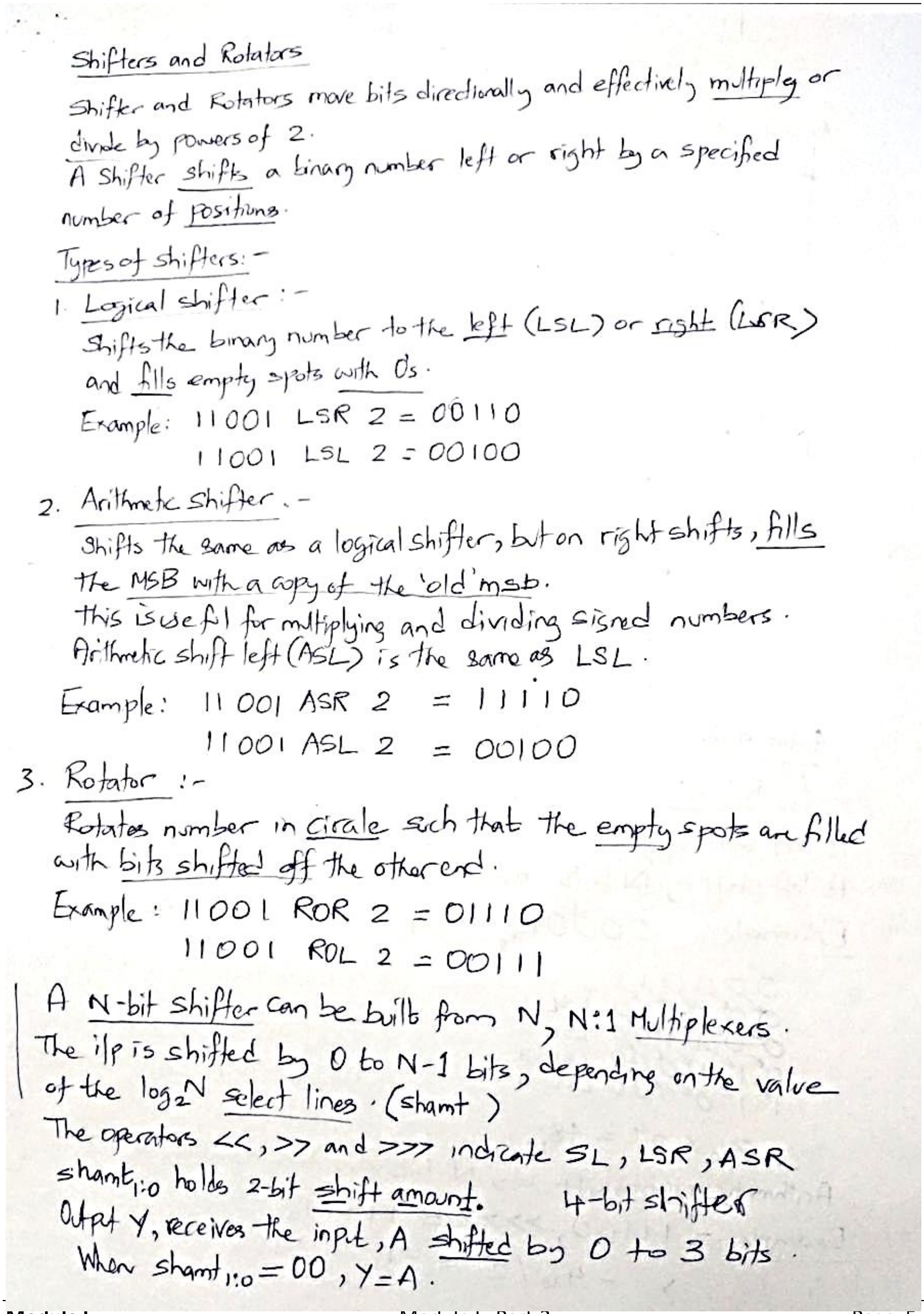
$+7_{10} = \overset{31}{0}\ 00\ldots 01\overset{1}{1}\overset{0}{1}$

$-7_{10} = 1\ 11\ldots 1001$ (2's complement)

With $F_{1:0} = 11$, the output MUX sets $Y = S_{31} = 1$

Zero extend unit produces,

$Y = \overset{31}{1}\ 00\ldots 000\overset{0}{0}$



Shifters and Rotators

Shifter and Rotators move bits directionally and effectively multiply or divide by powers of 2.

A Shifter shifts a binary number left or right by a specified number of positions.

Types of shifters:-

1. Logical shifter :-

   Shifts the binary number to the left (LSL) or right (LSR) and fills empty spots with 0s.

   Example: 11001 LSR 2 = 00110

   11001 LSL 2 = 00100

2. Arithmetic Shifter .-

   Shifts the same as a logical shifter, but on right shifts, fills the MSB with a copy of the 'old' msb.

   this is usefil for multiplying and dividing signed numbers.

   Arithmetic shift left (ASL) is the same as LSL.

   Example: 11 001 ASR 2 = 11110

   11001 ASL 2 = 00100

3. Rotator :-

   Rotates number in circle such that the empty spots are filled with bits shifted off the other end.

   Example: 11001 ROR 2 = 01110

   11001 ROL 2 = 00111

A N-bit shifter can be built from N, N:1 Multiplexers.

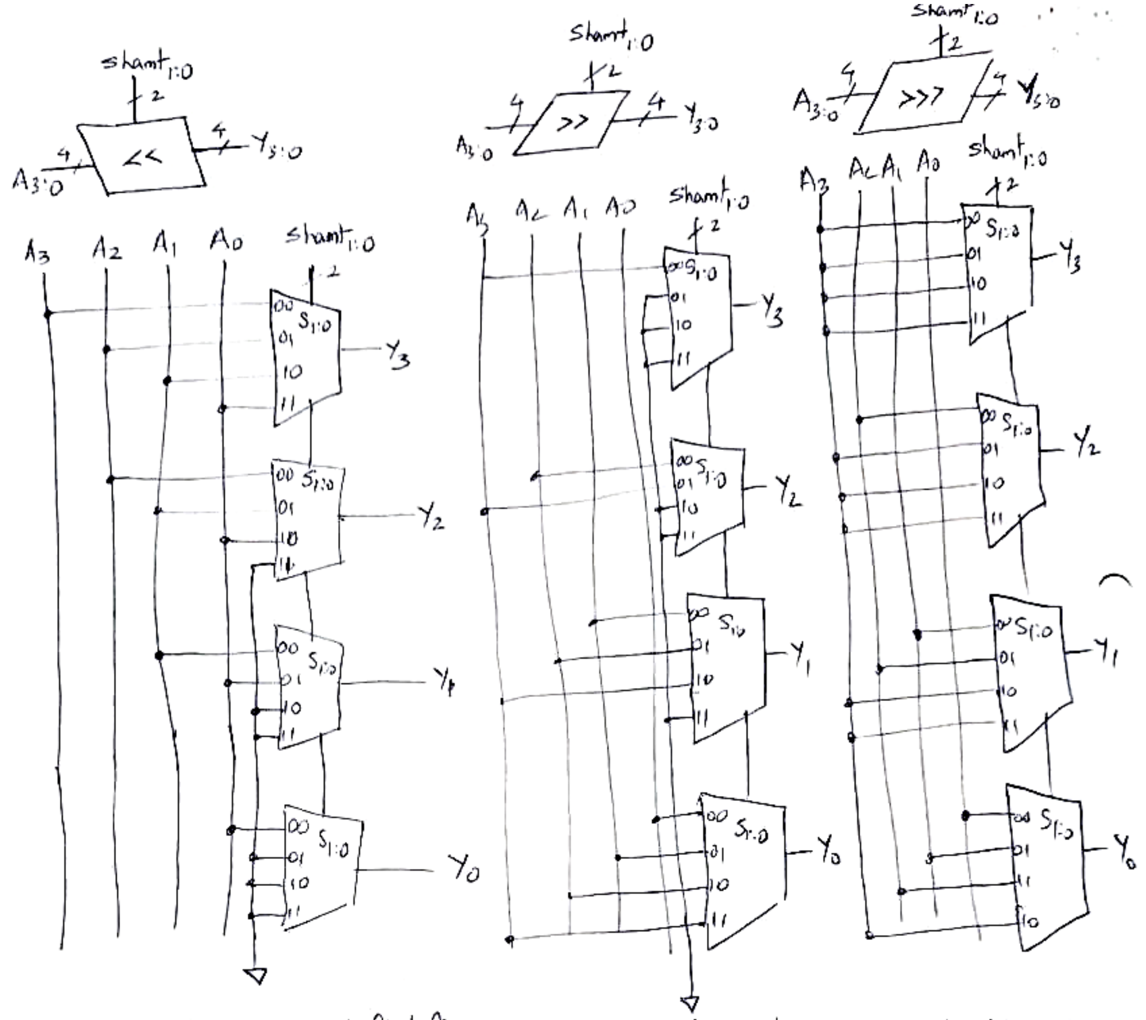
The i/p is shifted by 0 to N-1 bits, depending on the value of the $\log_2 N$ select lines. (shamt)

The operators <<, >> and >>> indicate SL, LSR, ASR

$shamt_{1:0}$ holds 2-bit shift amount. 4-bit shifter

Output Y, receives the input, A shifted by 0 to 3 bits.

When $shamt_{1:0} = 00$, Y = A.



4-bit shifter : logical shift left.   logical shift right.   arithmetic shift right.

Shift Rules :

A left shift is a special case of multiplication.

A left shift by 'N' bits multiplies the number by 2^N.

Example : $000011_2 << 4 = \overset{2^5}{1}\overset{2^4}{1}0000_2$

000011
000110 LSL 1
001100 LSL 2
011000 LSL 3
110000 LSL 4

Represent the following numbers as 8-bit signed binary and perform the following shifts. Validate the answer using the Shift rules

i) $+3_{10} << 4$ ii) $-4_{10} >>> 4$

$\therefore 3_{10} \times 2^4 = 48_{10}$

Arithmetic rightshift by N bits divides number by 2^N.

Example :- $11100_2 >>> 2 = 11111_2$

$\Rightarrow -4_{10}/2^2 = -1_{10}$ (show!)

Multiplication :

In multiplication operation of unsigned binary numbers, partial products are formed by multiplying (ANDing) a single bit of the multiplier with the entire multiplicand. Then the shifted partial products are added to form the result.

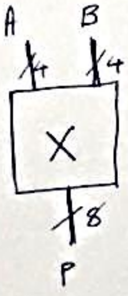
In general, an N×N Multiplier multiplies two N-bit numbers and produces a 2N-bit result. The partial products in binary multiplication are either the multiplicand or all 0's.

Multiplication of 1-bit binary numbers is same as the AND operation, so AND gates are used to form the partial products.

Example : Show $5_{10} \times 7_{10}$ on paper using binary representation

```
 5×  ≡  0101 ×   ← multiplicand
 7   ≡  0111     ← multiplier
---
35      -------
            0101     partial products
           0101
          0101    +
         0000
        -------
        0100011     addition of pps.
```

2^5 $2^1 2^0$



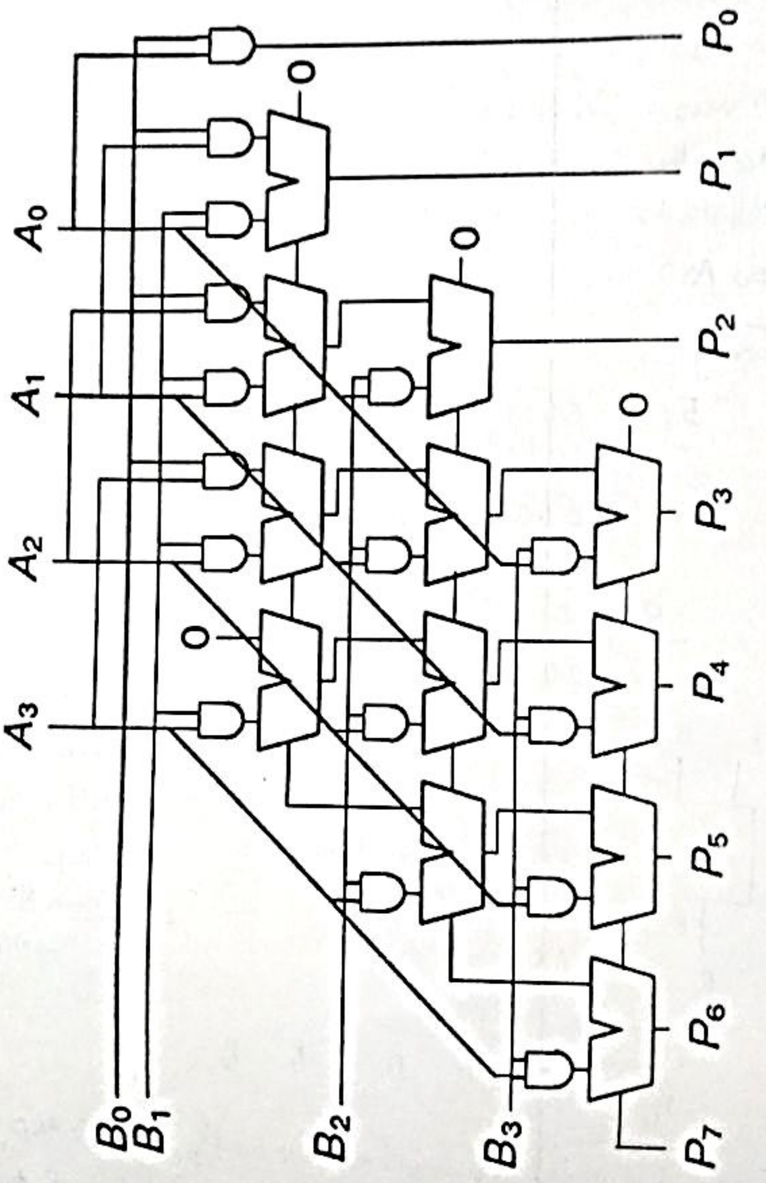
```
           A3    A2    A1    A0    ×
           B3    B2    B1    B0
          ---------------------------
           A3B0  A2B0  A1B0  A0B0
      A3B1 A2B1  A1B1  A0B1
  A3B2 A2B2 A1B2 A0B2                 +
A3B3 A2B3 A1B3 A0B3
---------------------------------------
P7  P6  P5  P4  P3   P2    P1    P0
```

With N-bit operands, there are N partial products and (N-1) stages of 1-bit adders.

Total: N^2 2-i/p AND GATES +N(N-1) 2-i/p (1-bit) ADDERS.

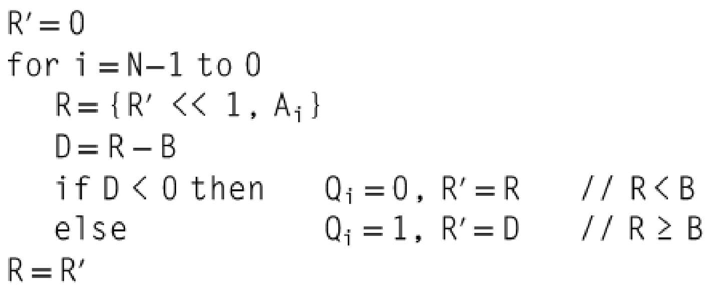
Each partial product is a single multiplier bit (B_3, B_2, B_1, B_0) AND the multiplicant bit (A_3, A_2, A_1, A_0). The partial product of first row is B_0 AND (A_3, A_2, A_1, A_0) This pp is added to shifted second pp, B_1 AND(A_3, A_2, A_1, A_0). Continue till the last pp for B_3 AND(A_3, A_2, A_1, A_0). [16 AND Gates + 12 1-bit ADDERS] ADDERS are needed to sum the corresponding values of the PPs.

HARDWARE FOR FOUR-BIT PARALLEL MULTIPLICATION



Division

Binary division can be performed using the following algorithm for N-bit unsigned numbers in the range [0, 2^N–1]:



```
R′ = 0
for i = N−1 to 0
  R = {R′ << 1, $A_i$}
  D = R − B
  if D < 0 then    $Q_i$ = 0, R′ = R    // R < B
  else             $Q_i$ = 1, R′ = D    // R ≥ B
R = R′
```

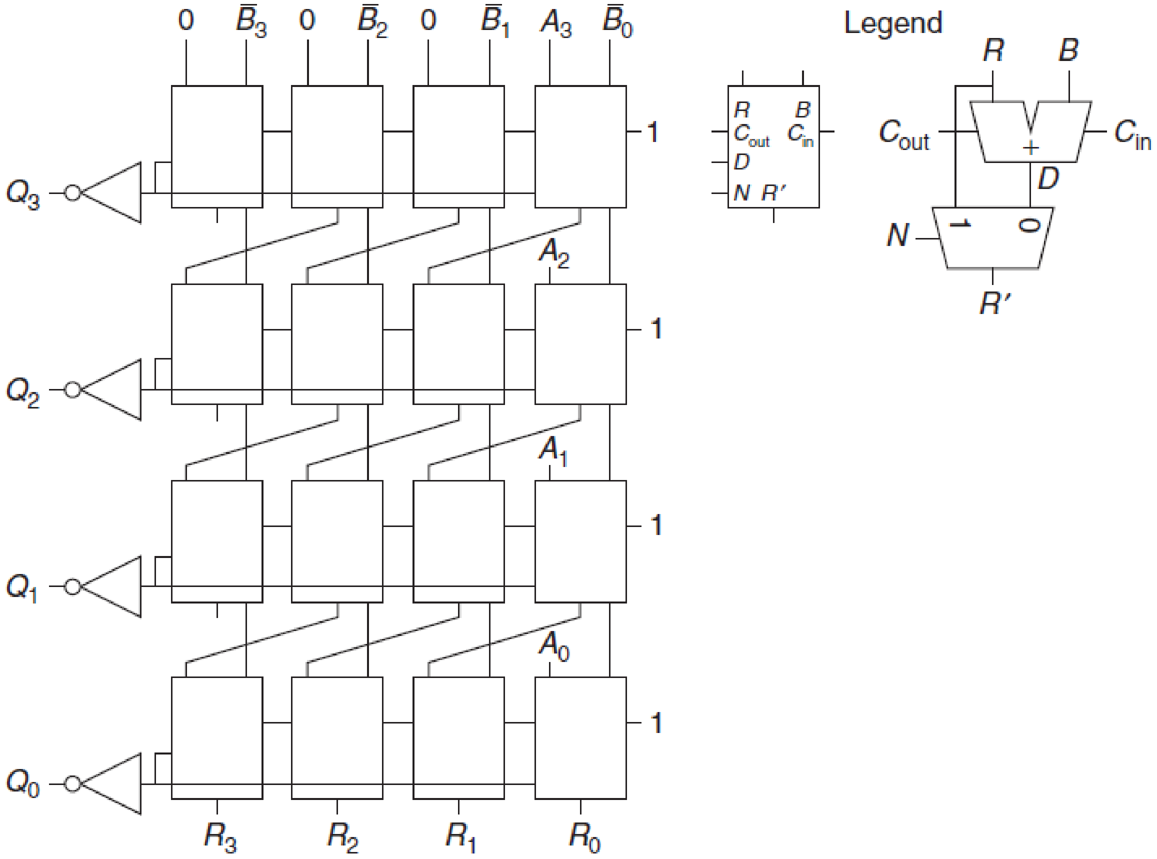
The partial remainder R is initialized to 0. The most significant bit of the dividend A then becomes the least significant bit of R. The divisor B is repeatedly subtracted from this partial remainder to determine whether it fits. If the difference D is negative (i.e., the sign bit of D is 1), then the quotient bit Qi is 0 and the difference is discarded. Otherwise, Qi is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R, and the process repeats. The result satisfies:

$$\frac{A}{B} = Q + \frac{R}{B}.$$

A schematic of a **4-bit Array Divider** is shown in the figure. The divider computes A/B and produces a quotient Q and a remainder R. The legend shows the symbol and schematic for each block in the array divider. The signal N indicates whether R − B is negative. It is obtained from the D output of the leftmost block in the row, which is the sign of the difference.

The delay of an N-bit array divider increases proportionally to N^2 because the carry must ripple through all N stages in a row before the sign is determined and the multiplexer selects R or D. This repeats for all N rows. Division is a slow and expensive operation in hardware and therefore should be used as infrequently as possible.

Four-bit Array Divider bit



Fixed-Point Number System :-

A fixed-point notation has an implied binary point between the integer and fraction bits.

Example : 1

$$0110.1100_2 \equiv 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75_{10}$$

Signed fixed-point numbers can use either sign magnitude notation or 2scomplement notation.

Example : 2

$$+2.375 \equiv 00\overset{2^1}{1}0.0\overset{2^{-2}}{1}\overset{2^{-3}}{1}0$$

$$-2.375 \equiv 1010.0110 \quad \text{(Sign and magnitude)}$$

$$\equiv 1101.1010 \quad \text{(2's complement)}$$

Example : 3 Arithmetic

Compute 0.75 + −0.625 using fixed-point numbers

$$0.75 \equiv 0000.\overset{2^{-1}}{1}\overset{2^{-2}}{1}00$$

$$+0.625 \equiv 0000.10\overset{2^{-3}}{1}0$$

$$(-0.625) \equiv 1111.0110 \quad \text{(2's Complement)}$$

$$0.125 \qquad \underset{\text{discard!}}{\textcircled{1}}\,0000.0010 \Rightarrow 0.125_{10}$$

Fixed-point numbers Systemsare commonly used for Banking and Financial applications that require precision, but not a large range.

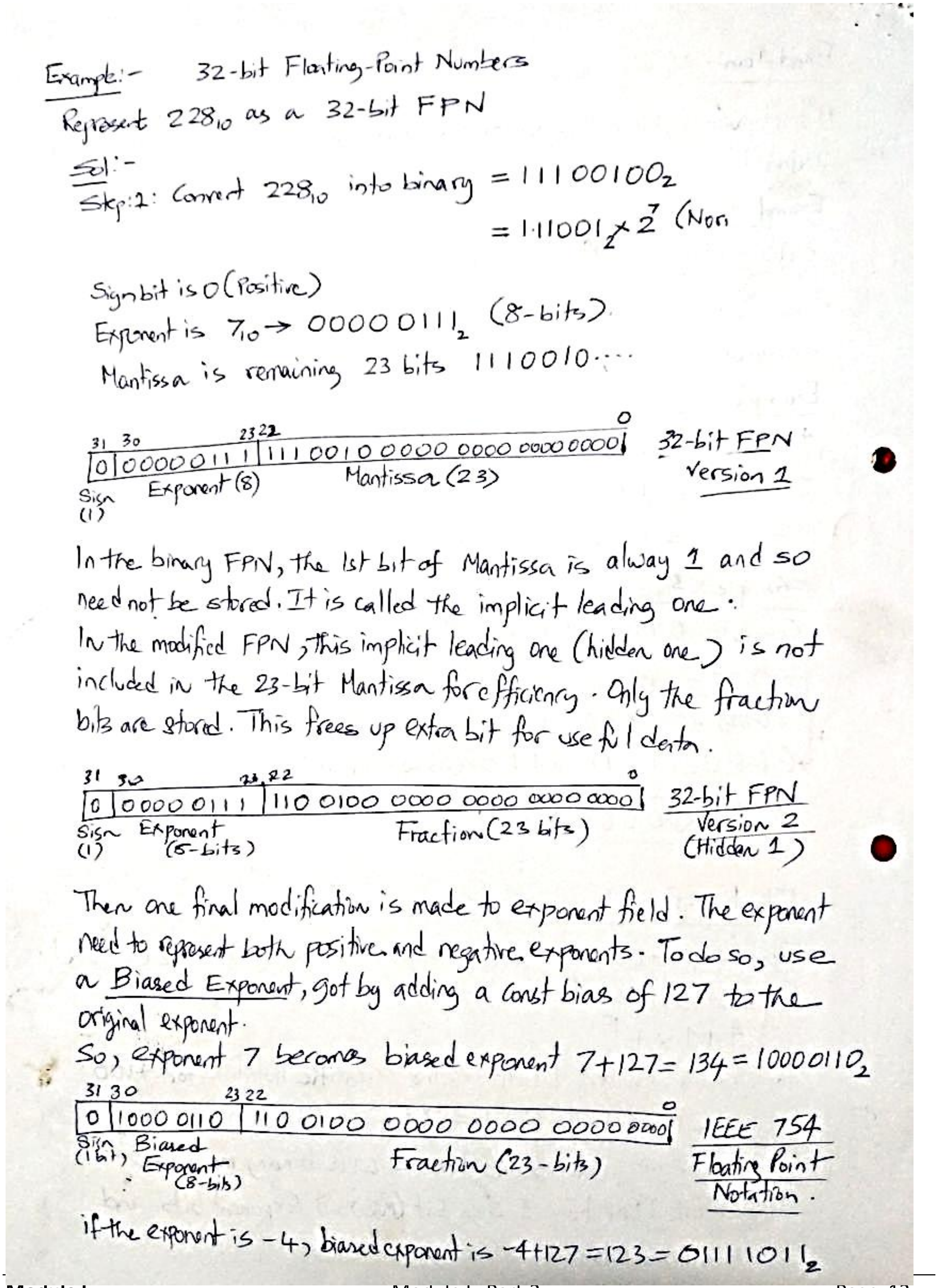
Floating-Point Number System :-

Floating-point numbers have a sign, Mantissa (M), Base (B) and Exponent (E) as shown below.

$$\pm M \times B^E$$

For example, number 4.1×10^3 is the scientific notation for 4100. It has M= 4.1, B=10, E= 3.

Floating-point numbers are base 2 with binary mantissa. A 32-bit FPN has 1 sign bit (msb), 8 Exponent bits and 23 Mantissa bits.



Example:- 32-bit Floating-Point Numbers

Represent 228_{10} as a 32-bit FPN

Sol:-

Step:1: Convert 228_{10} into binary $= 11100100_2$

$= 1.11001_2 \times 2^7$ (Nor

Sign bit is 0 (Positive)

Exponent is $7_{10} \rightarrow 00000111_2$ (8-bits).

Mantissa is remaining 23 bits 1110010····

31	30 23	22 0
0	00000111	1110010 0000 0000 00000000
Sign (1)	Exponent (8)	Mantissa (23)

32-bit FPN Version 1

In the binary FPN, the 1st bit of Mantissa is alway 1 and so need not be stored. It is called the implicit leading one.

In the modified FPN, this implicit leading one (hidden one) is not included in the 23-bit Mantissa for efficiency. Only the fraction bits are stored. This frees up extra bit for use ful data.

31	30 23	22 0
0	0000 0111	110 0100 0000 0000 0000 0000
Sign (1)	Exponent (5-bits)	Fraction (23 bits)

32-bit FPN Version 2 (Hidden 1)

Then one final modification is made to exponent field. The exponent need to represent both positive and negative exponents. To do so, use a Biased Exponent, got by adding a const bias of 127 to the original exponent.

So, exponent 7 becomes biased exponent $7+127 = 134 = 10000110_2$

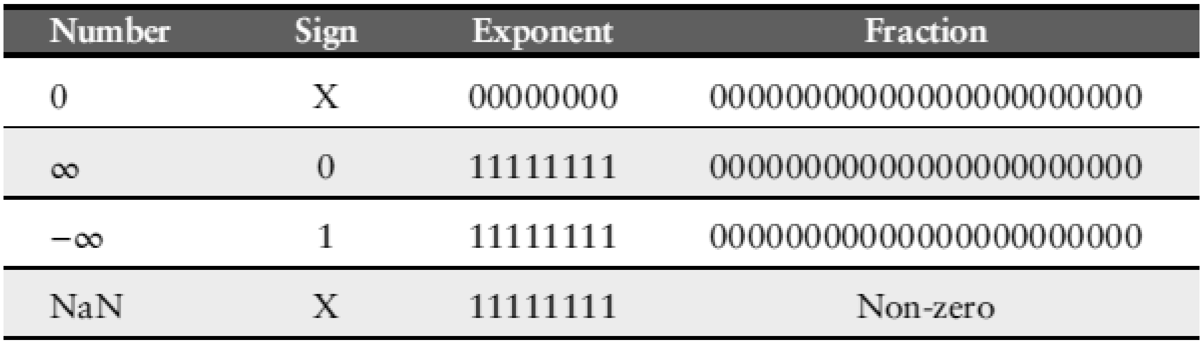
31	30 23	22 0
0	1000 0110	110 0100 0000 0000 0000 0000
Sign (1 bit)	Biased Exponent (8-bits)	Fraction (23-bits)

IEEE 754 Floating Point Notation.

if the exponent is -4, biased exponent is $-4+127=123=01111011_2$

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all l's are reserved for these special cases. Table shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating- point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.



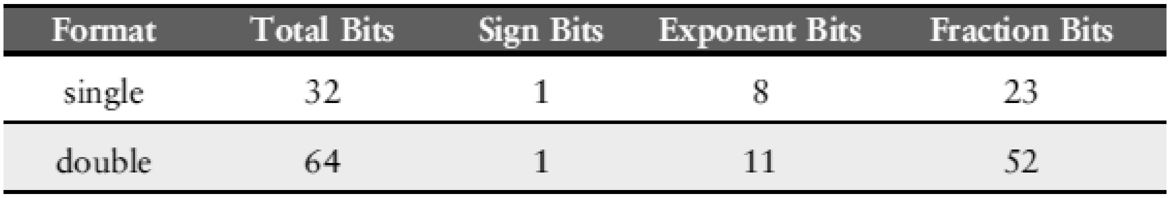
Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Single- and Double-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called single-precision, single, or float. The IEEE 754 standard also defines 64-bit double-precision numbers (also called doubles) that provide greater precision and greater range. Table shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$.

They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.



Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

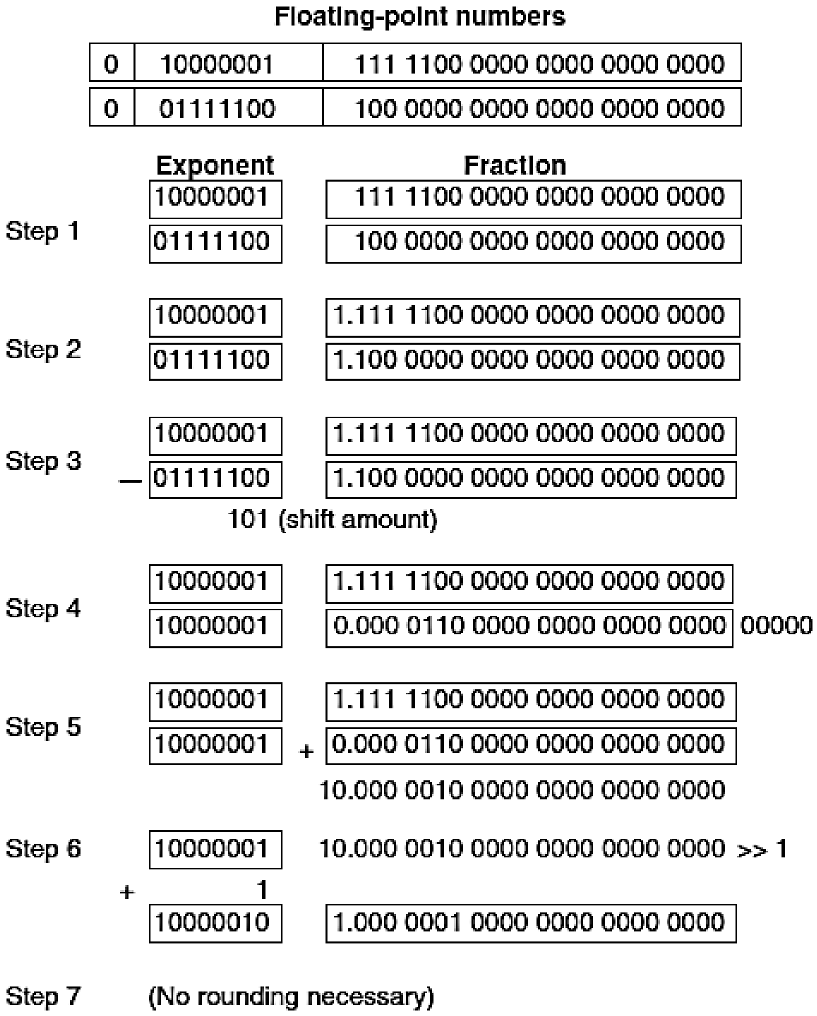
Recall that a number overflows when its magnitude is too large to be represented. Likewise, a number underflows when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to ±∞ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

Example shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}).
The result is 8.0625 (1.0000001 × 23).
After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.



Some interesting facts…

Floating-point cannot represent some numbers exactly, like 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999. . . . To handle this, some applications, such as calculators and financial software, use binary coded decimal (BCD) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example, the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A−F encodings are not used), and thus decreased performance. So for computer-intensive applications, floating-point is much faster.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the floating-point unit (FPU), is typically distinct from the central processing unit (CPU). The infamous floating-point division (FDIV) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly!

EC206 Computer Organization – Module II ECE SEM-4

Architecture :-

The architecture of the Computer is represented by the Instruction Set and Operand locations within Registers and Memory. It is the programmer's view of the Computer.

Many different Architectures exist, such as IA-32, MIPS, SPARC and PowerPC.

The complete vocabulary of the Instructions of a Computer is its Instruction Set.

Computer instructions include add, subtract, and jump and so on.

They indicate both the operation to perform and the Operands to use.

The Operands may come from Memory, from Registers or from the Instruction itself, as indicated by the Addressing mode used.

Every Architecture has its own language which is internally in binary format. So Intructions are in the form of bits.

However, for the ease of use by Progoammer (Humans), a symbolic ~~language~~ format is used to represent instructions and the program that is called Assembly language.

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the Operands on which to operate.

MIPS Architecture instructions are considered for Assembly language statements using registers, memory and constants.

Assembly language instructions form a program, that is converted to machine language of native architecture using Assembler.

Hence Assembly language is machine (architecture) dependent.

On the other hand, High Level Languages such as C, C++, Java are machine independent and use Interpreters or Compilers.

Patterson and Hennessy Principles

Design Principle 1: Simplicity favours Regularity.

The add and sub have consistent intruction format and also a consistent number of operands, two sources and one destination. Those are easier to encode and handle in hardware.

Design Principle 2:- Make the Common case fast.

Use of multiple assembly language instructions to perform more complex operations.

Design Principle 3: Smaller is faster.

The fewer the registers, the faster they can be accessed.

Design Principle 4:- Good design demands good compromises.

A Single instruction format would be simple but not flexible. The MIPS instr set makes the compromise of supporting three instruction formats.

A computer architecture does not define the underlying hardware implementation. So, many different h/w implementations of a single architecture exist.

For example, Intel and Advanced Micro Devices (AMD), both sell various microprocessors belonging to the same X86 architecture.

They all run the same programs, but use different underlying h/w and so offers trade-offs in Performance, Price and Power.

Some microprocessors are optimized for high performance servers, wheras others are optimized for long battery life in laptops.

MIPS Architecture - developed by John Hennessey at Stanford in 1980s

It is used in Silicon Graphics, Nintendo and CISCO..

Instructions :-

The most common operation computers perform is Addition.

Example:-1

Write code for adding variables b and c and writing result to a, in both high level language i.e., C and in MIPS assembly language.

C-language code	MIPS Assembly Code
a = b + c;	add a, b, c

Example:- 2 : Repeat for subtraction.

C-language code	MIPS Assembly Code
a = b - c;	sub a, b, c

The first part of the assembly instruction add or sub is called the mnemonic and indicates what operation to perform. The operation is performed on b and c, the source operands, and the result is written to a, the destination operand.

Example:- 3 : More complex code and use of comments

C-language code	MIPS Assembly Code

a = b + c - d; //single line comment
/* multiple-line
comment */
	sub t, c, d # t = c - d
	add a, b, t # a = b + t

The Assembly lang program requires a temp variable t, to store the intermediate result.

Reduced Instruction Set Computer (RISC):-

The MIPS Instruction Set has only simple, commonly used Instructions. The no:-of Instructions is kept small, so that the hardware needed to decode the Instruction and its Operands can be simple, small and fast. More elaborate operations, less common, are performed as sequence of multiple simple Instructions.

Complex Instruction Set Computer (CISC) :-

Architectures with many complex instructions, such as, Intel IA-32 are CISC. For example, IA-32 defines a "string move" instruction that copies a string of characters from one part of memory to another. Such an operation requires several simple instructions in a RISC machine. Also CISC architectures have more instructions, typically, 256, unlike a RISC with just 64 simple instructions.

But, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction coding by keeping the set of instructions small.

For example, an Instruction Set with 64 simple instructions needs $\log_2 64$ which is 6-bits to encode the operation (OPCODE size.)

But an Instruction Set with 256 instructions would need $\log_2 256 = 8$ bits of encoding per Instruction.

In a CISC machine, although the complex instructions are used rarely, they add overhead to all instructions, even the simple ones.

Operands : Registers, Memory, and Constants :-

An instruction operates on Operands that can be in memory, registes or Constants within the instruction itself.

MIPS is a 32-bit architecture, operating on 32-bit operands.

Operands stored in registers or as Constants are accessed quickly, but they hold only a small amount of data.

Additional size data must be accessed from memory, which is large but slow.

Registers :-

MIPS architectue uses 32 registers, called register set or register file.

Code Example-4 Register Operands

C language Code

```
a=b+c;
```

MIPS Assembly Code

```
#$s0=a, $s1=b, $s2=c
 add $s0, $s1, $s2   # a=b+c
```

Code Example-5 Temporary Registers

C language Code

```
a=b+c-d;
```

MIPS Assembly Code

```
# $s0=a, $s1=b, $s2=c, $s3=d
 sub $t0, $s2, $s3   # t=c-d
 add $s0, $s1, $t0   # a=b+t
```

MIPS register names are preceded by the $ sign.
The variables a, b and c are placed in $s0, $s1 and $s2.
The instruction adds the 32-bit values in $s1 and $s2 and writes the result to $s0.
MIPS stores variables in 18 of the 32 registers: $s0-$s7, and $t0-$t9.
Register names beginning with $s are called saved registers.
Register names beginning with $t are called temporary registers.
They store temporary variables.
$t0 is used to store the intermediate calculation of c-d.

Example-6 HLL to Assembly language

Translate the following HLL into ALP. Assume variables a-c are held in registers $s0-$s2 and f-j are in $s3-$s7.

```
a=b-c;
f=(g+h)-(i+j);
```

Sol:- The program uses four assembly language instructions.

```
#MIPS assembly code
#$s0=a, $s1=b, $s2=c, $s3=f, $s4=g, $s5=h
# $s6=i, $s7=j
  sub $s0, $s1, $s2   # a=b-c
  add $t0, $s4, $s5   #$t0=g+h
  add $t1, $s6, $s7   # $t1=i+j
  sub $s3, $t0, $t1   # f=(g+h)-(i+j)
```

The Register Set :-

MIPS architecture defines 32 registers. Each register has a name and a number ranging from 0 to 31.

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	function return value
$a0-$a3	4-7	function arguments
$t0-$t7	8-15	temporary variables
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	temporary var
$k0-$k1	26-27	Operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Memory:

Memory has many data locations compared to register file, but accessing it takes longer time. Whereas the register file is small and fast, memory is large and slow. So, commonly used variables are kept in registers.

MIPS architecture uses 32-bit memory addresses and 32-bit data words.

MIPS uses a byte-addressable memory, that is, each byte in memory has a unique address.

The 32-bit word address and 32-bit data values are expressed in hexadecimal.

Hex constants are written with the prefix 0x.

For example, data 0XF2F1AC07 represents a 32-bit operand in mem.

In a Memory Map, indicate low mem addresses towards bottom and high memory addresses towards the top.

Capacity of Memory is the max. no:-of addressable locations (bytes)

Byte-addressable memory:

Word Address	Data				Word
⋮		⋮			⋮
0000000C	40	F3	07	88	Word 3
00000008	01	EE	28	42	Word 2
00000004	F2	F1	AC	07	Word 1
00000000	AB	CD	EF	78	Word 0

← width = 4 bytes →

Memory Map

> Each 32-bit word in memory has four bytes (8-bits), each represented using two hexadecimal digits. So each hexadecimal word address is a multiple of 4.

> Byte addressable memories are organized in Big-Endian or Little-Endian fashion

Byte Address				Word Address	Byte Address			
	⋮			⋮		⋮		
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian

Little-Endian

Address Maps

Memory Addressing

In both formats, MSB is on the left and LSB (position & byte indicator) on right.

In big-endian machines, bytes are numbered starting with 0 at the big (most significant) end.

In little-endian machines, bytes are numbered starting with 0 at the little (least significant) end.

Word addresses are the same in both formats and refer to the same 4 bytes. Only the addresses of bytes within a word differ.

Memory access Instructions :-

The lw instruction specifies the effective address in memory as the sum of a base address and an offsets for load word.

The base address (in brackets) is a register.

The offset is a constant (before brackets).

The sw instruction is to write data word from register to memory. store word

Code Example - Accessing Byte-Addressable Memory

MIPS Assembly Code

```
lw $s0, 0($0)      # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)      # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)    # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)      # write $s3 to data word 1
sw $s4, 0x20($0)   # write $s4 to data word 8
sw $s5, 400($0)    # write $s5 to data word 100
```

Example : Big and Little-Endian Memory

Suppose that $s0 initially contains 0x23456789. Find the value of the register after the following program is run on :

(a) Big-Endian System (b) Little Endian System

```
sw $s0, 0($0)    # store $s0 to data word 0
lb $s0, 1($0)    # load byte at address (1+$0)=1 into least
                 # significant byte of $s0.
```

Big-Endian

Byte Address	0	1	2	3	Word Address
Data value	23	45	67	89	0
	MSB			LSB	sw $s0, 0($0)

Little-Endian

3	2	1	0	Byte Address
23	45	67	89	Data value
MSB			LSB	

(a) After store word instruction, memory word 0 contains the value shown. Then the load byte instruction would make ~~$s0~~ $s0 contain 0x00000045

(b) store word produces same value into mem word 0. After load byte instruction, ~~$0~~ $s0 will contain 0x00000067

IBM's Power PC used in Mac follows big-endian addressing, as well as Motorolla 680X0 series, and SPARC on Sun machines.
Intel's IA-32 architecture (PCs) as well as Intel X86 uses little-endian addressing.
MIPS has dual personality, but in examples little-endianness is preferred and used.
The choice of endianness is completely arbitrary but leads to hassles when sharing data between hetrogenous computers.
In the MIPS architecture, word addresses for lw and sw must be word alligned, i.e., the address must be divisible by 4.
Given the instruction, lw $s0, 7($0) it is an illegal instruction!
Note that byte addresses for lb and sb, need not be word alligned.

Constants:-

Constants when used with lw and sw are called immediates, since their values are immediately available within the Instruction, without a register or memory access.
Add immediate addi is a MIPS instr that uses immediate operand.
The constant in an instruction is a 16-bit 2's complement number in the range [-32,768, 32,767]. — i.e $[-2^{16}$ to $+2^{16}-1]$

Code Example Immediate Operands

C language code	MIPS Assembly Code
a = a+4; // add const 4 to a	# $s0 = a, $s1 = b
b = a - 12;	addi $s0, $s0, 4 #a=a+4
	addi $s1, $s0, -12 #b=a-12

The simple add and sub instr have three register operands.
The instr lw and addi, have 2 register operands and a 16-bit Const.

Machine Language :-

A program written in assembly language need to be translated from mnemonics to a representation using only bits (i.e., 0's and 1's) known as Machine Language. Only then can the digital circuits operate. MIPS uses 32-bit instructions. All Instructions can be encoded as words stored in memory. Variable-length instructions is ideal as all instructions may not require all 32-bits of encoding. But it adds too much complexity. On the other hand, a single instruction format results in simplicity, yet, it is too restrictive.

MIPS make the compromise by defining three Instruction Formats: R-type, I-type, and J-type.

- R-type instructions operate on three registers.
- I-type instructions operate on two registers and a 16-bit constant.
- J-type (jump) instructions operate on one 26-bit immediate.

This small number of formats allow for some regularity among all the types, and so simpler hardware can be used, while also accommodating different instruction needs, such as need to encode large constants in the Instruction.

R-type Instructions:

This stands for Register type instructions that use three registers as operands: Two registers as sources and one destination register.

op	rs	rt	rd	shamt	funct
6-bits	5-bits	5-bits	5-bits	5-bits	6-bits

R-type machine instruction format

The 32-bit R-type Instruction has Six fields: op, rs, rt, rd, shamt and funct. Each field is 5 or 6 bits, as indicated.

The operation the instruction performs is encoded in the two fields op (OPCODE) and funct (function)

All R-type instructions have an OPCODE of 0

The specific R-type operation is determined by the funct field.

For example, OPCODE and funct fields for add instruction are 0(000000_2) and 32($\overset{2^5}{1}0000\overset{2^0}{0}_2$), respectively.

Similarly, the sub instr has an op and funct field of 0 and 34.

The Operands are encoded in the 3 fields: rs, rt and rd.

The first two registers, rs and rt, are source registers and rd is the destination register. The fields contain register numbers. For example, $s0 is register 16.

The fifth field, shamt is used only in shift operations. The binary value stored in the 5-bit shamt field indicates the amont to shift. For all other R-type instr, shamt is 0.

Example : Machine code for R-type Instructions

Assembly code	op	rs	rt	rd	shamt	funct
add $s0, $s1, $s2	0	17	18	16	0	32
sub $t0, $t3, $t5	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Field values

op	rs	rt	rd	shamt	funct	
000000	$\overset{2^4}{1}000\overset{2^0}{1}$	10010	10000	00000	$\overset{2^5}{1}0000\overset{2^0}{0}$	(0x02328020)
000000	01011	01101	01000	00000	$\overset{2^5}{1}000\overset{2^1}{1}0$	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Machine code

HW! Translate to Machine code add $t0, $s4, $s5

I-Type Instructions :-

Immediate type instructions use two register operands and one immediate operand.

The 32-bit instruction has four fields: op, rs, rt and imm.

The first three fields are like those of R-type instructions.

The imm field holds the 16-bit immediate.

I-type Instruction

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

I-type instruction format

The operation is determined solely by the opcode.

The operands are specified in the three fields, rs, rt and imm.

rs and imm are always used as source operands.

rt is used as destination for some instructions (like addi and lw), but as another source for other instructions (like sw)

Example:- Translating I-type Assembly Instructions to Machine code

Translate the following I-type instruction into machine code.

lw $s3, -24($s4)

This instruction loads the memory word located at base address $s4 with an offset of -24 (16-bit value) into the register $s3.

lw $s3, -24($s4)

op	rs	rt	imm
35	20	19	-24
6 bits	5 bits	5 bits	16 bits

op	rs	rt	imm
100011	10100	10011	1111 1111 1110 1000
8 E		9 3	F F E 8

(0x8E93FFE8)

I-type instructions have a 16-bit immediate field that has to be used in a

32-bit operation.

For example, lw adds a 16-bit offset to a 32-bit register (base).

For positive immediates, the upper half of 32 bits should be all 0's,

dis! but for negative immediates, the upper half should be 1's.

This is known as sign extension rule.

An N-bit 2's complement number is sign-extended to an M-bit num (M>N) by copying the sign bit (msb) of the N-bit number into all of the upper bits of the M-bit number.

Sign-extending a 2's complement number does not change its value.

Most MIPS instructions sign-extend the immediate.

For example, addi, lw and sw do signextend to support both positive and negative immediates.

But, note that an exception to this rule is that logical instructions (andi, ori, xori) place 0's in the upper half. This is called zero extension rather than sign extension.

Example code

Field Values

Assembly Code	op	rs	rt	imm
addi $s0, $s1, 5	8	17	16	5
addi $t0, $s3, -12	8	19	8	-12
lw $t2, 32($0)	35	0	10	32
sw $s1, 4($t1)	43	9	17	4
	6 bits	5bits	5bit	16-bits

Machine Code

op	rs	rt	imm	
001000 2	10001 2 3	10000 0	0000 0000 0000 0101 0 0 0 5	(0x22300005)
001000 2	10011 2 6	01000 8	1111 1111 1111 0100 F F F 4	(0x2268FFF4)
				(0x8C0A0020)
				(0xAD310004)

J-Type Instruction :-

This is Jump type instruction and the format has a 6-bit opcode and a single 26-bit address operand addr

J-type

op	addr
6 bits	26 bits

A program can unconditionally branch or jump, using three types of Jump instructions:

1. Jump (j) directly jumps to the instruction at the specified label.
2. Jump and link (jal) is used by functions to save a return address
3. Jump register (jr) jumps to the address held in a register.

Code Example: Unconditional branching using j

MIPS Assembly Code

```
  addi   $s0, $0, 4       # $s0 = 4
  addi   $s1, $0, 1       # $s1 = 1
  j      target           # jump to target
 addi    $s1, $s1, 1      # notexecuted!
 sub     $s1, $s1, $s0    # not executed!

target:
 add    $s1, $s1, $s0     # $s1 = 1+4=5
```

Example: Translate the Jump (j) instruction given that target is a label whose address is 0xDEF1
op of j is 2 and of jal is 3.

Interpreting Machine Language Code :-

To interpret machine language, need to decipher the fields of each 32-bit instruction word.

Although different instructions use different formats, all formats begin with a 6-bit opcode field.

If the opcode is 0, the instruction is R-type ; else it is I-type or J-type.

Example :-

Translate the following machine language code into assembly language.

1. 0x2237FFF1 : $\boxed{001000}$10 ...... op is 8_{10} => addi
2. 0x02F34022 : $\boxed{000000}$01 .... op is 0 => R-type instr

Sol :-

1. Represent each instruction in binary and inspect the six most significant bits to find opcode of each instruction.

	op(6)	rs(5)	rt(5)	imm(16)
0x2237FFF1	001000	10001	10111	1111 1111 1111 0001

op	rs	rt	imm	
8	17	23	-15	addi $s7, $s2, -15

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	`add`	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	`addiu`	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	`addu`	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	`and`	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	`andi`	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	`jr`	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	`lbu`	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	`or`	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		$0 / 2a_{hex}$
Set Less Than Imm.	`slti`	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	`sltiu`	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	`sltu`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	$0 / 2b_{hex}$
Shift Left Logical	`sll`	R	R[rd] = R[rt] << shamt		$0 / 00_{hex}$
Shift Right Logical	`srl`	R	R[rd] = R[rt] >> shamt		$0 / 02_{hex}$
Store Byte	`sb`	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	`sc`	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	`sh`	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	`sw`	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	`sub`	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	`subu`	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	`bc1t`	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	`bc1f`	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	`div`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	`divu`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	`add.s`	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	`add.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	`c.x.s*`	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	`c.x.d*`	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	`div.s`	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	`div.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	`mul.s`	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	`mul.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	`sub.s`	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	`sub.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	`lwc1`	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	No

Module - III   EC206 CO

Addressing Modes :-

MIPS has five addressing modes that defines the ways of reading or writing Operands, as well as, manipulating current Instruction addresses, that are held in the Program Counter (PC).

1. Register-only Addressing:

This addressing mode uses only registers for all source and destination operands. All R-type Instructions use Register-only addressing.

2. Immediate Addressing:

Immediate addressing uses the 16-bit immediate along with registers as Operands. Some I-type instructions, like add immediate (addi) and load upper immediate (lui), use immediate addressing.

3. Base Addressing:

Memory access instructions, such as load word (lw) and store word (sw) use base addressing.

The effective address (EA) of the memory Operand is found by adding the base address in register rs to the sign-extended 16-bit offset found in the immediate field.

4. PC-relative addressing:

Conditional branch instructions use PC-relative addressing to specify the new value of the PC, if the branch is taken.

The signed offset in the immediate field is added to the PC to obtain the new PC.

So, the branch destination address is said to be relative to current PC.

Code Example Calculating the Branch Target Address

MIPS Assembly Code (fragment)

```
0xA4          beq   $t0, $0, else
0xA8          addi  $v0 , $0 , 1
0xAC          addi  $sp , $sp , 8
0xB0          jr    $ra
0xB4   else:  addi  $a0, $a0, -1
0xB8          jal   factorial
```

The branch targetr address (BTA) is the address of next instruction to execute if the branch is taken.

The branch if equal (beq) instruction has a BTA of 0xB4, the instruction address of the else label.

The 16-bit immediate field of beq instr, gives the no:-of instr between BTA and the instruction after the branch instruction with address PC+4.

As BTA is 3 instructions past PC+4 (0xA8), immediate field value is 3.

Assembly code

beq $t0, $0, else

Field Values

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	16-bits

Machine code

op	rs	rt	imm
000$\overset{2^2}{1}$00	0$\overset{2^3}{1}$000	00000	0000 0000 0000 00$\overset{2^1}{1}\overset{2^0}{1}$
6 bits	5 bits	5 bits	16-bits

(0x11000003)

Machine code for beq instruction

Example:

Calculate the immediate field and show the machine code for the bne (branch not equal) instruction in the following program.

```
#MIPS Assembly code
0x40 loop: add   $t1, $a0, $s0
           lb    $t1, 0($t1)
0x44
           add   $t2, $a1, $s0
0x48
           sb    $t1, 0($t2)
0x4C
           addi  $s0, $s0, 1
0x50
           bne   $t1, $0, loop
0x54
           lw    $s0 , 0($sp)
0x58
```

Sol:-

The branch target address (bta), 0x40, is 6 instructions behind PC+4 (0x58), so the immediate field is -6.

Assembly code

bne $t1, $0, loop

Field values

op	rs	rt	imm
5	9	0	-6
6 bits	5 bits	5 bits	16 bits

op	rs	rt	imm
000101	01001	00000	1111 1111 1111 1010
6-bits	5-bits	5-bits	16-bits

(0x1520FFFA)

Pseudo-Direct Addressing:

In direct addressing, an address is specified in the Instruction.
The jump instructions, j and jal, ideally would use direct addressing to specify a 32-bit jump target address (JTA), to indicate the instruction address to execute next.
But, the J-type instruction encoding does have only 26 bits to encode the JTA, as six bits are used for the opcode. So how to solve this dilemma?
To find solution to above problem, inspect every bit of JTA.
$JTA_{1:0}$, should always be 0, since Instructions are word alligned.
The next 26 bits $JTA_{27:2}$, are taken from the addr field of the J-Instruction.
The four most significant bits $JTA_{31:28}$, are obtained from the four most significant bits of PC+4.
Such a type of addressing mode is called Pseudo-direct addressing.

Code Example: Calculating the Jump Target Address (JTA)

MIPS Assembly code

```
0x0040005C        jal       sum
...
0x004000A0  sum:  add       $v0, $a0, $a1
```

The JTA of the jal instruction is 0x004000A0
The instruction uses pseudo-direct addressing.

Field values

Assembly Code	op 6-bits	addr 26-bits
jal sum	3	0x0100028

```
                     addr
  op
|000011|00 0001 0000 0000 0000 00101000|  (0x0C100028)
 6 bits
```

```
                                              (0x004000A0)
JTA  0000 0000 0100 0000 0000 0000 1010 0000
 ⇓
26-bit addr  0000 0000 0100 0000 0000 0000 1010 0000
                0   1    0    0    0   2    8
                                              (0x0100028)
```

The JTA of the jal Instruction is 0x004000A0.
The top four bits and bottom two bits of the JTA are discarded
The remaining bits are stored in the 26-bit address field (addr).

The Processor calalates the JTA from the J-type Instruction by appending two 0's and prepending the four most significant bits of (PC+4) to the 26-bit address field (addr).

Since the four most significant bits of the JTA are taken from (PC+4), the jump range is limited.

All J-type instructions, j and jal, use pseudo-direct addressing.

Note that the jump register instruction, jr, is not a J-type instr. It is an R-type instruction that jumps to the 32-bit value held in register rs.

The Memory Map :-

Rule:

For an address bus of size 'N' bits, the total number of addressable memory location bytes or byte capacity (M) is given by:

$$M = 2^N \text{ Bytes (B)}$$

Example:

Calalate the Memory byte capacity of machines with address bus size of (a) 16-bits, (b) 24-bits (c) 32-bits or h

(a) N = 16 bits

$$\therefore M = 2^N = 2^{16} B \dashrightarrow 1K$$
$$= 2^{10} \times 2^{6} B$$
$$= \underline{64KB}$$

(b) N = 24-bits $\dashrightarrow$ 1M

$$M = 2^{24} = 2^{20} \times 2^{4} B$$
$$= \underline{16 MB}$$

(c) N = 32 - bits $\dashrightarrow$ 1G

$$M = 2^{32} = 2^{30} \times 2^{2} B$$
$$= \underline{4GB}$$

Digital Measures

$1B = 8b$

$1K = 2^{10} B$

$1M = 2^{20} B = 2^{10} K$

$1G = 2^{30} B = 2^{10} M$

$1T = 2^{40} B = 2^{10} T$

MIPS has 32-bit addresses, the MIPS address spans 4 GB. Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC The MIPS Memory Map has addresses for data words for this complete range. It also incorporates the Stored Program concept and so the code Instructions as well as Operands are stored in memory and have a unique address.

Pseudoinstructions
Exceptions
Signed and unsigned instructions
Floating-point instructions

Module III (Cont…)

Based on true facts from:

David Money Harris, Sarah L Harris, Digital Design and Computer Architecture, Morgan Kaufmann – Elsevier, 2009

Pseudo-instructions

- MIPS is a RISC, so instr size and h/w complexity are minimized by keeping no: of instrs small.
- If an instr is not available in MIPS instr set, it is due to the fact that the same operation can be performed using one or more existing MIPS instrs.
- MIPS defines *Pseudo-instructions* that are not actually part of instr set but are commonly used by programmers and compilers.
- When converted to machine code, Pseudo-instructions are translated into one or more MIPS instrs.

Pseudo-instructions

- For ex, `load immediate` pseudo-instruction (`li`) loads a 32-bit constant using a combination of `lui` and `ori` instructions.
- The `multiply` pseudo-instruction (`mul`) provides a three-operand multiply, multiplying two registers and putting the 32 least significant bits of the result into a third register.
- The `no operation` pseudo-instruction (`nop`) performs no operation.
  - The PC is incremented by 4 upon its execution.
  - No other registers or memory values are altered.
  - The machine code for the `nop` instruction is `0x00000000`.

Pseudo-instruction	MIPS Instructions

`li $s0, 0x1234AA77`	`lui $s0, 0x1234`
	`ori $s0, 0xAA77`

`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1,$0`
`nop`	`sll $0, $0, 0`

Pseudo-instructions

- Some pseudo-instructions require a temporary register for intermediate calculations.
- For example, the pseudo-instruction `beq $t2, imm`$_{15:0}$`, Loop` compares `$t2` to a 16-bit immediate, `imm`$_{15:0}$.
- Need a temp register to store the 16-bit immediate.
- Assemblers use the assembler register, `$at`, for such purposes.
- It uses `$at` to converting a pseudo-instruction to real MIPS instructions.-→

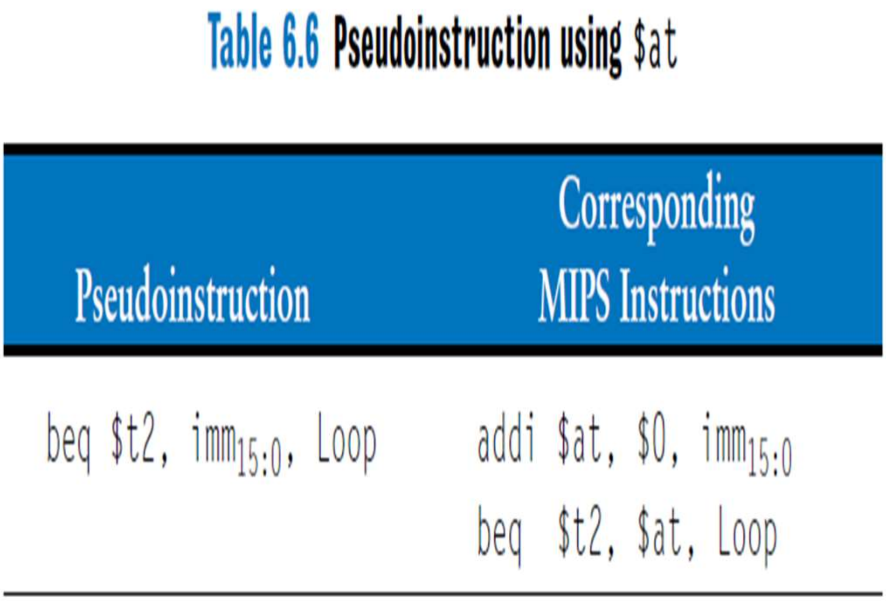


Table 6.6 Pseudoinstruction using $at

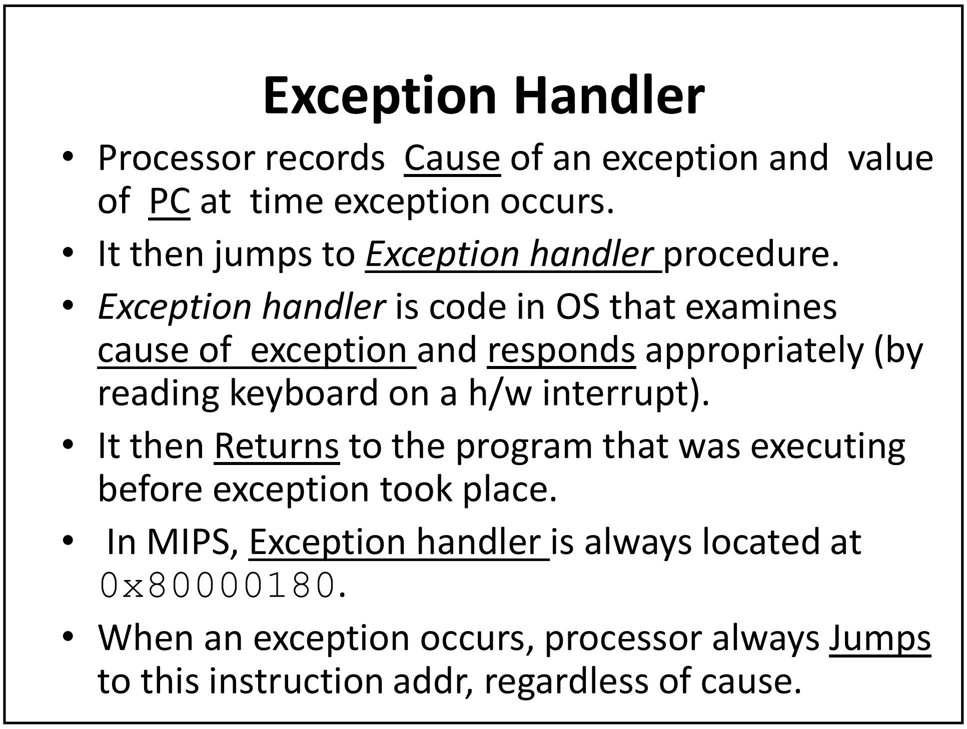
Pseudoinstruction	Corresponding MIPS Instructions
beq $t2, imm$_{15:0}$, Loop	addi $at, $0, imm$_{15:0}$ beq $t2, $at, Loop

Exceptions

- An *Exception* is an Unexpected, Unscheduled Procedure Call that causes the Processor to **Jump** to a new Address in Memory.
- Exceptions may be caused by h/w or s/w.
- For ex, processor may receive signal that user pressed a key on keyboard.
- Processor stops what it is doing, determine which key was pressed, save it for future reference, then resume the program that was running.
- Such a h/w exception triggered by an Input/ Output (I/O) device such as a keyboard is called an *Interrupt*.

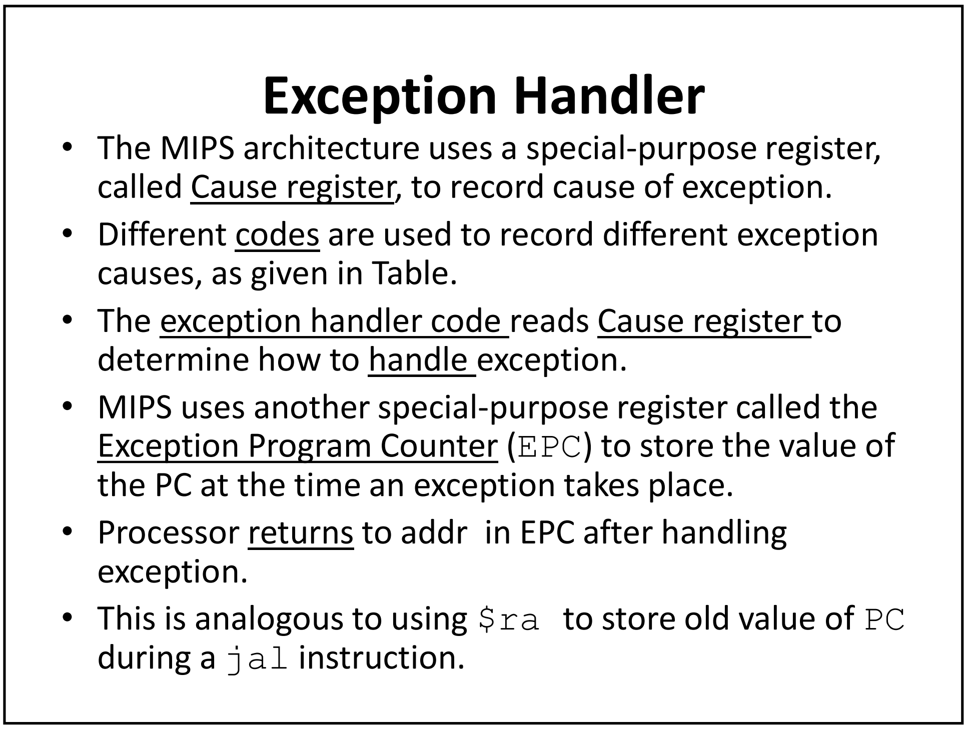
Exceptions

- Alternatively, program may encounter an error condition such as an undefined instruction.
- The program then jumps to code in *operating system* (*OS*), which may choose to terminate the offending program.
- S/w exceptions are sometimes called *Traps*.
- Other causes of exceptions incl div by 0, attempts to read non-existent memory, h/w malfunctions, debugger break-points, and arithmetic overflow.



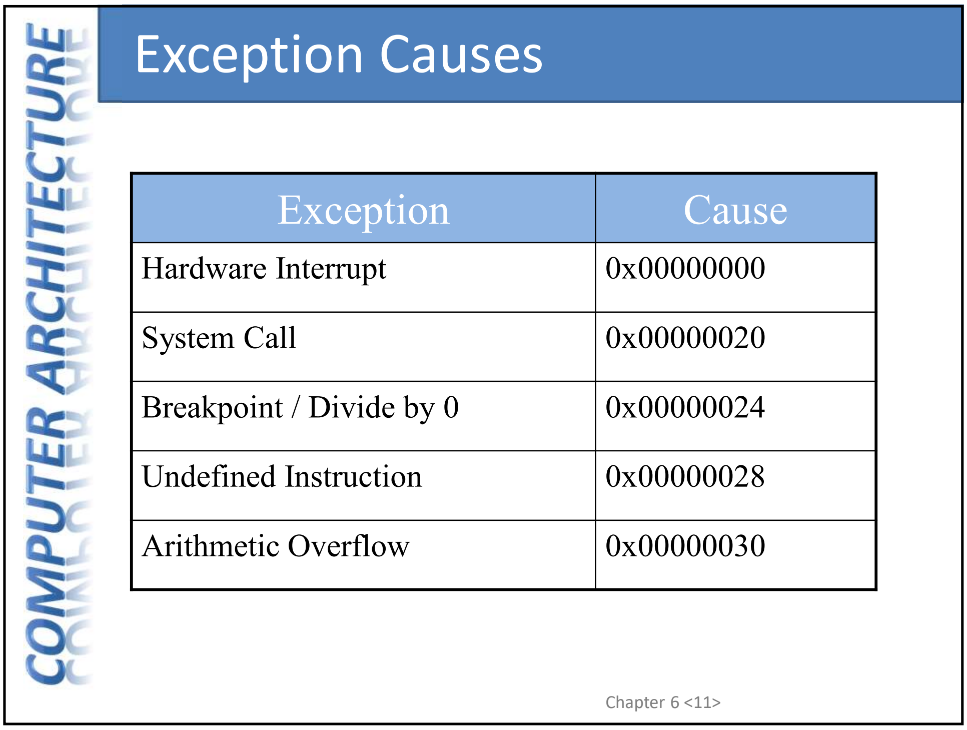
Exception Handler

- Processor records Cause of an exception and value of PC at time exception occurs.
- It then jumps to *Exception handler* procedure.
- *Exception handler* is code in OS that examines cause of exception and responds appropriately (by reading keyboard on a h/w interrupt).
- It then Returns to the program that was executing before exception took place.
- In MIPS, Exception handler is always located at `0x80000180`.
- When an exception occurs, processor always Jumps to this instruction addr, regardless of cause.



Exception Handler

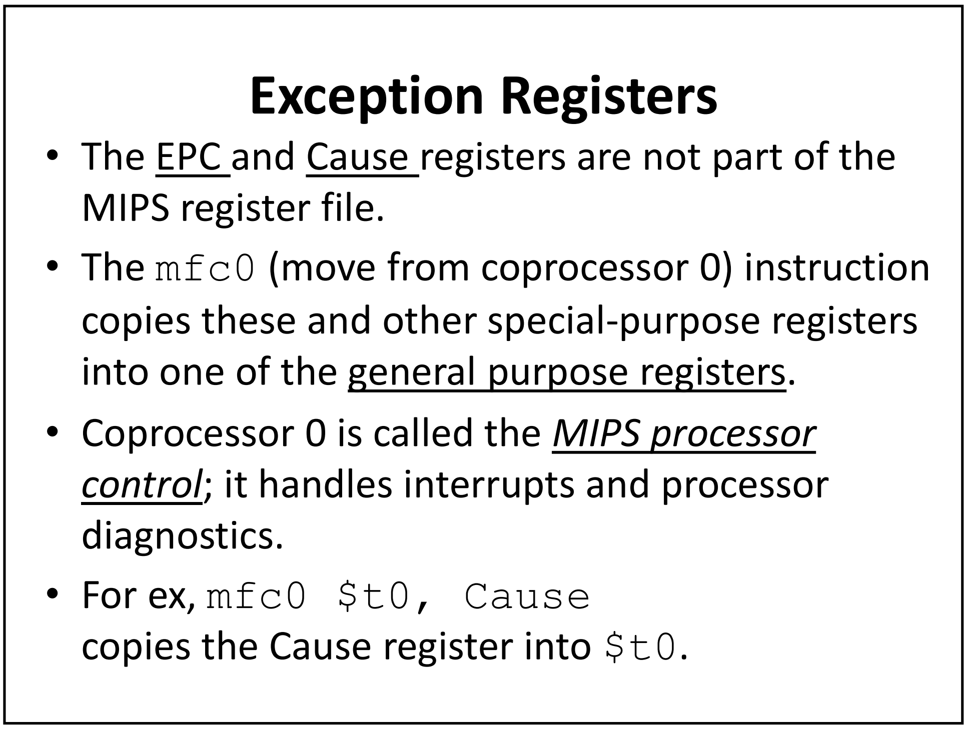
- The MIPS architecture uses a special-purpose register, called Cause register, to record cause of exception.
- Different codes are used to record different exception causes, as given in Table.
- The exception handler code reads Cause register to determine how to handle exception.
- MIPS uses another special-purpose register called the Exception Program Counter (`EPC`) to store the value of the PC at the time an exception takes place.
- Processor returns to addr in EPC after handling exception.
- This is analogous to using `$ra` to store old value of `PC` during a `jal` instruction.


COMPUTER ARCHITECTURE

Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

Chapter 6 <11>



Exception Registers

- The <u>EPC</u> and <u>Cause</u> registers are not part of the MIPS register file.
- The `mfc0` (move from coprocessor 0) instruction copies these and other special-purpose registers into one of the <u>general purpose registers</u>.
- Coprocessor 0 is called the *<u>MIPS processor control</u>*; it handles interrupts and processor diagnostics.
- For ex, `mfc0 $t0, Cause`
  copies the Cause register into `$t0`.

Exception Registers

- The `syscall` and `break` instructions cause traps to perform system calls or debugger breakpoints.
- Exception handler uses EPC to look up instr and determine nature of system call or breakpoint by looking at fields of instr.

-
-
-

Signed and Unsigned Instructions

- Addition and Subtraction
- Multiplication and Division
- Set less than

Addition and Subtraction

- Addition and subtraction are performed identically whether number is signed or unsigned.
- But, interpretation of the results is different.
- If two large signed numbers are added together, the result may incorrectly produce the opposite sign.
- For ex, adding the following two huge positive numbers gives a negative result: `0x7FFFFFFF+0x7FFFFFFF=0xFFFFFFFE=-2`
- Similarly, adding two huge negative numbers gives a positive result,

  `0x80000001+0x80000001=0x00000002.`
- This is called Arithmetic *Overflow*.
- MIPS processor takes an Exception on Arithmetic Overflow.

Addition and Subtraction

- MIPS provides signed and unsigned versions of Addition and Subtraction.
- Signed versions are `add`, `addi`, `sub`.
- Unsigned versions are `addu`, `addiu`, `subu`.
- The two versions are identical except that signed versions trigger an Exception on Overflow, whereas unsigned versions do not.
- Because C ignores Exceptions, C programs technically use the unsigned versions of these instructions.

Multiplication and Division

- Multiplication and division behave differently for signed and unsigned numbers.
- For ex, as an unsigned number, `0xFFFFFFFF` represents a large number, but as a signed number it represents `1`.
- Hence, `0xFFFFFFFF x 0xFFFFFFFF` would equal `0xFFFFFFFE00000001` if the numbers were unsigned but `0x0000000000000001` if the numbers were signed.

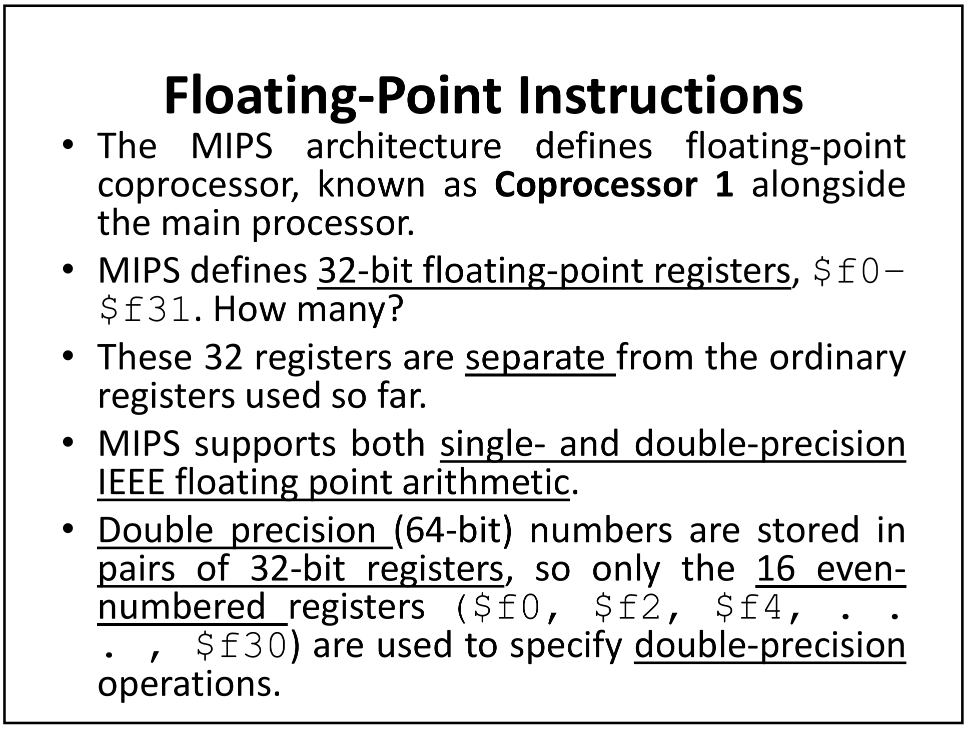
- Therefore, multiplication and division come in both signed and unsigned flavors.
- `mult` and `div` treat the operands as signed numbers.
- `multu` and `divu` treat the operands as unsigned numbers.

Set Less Than

- Set less than instructions can compare either *two registers* (`slt`) or a *register and an immediate* (`slti`).
- Set less than also comes in signed (`slt` and `slti`) and unsigned (`sltu` and `sltiu`) versions.
- In a signed comparison, `0x80000000` is less than any other number, because it is the most negative 2's complement number.
- In an unsigned comparison, `0x80000000` is greater than `0x7FFFFFFF` but less than `0x80000001`, because all numbers are positive.
- Beware that `sltiu` sign-extends the immediate before treating it as an unsigned number.

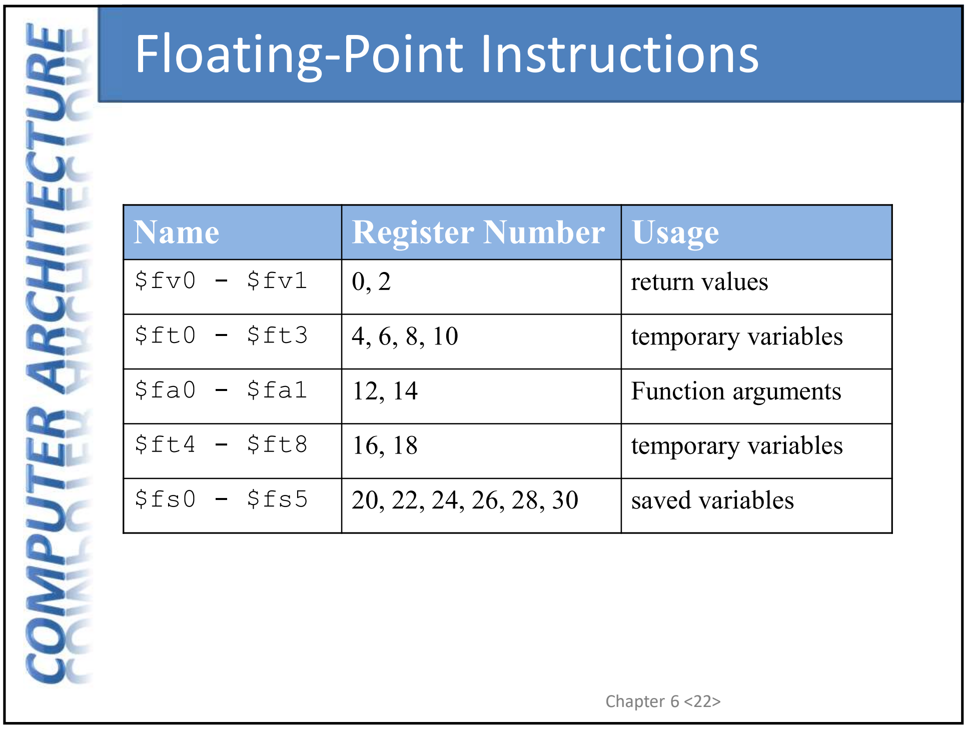
Loads

- Byte loads come in signed (`lb`) and unsigned (`lbu`) versions.
- `lb` sign-extends the byte, and `lbu` zero-extends the byte to fill the entire 32-bit register.
- Similarly, MIPS provides signed and unsigned half-word loads (`lh` and `lhu`), which load two bytes into the *lower half* and sign- or zero-extend the *upper half* of the word.



Floating-Point Instructions

- The MIPS architecture defines floating-point coprocessor, known as **Coprocessor 1** alongside the main processor.
- MIPS defines 32-bit floating-point registers, `$f0`–`$f31`. How many?
- These 32 registers are separate from the ordinary registers used so far.
- MIPS supports both single- and double-precision IEEE floating point arithmetic.
- Double precision (64-bit) numbers are stored in pairs of 32-bit registers, so only the 16 even-numbered registers (`$f0, $f2, $f4, . . . , $f30`) are used to specify double-precision operations.

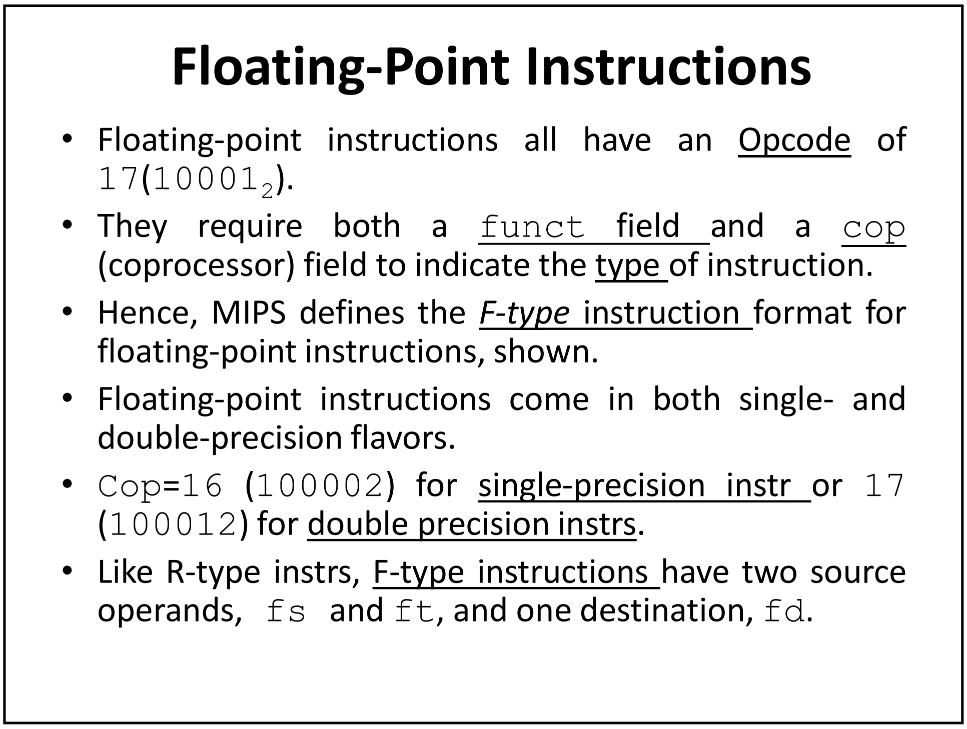


COMPUTER ARCHITECTURE

Floating-Point Instructions

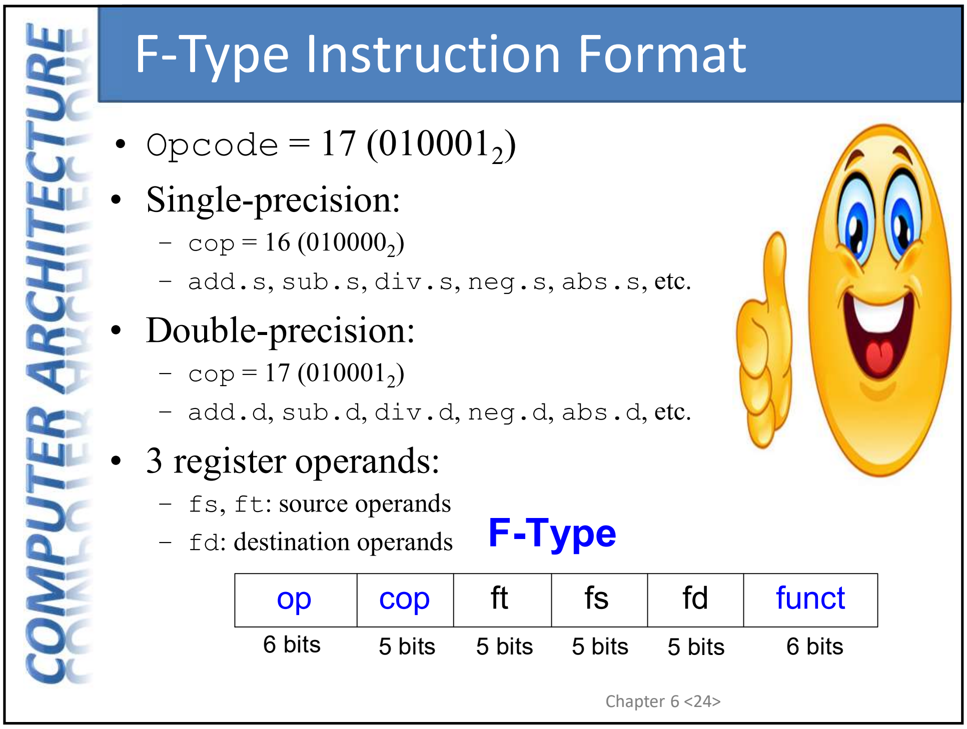
Name	Register Number	Usage
`$fv0 – $fv1`	0, 2	return values
`$ft0 – $ft3`	4, 6, 8, 10	temporary variables
`$fa0 – $fa1`	12, 14	Function arguments
`$ft4 – $ft8`	16, 18	temporary variables
`$fs0 – $fs5`	20, 22, 24, 26, 28, 30	saved variables

Chapter 6 <22>



Floating-Point Instructions

- Floating-point instructions all have an Opcode of 17(10001_2).
- They require both a funct field and a cop (coprocessor) field to indicate the type of instruction.
- Hence, MIPS defines the *F-type* instruction format for floating-point instructions, shown.
- Floating-point instructions come in both single- and double-precision flavors.
- Cop=16 (100002) for single-precision instr or 17 (100012) for double precision instrs.
- Like R-type instrs, F-type instructions have two source operands, fs and ft, and one destination, fd.



COMPUTER ARCHITECTURE

F-Type Instruction Format

- Opcode = 17 (010001_2)
- Single-precision:
  - cop = 16 (010000_2)
  - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Double-precision:
  - cop = 17 (010001_2)
  - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 register operands:
  - fs, ft: source operands
  - fd: destination operands

F-Type

op	cop	ft	fs	fd	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

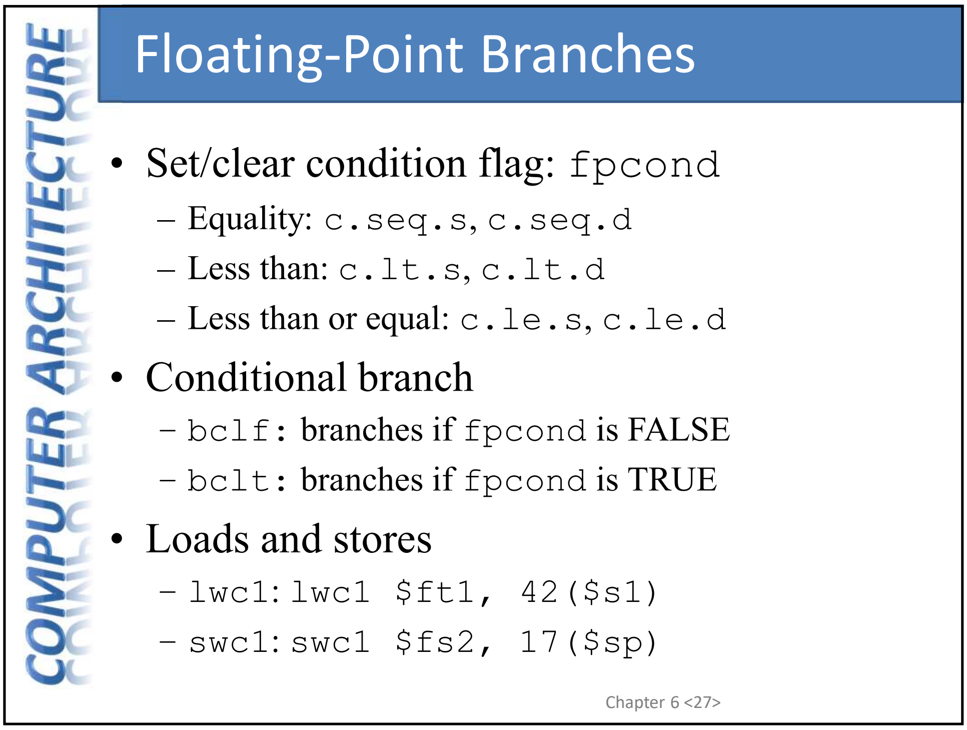
Chapter 6 <24>

Floating-Point Instructions

- Instruction precision is indicated by `.s` and `.d` in the mnemonic.
- Floating-point arithmetic instructions include:
- Addition (`add.s`, `add.d`)
- Subtraction (`sub.sz`, `sub.d`)
- Multiplication (`mul.s`, `mul.d`),
- Division (`div.s`, `div.d`)
- Negation (`neg.s`, `neg.d`)
- Absolute value (`abs.s`, `abs.d`).

Floating-point branches

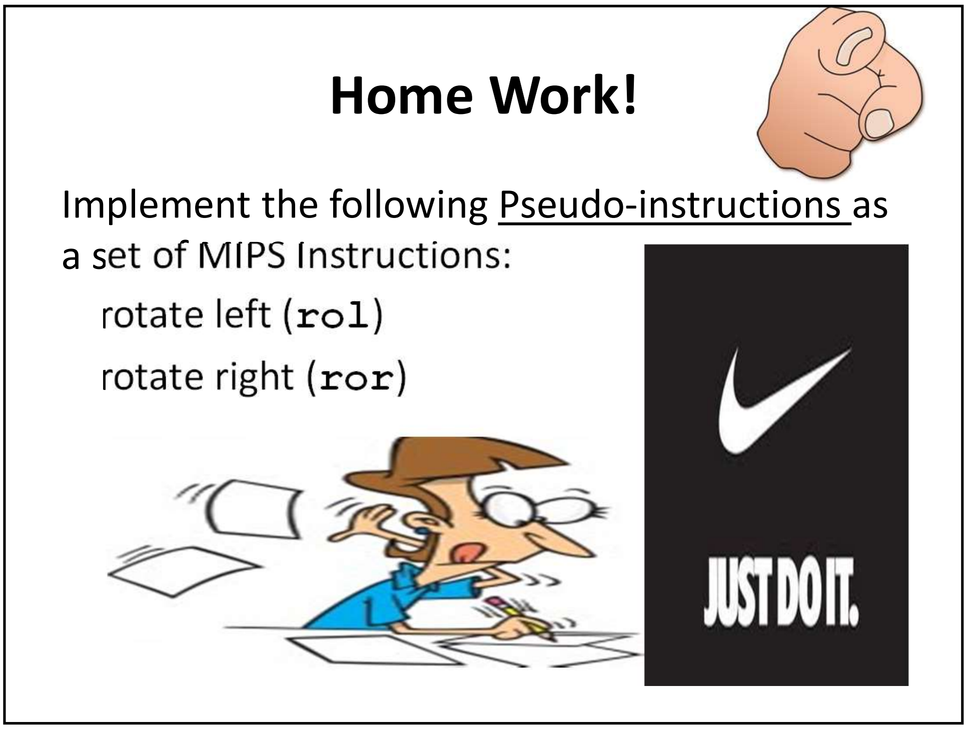
- These have two parts:
- First, a compare instruction is used to set or clear the *floating-point condition flag* (`fpcond`).
- Then, a conditional branch checks the value of the flag.
- The compare instructions include equality (`c.seq.s`/`c.seq.d`), less than (`c.lt.s`/`c.lt.d`), and less than or equal to (`c.le.s`/`c.le.d`).
- The conditional branch instructions are `bc1f` and `bc1t` that branch if `fpcond` is FALSE or TRUE, respectively.
- Inequality, greater than or equal to, and greater than comparisons are performed with `seq`, `lt`, and `le`, followed by `bc1f`.
- Floating-point registers are loaded and stored from memory using `lwc1` and `swc1`.
- These instructions move 32 bits, so two instructions are necessary to handle a double-precision number.



Floating-Point Branches

- Set/clear condition flag: `fpcond`
  - Equality: `c.seq.s`, `c.seq.d`
  - Less than: `c.lt.s`, `c.lt.d`
  - Less than or equal: `c.le.s`, `c.le.d`
- Conditional branch
  - `bc1f`: branches if `fpcond` is FALSE
  - `bc1t`: branches if `fpcond` is TRUE
- Loads and stores
  - `lwc1`: `lwc1 $ft1, 42($s1)`
  - `swc1`: `swc1 $fs2, 17($sp)`

Chapter 6 <27>



Home Work!

Implement the following Pseudo-instructions as a set of MIPS Instructions:

rotate left (**`rol`**)

rotate right (**`ror`**)

Translating and Starting a Program

Module III Topic

Based on true facts from:

David Money Harris, Sarah L Harris, Digital Design and Computer Architecture, Morgan Kaufmann – Elsevier, 2009

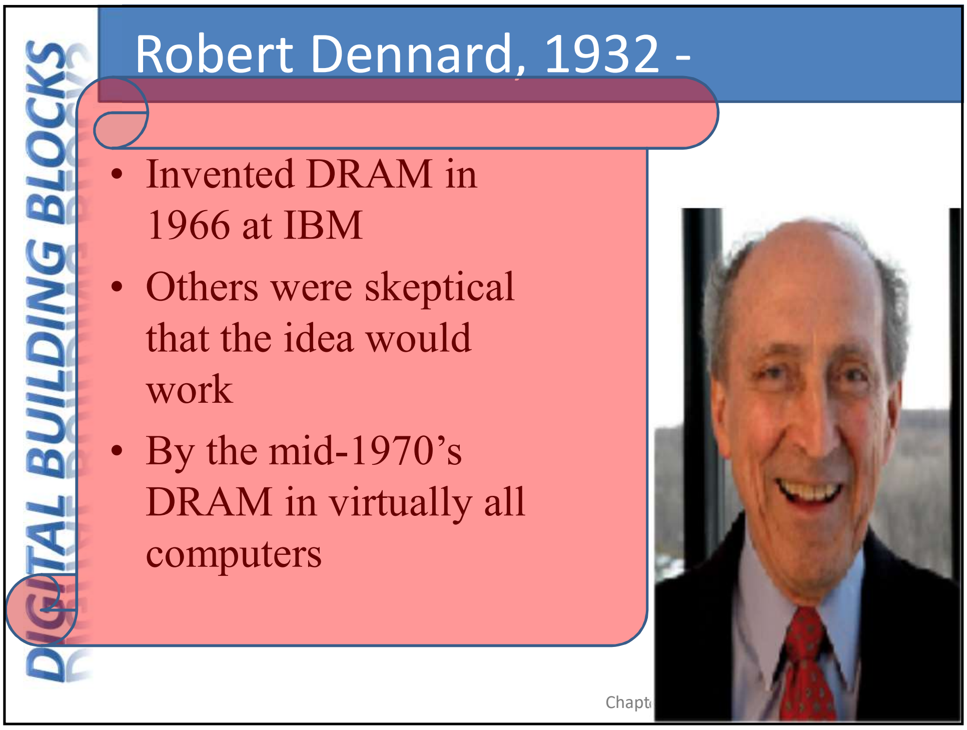
- Firstly…
- Some **Pioneers** of *Computer Organization and Architecture*…
- To **Inspire** you all…

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage’s Analytical Engine
- She was the daughter of the poet Lord Byron

Grace Hopper, 1906-1992

- Graduated from Yale University with a Ph.D. in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others

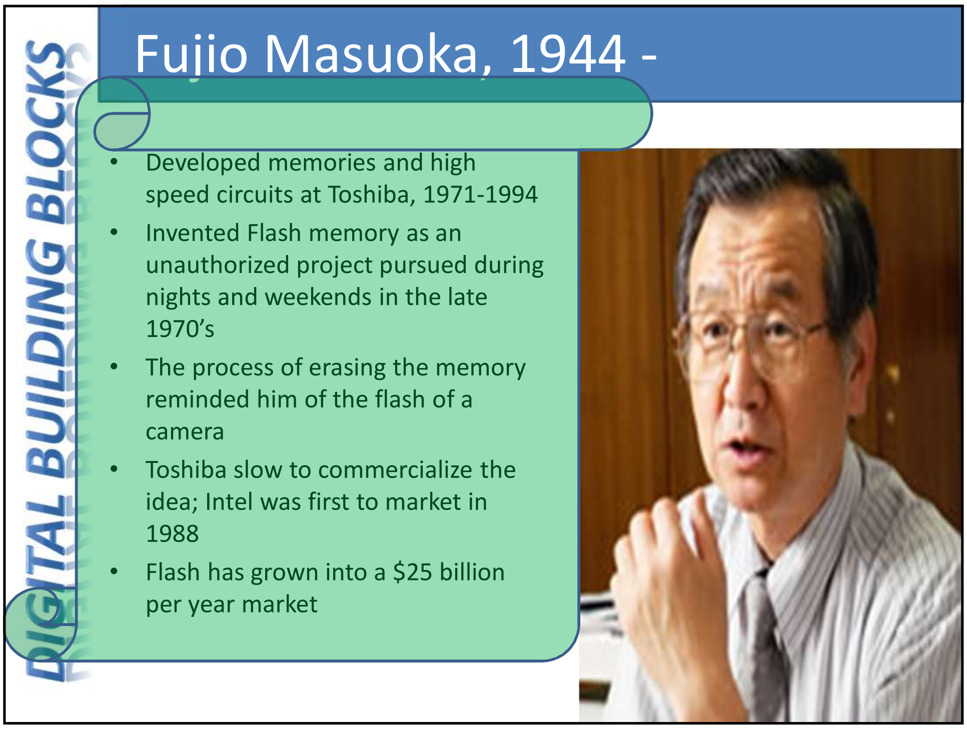


DIGITAL BUILDING BLOCKS

Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers

Chapt



DIGITAL BUILDING BLOCKS

Fujio Masuoka, 1944 -

- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a $25 billion per year market

John Hennessy

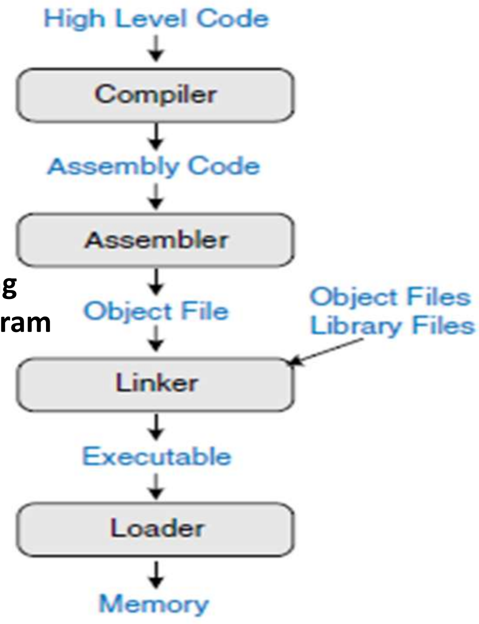
- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Co-invented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

MIPS Memory Map

Address	Segment
0xFFFFFFFC 0x80000000	Reserved
0x7FFFFFFC 0x10010000	Stack ↓ Dynamic Data ↑ Heap
0x1000FFFC 0x10000000	Static Data
0x0FFFFFFC 0x00400000	Text
0x003FFFFC 0x00000000	Reserved

Chapter 6 <8>

Steps for Translating and Starting a Program

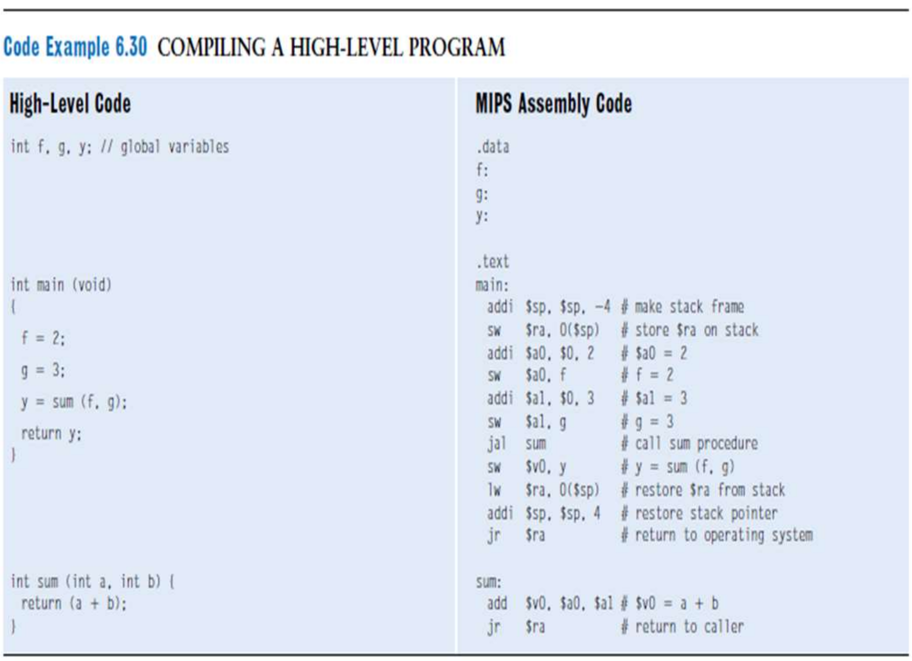


Steps…

- Steps required to translate a program from a high level language into machine language and to start executing that program are indicated .
- First, the high-level code is compiled into assembly code.
- The assembly code is assembled into machine code in an *object file*.
- The linker combines the machine code with object code from libraries and other files to produce an entire executable program.
- In practice, most compilers perform all three steps of compiling, assembling, and linking.
- Finally, the loader loads the program into memory and starts execution.

Step 1: Compilation

- A compiler translates high-level code into assembly language.
- Code Example 6.30 shows a simple hll program with three global variables and two procedures, along with the assembly code produced by a typical compiler.
- The `.data` and `.text` keywords are *assembler directives* that indicate where the text and data segments begin.
- Labels are used for global variables `f`, `g`, and `y`.
- Their storage location will be determined by the assembler.



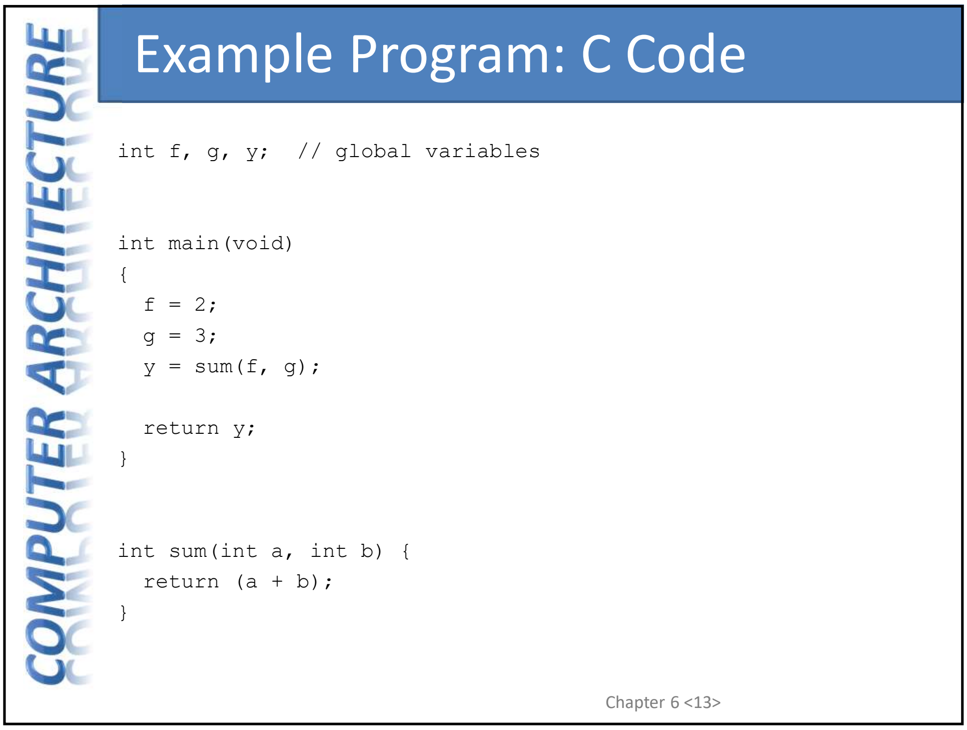
Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables




int main (void)
{
  f = 2;
  g = 3;
  y = sum (f, g);
  return y;
}



int sum (int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

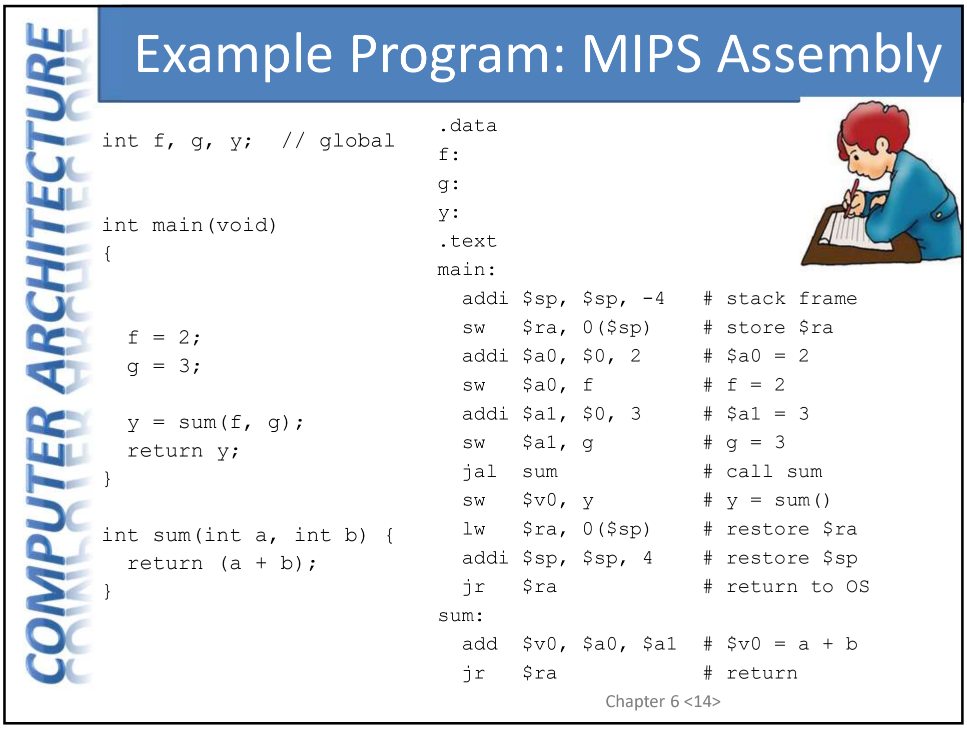
```
.data
f:
g:
y:

.text
main:
  addi  $sp, $sp, -4  # make stack frame
  sw    $ra, 0($sp)   # store $ra on stack
  addi  $a0, $0, 2    # $a0 = 2
  sw    $a0, f        # f = 2
  addi  $a1, $0, 3    # $a1 = 3
  sw    $a1, g        # g = 3
  jal   sum           # call sum procedure
  sw    $v0, y        # y = sum (f, g)
  lw    $ra, 0($sp)   # restore $ra from stack
  addi  $sp, $sp, 4   # restore stack pointer
  jr    $ra           # return to operating system

sum:
  add   $v0, $a0, $a1 # $v0 = a + b
  jr    $ra           # return to caller
```



Example Program: C Code

```
int f, g, y;  // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

Chapter 6 <13>



Example Program: MIPS Assembly

```
int f, g, y;  // global


int main(void)
{


  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

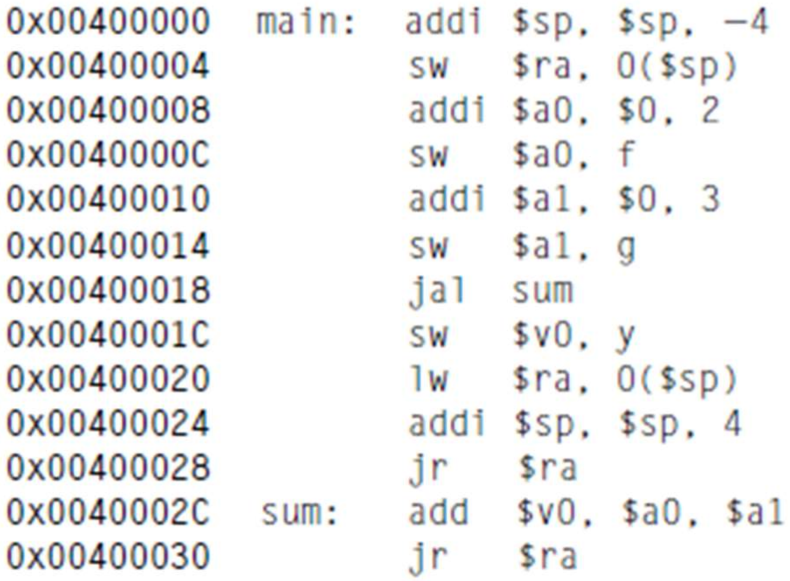
```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4   # stack frame
  sw   $ra, 0($sp)    # store $ra
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # call sum
  sw   $v0, y         # y = sum()
  lw   $ra, 0($sp)    # restore $ra
  addi $sp, $sp, 4    # restore $sp
  jr   $ra            # return to OS
sum:
  add  $v0, $a0, $a1  # $v0 = a + b
  jr   $ra            # return
```

Chapter 6 <14>

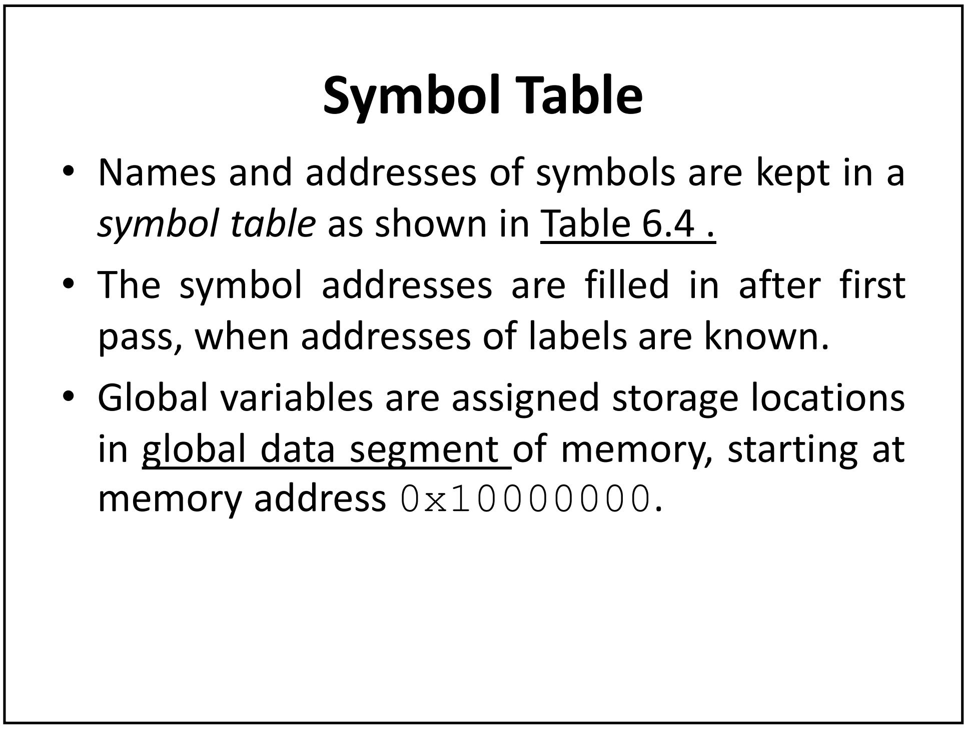
Step 2: Assembling

- The assembler turns assembly language code into *object file* containing machine lang code.
- The assembler makes two passes thro’ assembly code.
- On first pass, assembler assigns instruction addresses and finds all *symbols*, such as labels and global variable names.
- The code after first assembler pass is shown.

Assembling the code

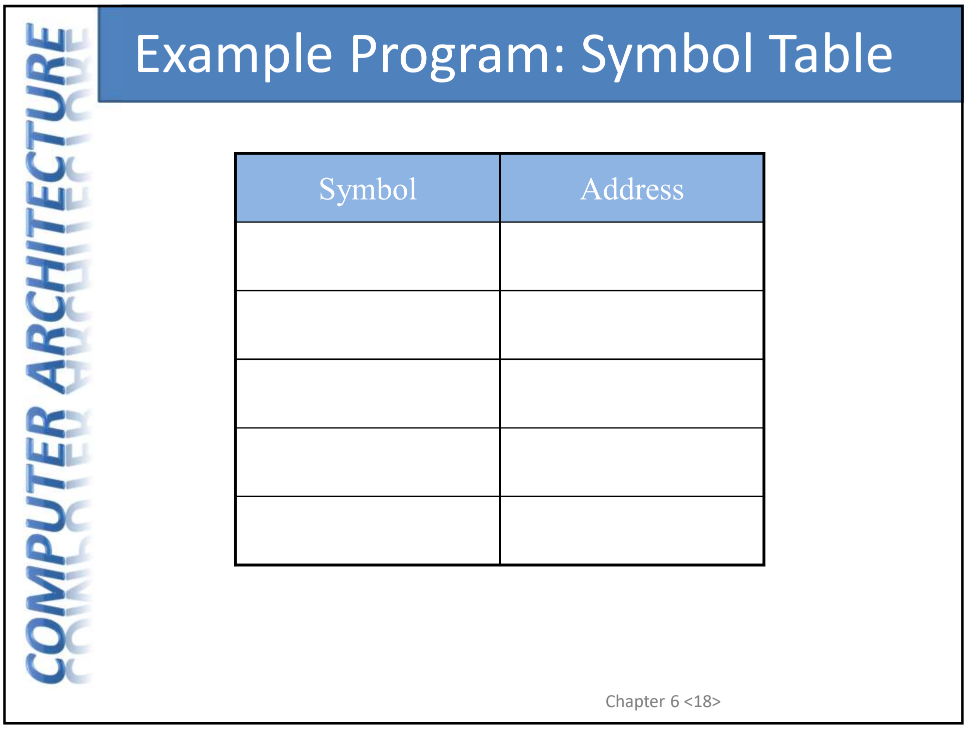


```
0x00400000  main:  addi $sp, $sp, -4
0x00400004         sw   $ra, 0($sp)
0x00400008         addi $a0, $0, 2
0x0040000C         sw   $a0, f
0x00400010         addi $a1, $0, 3
0x00400014         sw   $a1, g
0x00400018         jal  sum
0x0040001C         sw   $v0, y
0x00400020         lw   $ra, 0($sp)
0x00400024         addi $sp, $sp, 4
0x00400028         jr   $ra
0x0040002C  sum:   add  $v0, $a0, $a1
0x00400030         jr   $ra
```



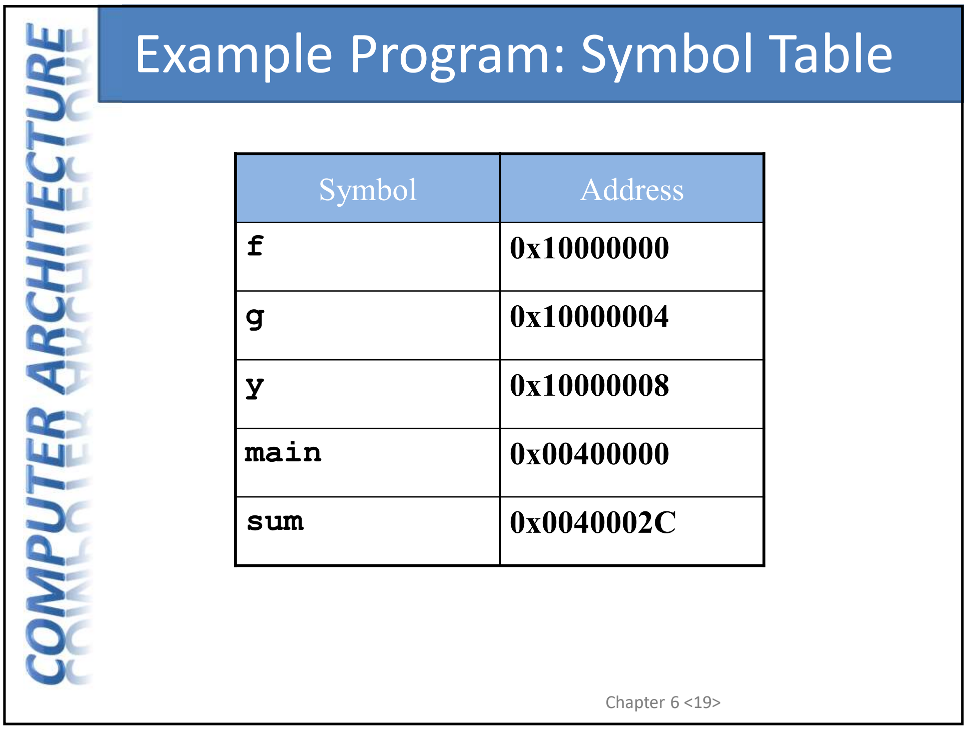
Symbol Table

- Names and addresses of symbols are kept in a *symbol table* as shown in Table 6.4 .
- The symbol addresses are filled in after first pass, when addresses of labels are known.
- Global variables are assigned storage locations in global data segment of memory, starting at memory address `0x10000000`.



Example Program: Symbol Table

Symbol	Address



COMPUTER ARCHITECTURE

Example Program: Symbol Table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Chapter 6 <19>

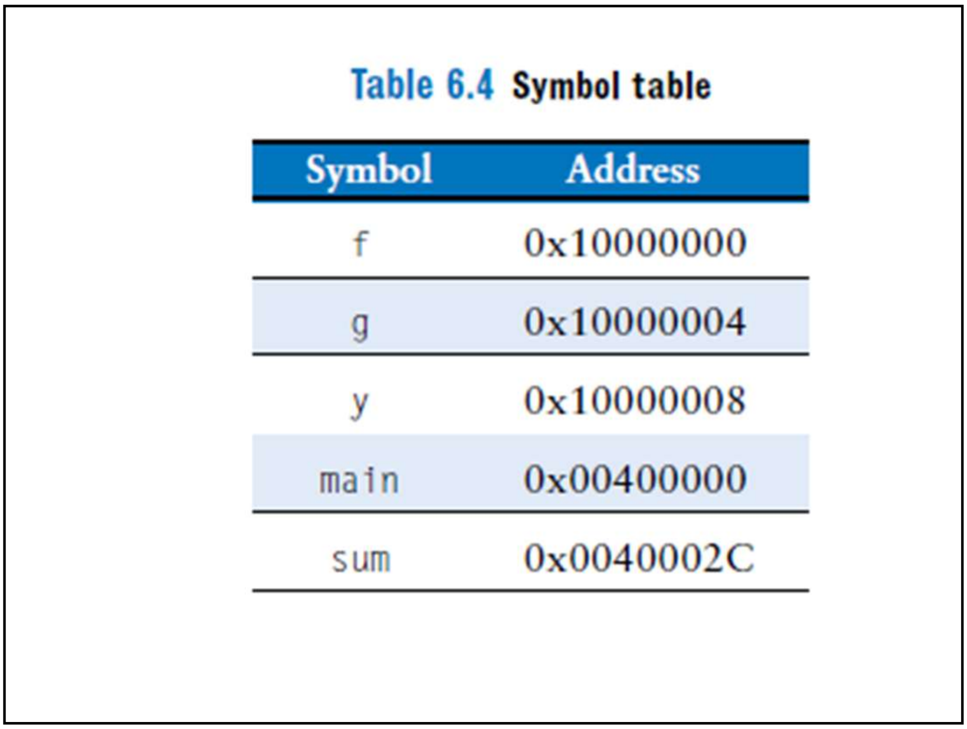
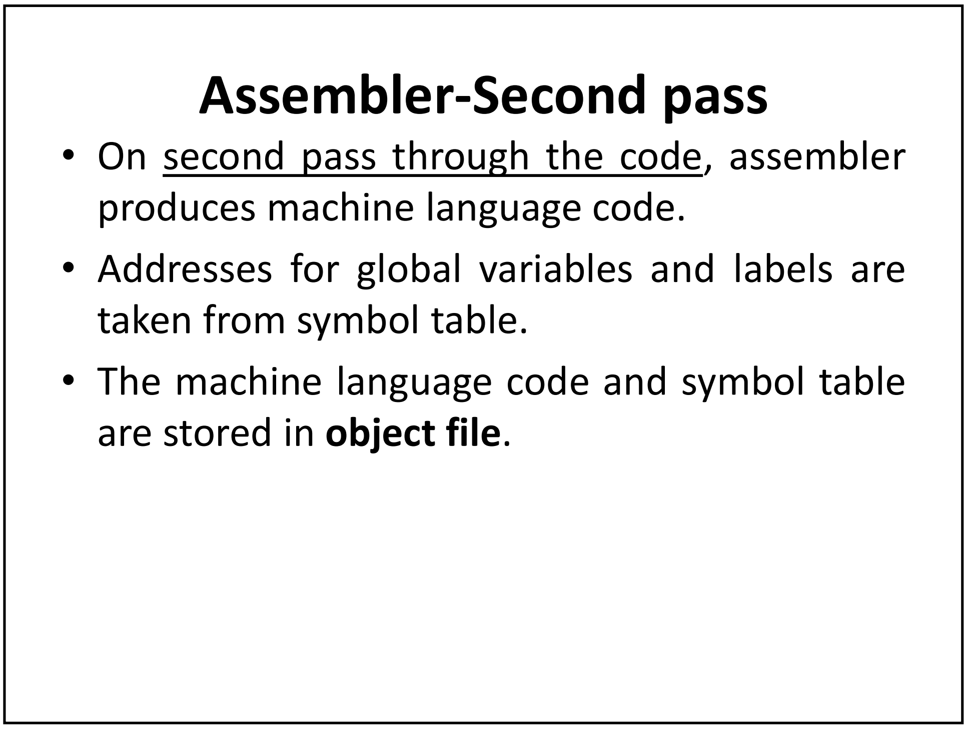


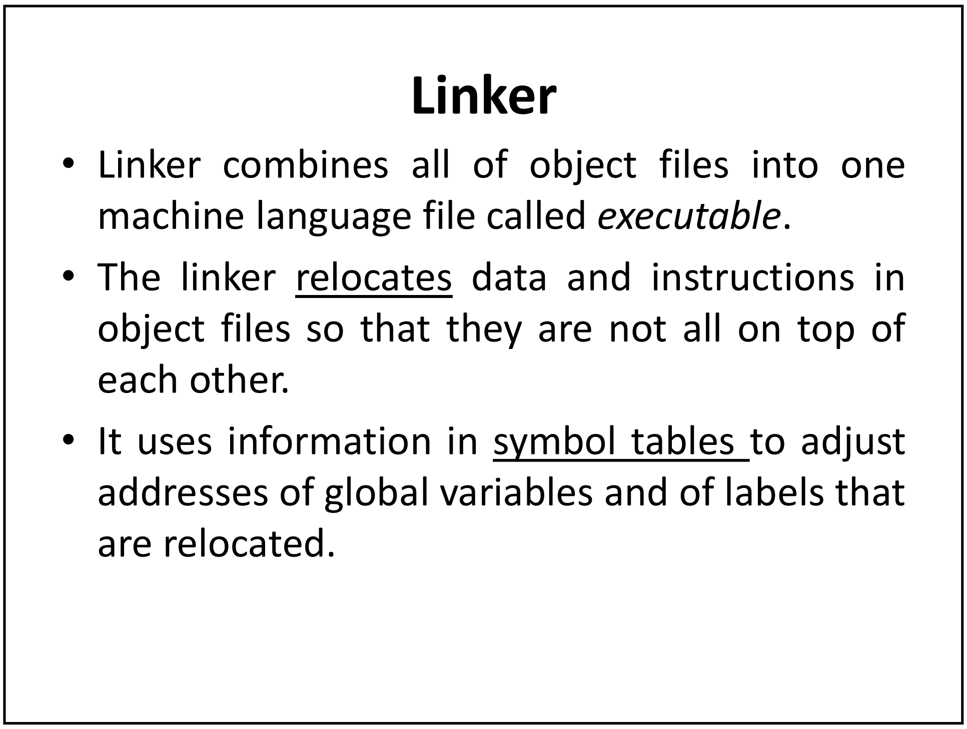
Table 6.4 Symbol table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C



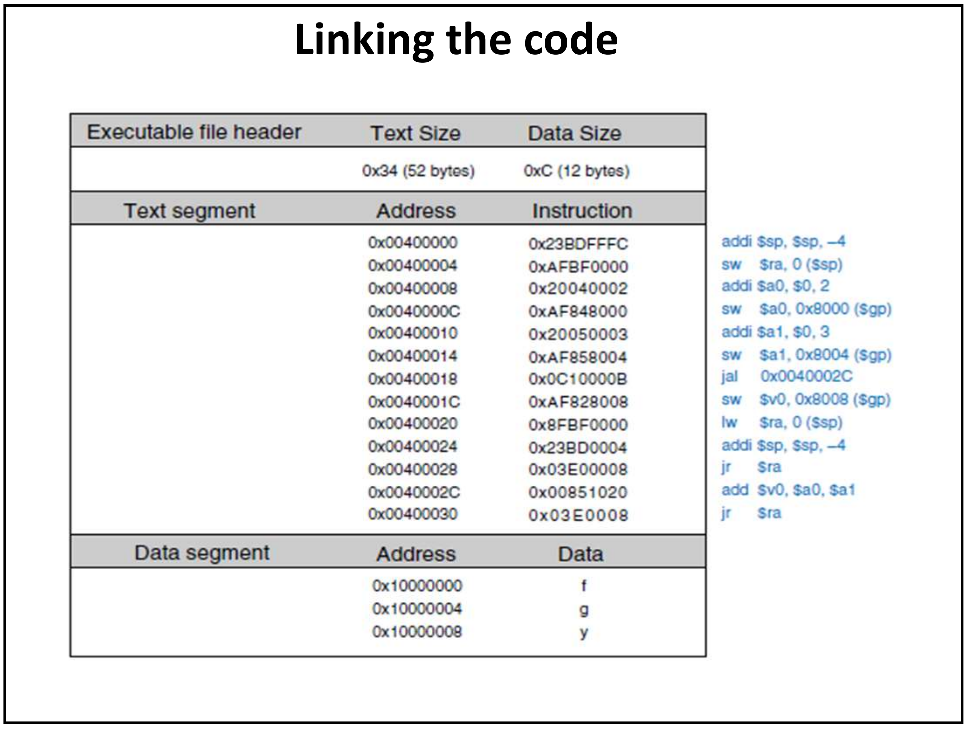
Assembler-Second pass

- On second pass through the code, assembler produces machine language code.
- Addresses for global variables and labels are taken from symbol table.
- The machine language code and symbol table are stored in **object file**.



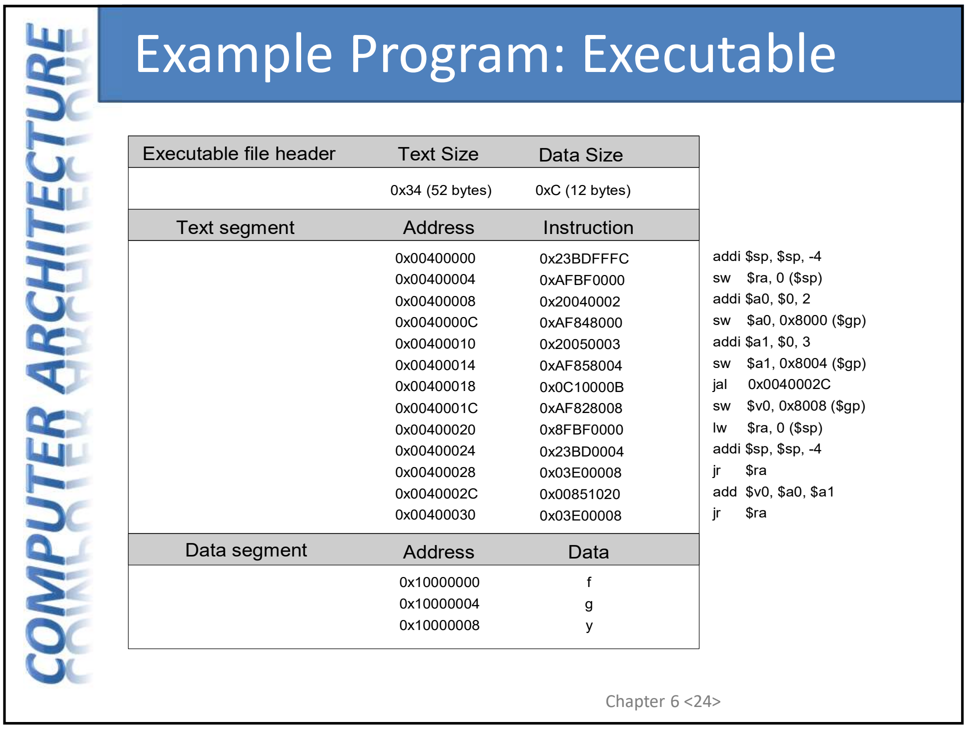
Linker

- Linker combines all of object files into one machine language file called *executable*.
- The linker relocates data and instructions in object files so that they are not all on top of each other.
- It uses information in symbol tables to adjust addresses of global variables and of labels that are relocated.



Linking the code

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	Address	Instruction	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E0008	jr $ra
Data segment	Address	Data	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	



Example Program: Executable

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	Address	Instruction	
	0x00400000	0x23BDFFFC	addi $sp, $sp, -4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, -4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	Address	Data	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

Chapter 6 <24>

Linking the code

- It has three sections: executable file header, text segment, and data segment.
- The executable file header reports text size (code size) and data size (amount of globally declared data).
- Both are given in units of bytes.
- The text segment gives instructions and addresses where they are to be stored.

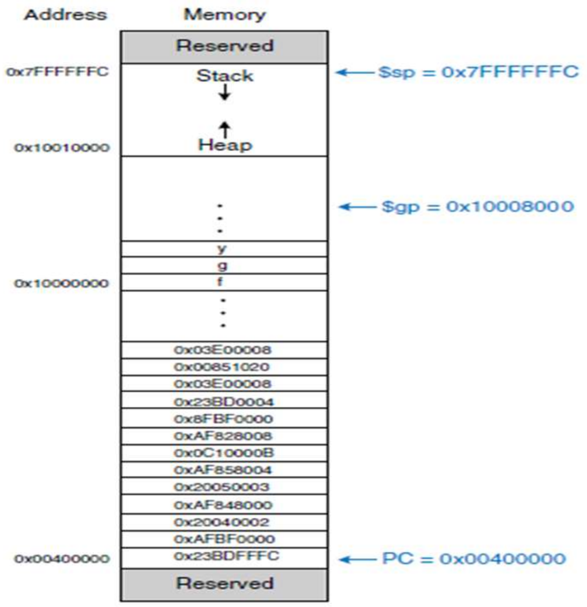
Linking

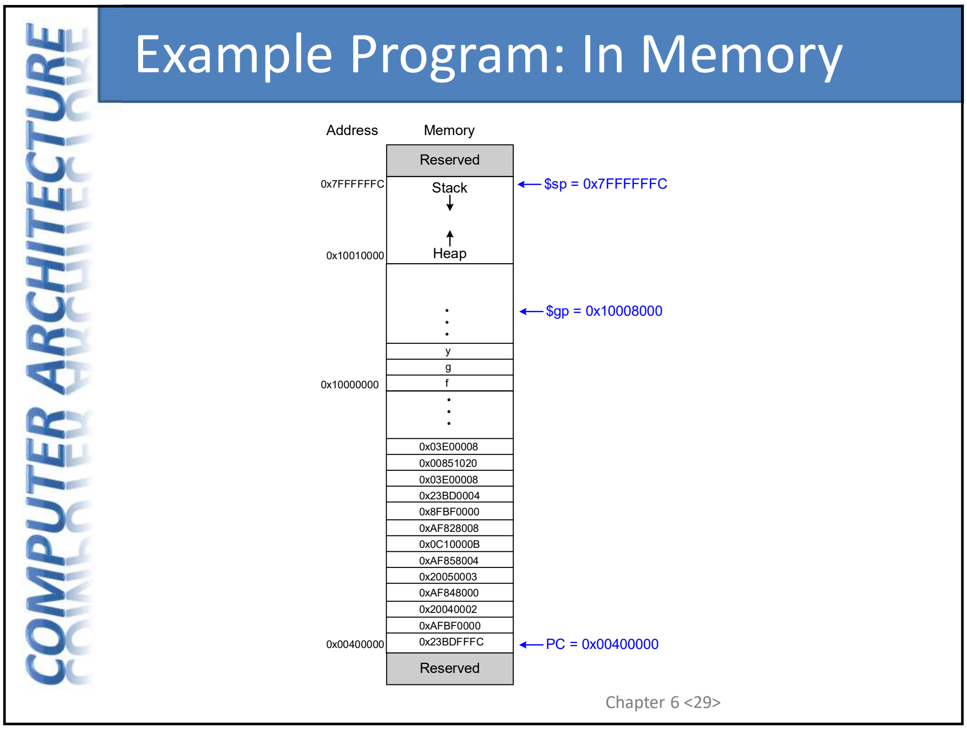
- Data segment gives addr of each global variable.
- Global variables are addressed wrt base addr given by global pointer, `$gp`.
- For ex, first store instruction,
  `sw $a0, 0x8000($gp)`
  stores value `2` to global var `f`, which is located at memory addr `0x10000000`.
- Offset, 0x8000, is a 16-bit signed num that is sign-extended and added to base address, `$gp`.
- So, `$gp + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000`, the memory address of variable `f`.

Step 4: Loading

- The OS <u>loads</u> a program by reading text segment of executable file from a storage device (usually the hard disk) into <u>text segment</u> of memory.
- The OS sets `$gp` to `0x10008000` (middle of global data segment) and `$sp` to `0x7FFFFFFC` (top of dynamic data segment), then performs a `jal  0x00400000` to jump to <u>beginning</u> of program.
- The diagram shows memory map at the beginning of program execution.

Loading the code





Example Program: In Memory

Address	Memory	
	Reserved	
0x7FFFFFFC	Stack ↓	$sp = 0x7FFFFFFC
0x10010000	↑ Heap	
	⋮	$gp = 0x10008000
	y	
	g	
0x10000000	f	
	⋮	
	0x03E00008	
	0x00851020	
	0x03E00008	
	0x23BD0004	
	0x8FBF0000	
	0xAF828008	
	0x0C10000B	
	0xAF858004	
	0x20050003	
	0xAF848000	
	0x20040002	
	0xAFBF0000	
0x00400000	0x23BDFFFC	PC = 0x00400000
	Reserved	

MIPS MICROARCHITECTURE

EC206 CO MODULE IV

Based on true facts from:

David Money Harris, Sarah L Harris, Digital Design and Computer Architecture, Morgan Kaufmann – Elsevier, 2009

P.RAJKUMAR Prof/ECE/NCERC
qraj2015@hotmail.com

- To begin with…
- Some more **Pioneers** of **Computer Organisation**!

John von Neumann (December 28, 1903 – February 8, 1957) was a Hungarian-American Mathematician, Physicist, Computer scientist, and Polymath. He made major contributions to a number of fields, including Mathematics, **Computing (Von Neumann architecture**, linear programming, self-replicating machines, stochastic computing), and statistics.

David Patterson is the Pardee Professor of Computer Science, Emeritus at the University of California at Berkeley, which he joined after graduating from UCLA in 1976. He championed Reduced Instruction Set Computers (RISC), Redundant Array of Inexpensive Disks (RAID), SPARC for SUN Microsystems and Networks of Workstations (NOW),each of which helped lead to billion dollar industries...

INTRODUCTION TO **MICROARCHITECTURE**

- Microarchitecture, is the connection between Logic and Architecture and is the specific arrangement of Registers, ALUs, Finite State Machines (FSMs), Memories, and other logic building blocks needed to implement an Architecture.
- A particular Architecture, such as MIPS, may have many different Microarchitectures, each with different trade-offs of Performance, Cost, and Complexity.
- They all run the same programs, but their internal designs vary widely.

Architectural State and Instruction Set

- Computer architecture is defined by its *Instruction Set* and *Architectural State*.
- The *Architectural State* for the MIPS Processor consists of the Program Counter and the 32 Registers.
- Any MIPS Microarchitecture must contain all of this State.
- Based on the current architectural state, the Processor executes a particular instruction with a particular set of data to produce a new architectural state.
- Some microarchitectures contain additional *non-architectural state* to either simplify the logic or improve performance.

Architectural State and Instruction Set

- Consider only a subset of the MIPS instruction set.
- R-type arithmetic/logic instructions: `add, sub, and, or, slt`
- Memory instructions: `lw, sw`
- Branches: `beq`
- After building the microarchitectures with these instructions, extend them to handle `addi` and `j`.
- These particular instructions were chosen because they are sufficient to write many interesting programs.

Design Process

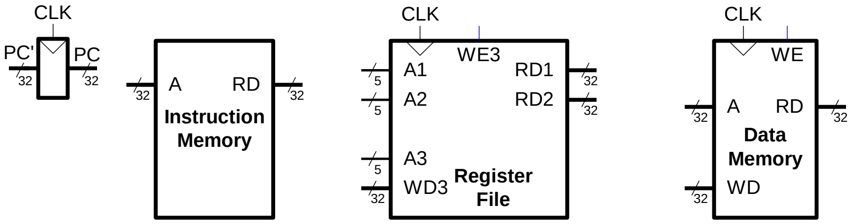
- Divide the Microarchitectures into two interacting parts:
- The *Datapath* and the *Control*.
- The Datapath operates on words of data.
- It *contains structures such as memories, registers,* ALUs, and *multiplexer*s.
- **MIPS** is a 32-bit architecture, so we will use a 32-bit datapath.
- The Control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction.
- Specifically, the Control unit produces *multiplexer select*, *register enable*, and *memory write signals* to control the operation of the datapath.

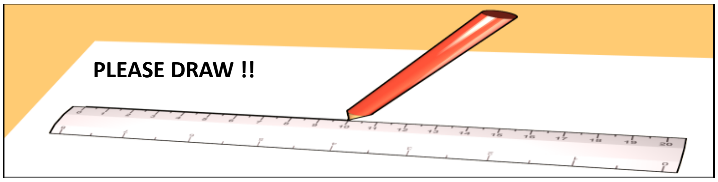
Design Process

- To start with hardware containing the state elements.
- These elements incl memories and architectural state (the program counter and registers).
- Then, add blocks of combinational logic between state elements to compute new state based on current state.
- The instruction is read from part of memory;
- `load` and `store` instructions then read or write data from another part of memory.
- Hence, to partition the overall memory into two smaller memories, one containing instructions and the other containing data.
- The diagram shows a block diagram with four state elements: the program counter, register file, and instruction and data memories.

MICROARCHITECTURE

MIPS State Elements





Design Process

- *Program Counter* is an ordinary 32-bit register.
- Its output, *PC*, points to the current instruction.
- Its input, *PC'*, indicates the address of the next instruction.
- The *Instruction memory* has a single read port.
- It takes a 32-bit instruction address input, *A*, and reads 32-bit data (i.e., instruction) from that address onto read data output, *RD*.
- The 32-element 32-bit *Register file* has 2 read ports and 1 write port.
- The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of 2^5 = 32-bit registers as source operands.
- They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively.
- The write port takes a 5-bit address input, *A3*; a 32-bit write data input, *WD*; a write enable input, *WE3*; and a clock.
- If write enable is 1, register file writes data into specified register on rising edge of the clock.

Design Process

- The *data memory* has a single read/write port.
- If write enable, *WE*, is 1, it writes data *WD* into address *A* on rising edge of clock.
- If write enable is 0, it reads address *A* onto *RD*.
- The instruction memory, register file, and data memory are all read *combinationally*.
- In other words, if address changes, new data appears at *RD* after some propagation delay; no clock is involved.
- They are written only on rising edge of clock.
- In this fashion, state of the system is changed only at clock edge.
- The address, data, and write enable must setup sometime before clock edge and must remain stable until a hold time after clock edge.

MIPS Microarchitectures

- There are three microarchitectures for the MIPS processor architecture: *Single-cycle*, *Multi-cycle*, and *Pipelined*.
- They differ in the way that the state elements are connected together and in the amount of non-architectural state.
- **Single-cycle Microarchitecture**
  - The *single-cycle microarchitecture* executes an entire instruction in one cycle.
  - It is easy to explain and has a simple control unit.
  - Because it completes the operation in one cycle, it does not require any non-architectural state.
  - However, the cycle time is limited by the slowest instruction.

MIPS Microarchitectures

- **Multi-cycle Microarchitecture**
  - The *multi-cycle microarchitecture* executes instructions in a series of shorter cycles.
  - Simpler instructions execute in fewer cycles than complicated ones.
  - Moreover, multi-cycle microarchitecture reduces hardware cost by reusing expensive hardware blocks such as adders and memories.
  - For ex, the adder may be used on several different cycles for several purposes while carrying out a single instruction.
  - The multi-cycle microprocessor accomplishes this by adding several non-architectural registers to hold intermediate results.
  - The multi-cycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles.

MIPS Microarchitectures

- **Pipelined Microarchitecture**
  - The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture.
  - It therefore can execute several instructions simultaneously, improving the throughput significantly.
  - Pipelining must add logic to handle dependencies between simultaneously executing instructions.
  - It also requires non-architectural pipeline registers.
  - The added logic and registers are worthwhile;
- All commercial high-performance Processors use Pipelining today.
- MIPS- **M**icroprocessor without **I**nterlocked **P**ipelining **S**tages
- SPARC- **S**calable **Processor ARC**hitecture
- ARM – **A**dvanced **R**ISC **M**achine

Test your understanding!

1.Compare and Contrast the types of MIPS Microarchitectures.

2. Which Microarchitectures need Non-architectural elements and Why?

3. State the main parts of the Microarchitecture and provide their features.

4. Define Microarchitecture and clarify whether a specific Architecture can support more than one Microarchitecture.

5. Construct the State Elements with the aid of well labelled diagrams and provide their operation.

PERFORMANCE ANALYSIS

- The execution time of a program, measured in seconds, is given by:

$$Execution\ Time = \left(\#\ instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

- No: of instructions for each program is constant, independent of microarchitecture.
- No: of Cycles Per Instruction, often called **CPI**, is the no: of clock cycles required to execute an average instruction.

CPI (Cont…)

- It is the reciprocal of the throughput (Instructions Per Cycle, or IPC).
- Different microarchitectures have different CPIs.
- Assume an ideal memory system that does not affect the CPI.
- Multi-cycle data path and control use the term **CPI(M)**, which indicates mean clock cycles per instruction.

Clock Period T_c

- The number of seconds per Cycle is the Clock Period, *Tc*.
- The clock period is determined by critical path through logic on processor.
- Different microarchitectures have different clock periods.
- Logic and circuit designs also significantly affect clock period.
- For ex , a carry-look ahead adder is faster than a ripple-carry adder.

Challenge for the Microarchitect

- Choose the design that minimizes execution time while satisfying constraints on cost and/or power consumption.
- Microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, so determining the best choice requires careful analysis.
- There are many other factors that affect overall computer performance.
- For ex , hard disk, memory, graphics system, and network connection may be limiting factors that make processor performance irrelevant.

Test your understanding!

1. Identify the various quantities that can be used for performance analysis and give their definitions.
2. Define and quantify the following: PET, CPI, IPC, CPI(M), CP.
3. Explain the challenges faced by the designer of Microarchitecture.

SINGLE-CYCLE PROCESSOR

Design a MIPS microarchitecture that executes instructions in a single cycle in following stages:

1. Construct Datapath by connecting State Elements with combinational logic that can execute various instructions.
2. Control signals determine which specific instruction is carried out by datapath at any given time.
3. Controller contains combinational logic that generates appropriate control signals based on current instruction.
4. Then do Performance Analysis.

Single-Cycle Datapath

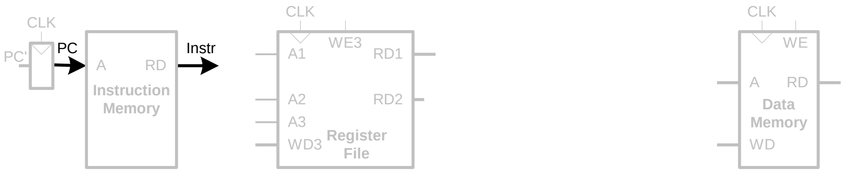
Develop single-cycle datapath, adding one piece at a time to State Elements.

New connections are shown in black, while h/w that has already been studied is shown in gray, with control signals shown in blue.

Step-1:**Fetch Instruction from memory**

- Program Counter (PC) contains address of current instruction to execute.
- First step is to read this instruction from Instruction Memory.
- Program Counter (PC) is simply connected to address i/p of Instruction Memory.
- Instruction Memory reads out, or *Fetches*, the 32-bit instruction, labeled `Instr`.

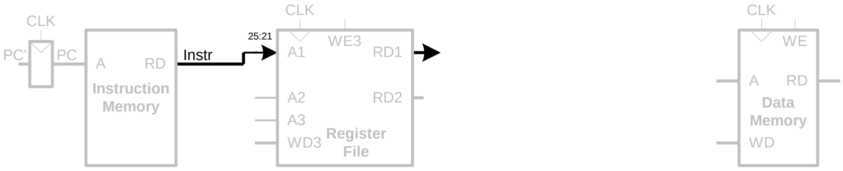
Step: 1: Fetch Instruction from memory



Step: 2: Read Source Operand from Register File

- The processor's actions depend on <u>specific instruction</u> that was fetched.
- First work out Datapath connections for `lw` instruction.
- Generalize Datapath to handle other instructions.
- For a `lw` instruction, next step is to read source register containing base address.
- This reg is in `rs` field of instruction, *Instr*25:21.
- These bits of instruction are connected to address i/p of one of Register File read ports, *A1*.
- Register file reads register value onto *RD1*.

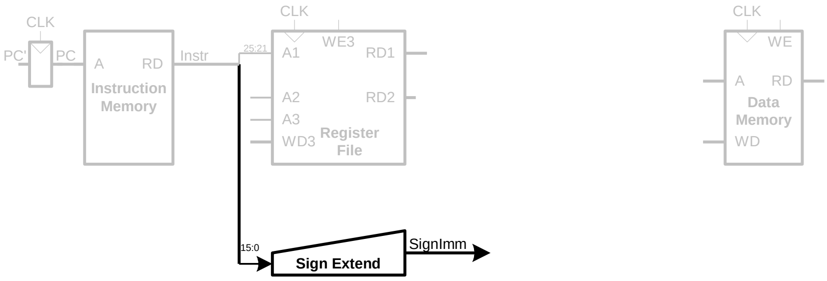
Step: 2: Read Source Operand from Register File



Step: 3: Sign Extend the Immediate

- The `lw` instruction also requires an offset.
- The offset is stored in immediate field of instruction, $Instr_{15:0}$.
- The 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits.
- The 32-bit sign-extended value is called *SignImm*.
- Sign Extension simply copies sign bit (msb) of a short input into all of upper bits of longer output.
- $SignImm_{15:0} = Instr_{15:0}$ and $SignImm_{31:16} = Instr_{15}$.

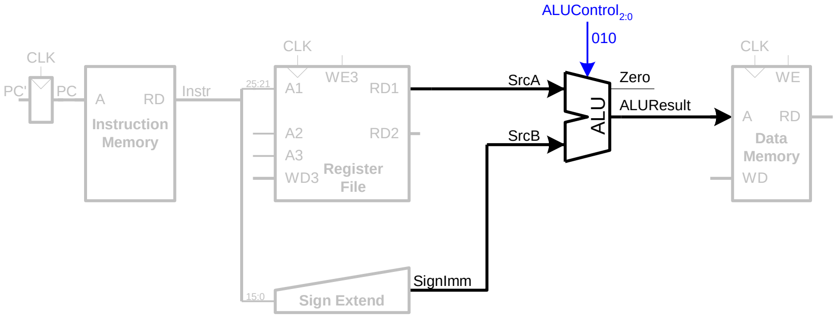
Step: 3: Sign Extend the Immediate



Step: 4: Compute the Memory Address

- Processor must add base address to offset to find effective address (EA) to read from memory using ALU.
- ALU receives two operands, *SrcA* and *SrcB*.
- *SrcA* comes from register file, and *SrcB* comes from sign-extended immediate.
- The ALU can perform many operations.
- The 3-bit *ALUControl* signal specifies the operation.
- The ALU generates a 32-bit *ALUResult* and a *Zero* flag, that indicates whether *ALUResult* 0.
- For a `lw` instruction, the *ALUControl* signal should be set to `010` to add the base address and offset.
- *ALUResult* is sent to the data memory as the address for the load instruction.

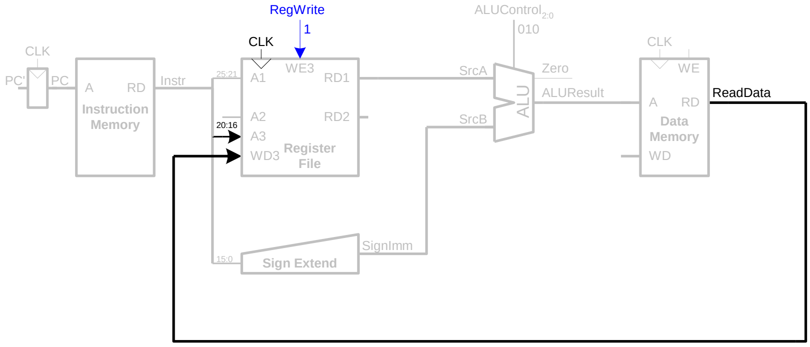
Step: 4: Compute the Memory Address



Step: 5: Read data from memory and write it back to register file

- The data is read from data memory onto *ReadData* bus, then written back to destn register in register file at end of cycle.
- Port 3 of register file is write port.
- Destn register for `lw` instruction is specified in `rt` field, $Instr_{20:16}$, which is connected to port 3 address input, *A3*, of register file.
- The *ReadData* bus is connected to port 3 write data input, *WD3*, of register file.
- A control signal called *RegWrite* is connected to port 3 write enable input, *WE3*, and is asserted during a `lw` instruction so that the data value is written into register file.
- The write takes place on rising edge of the clock at end of cycle.

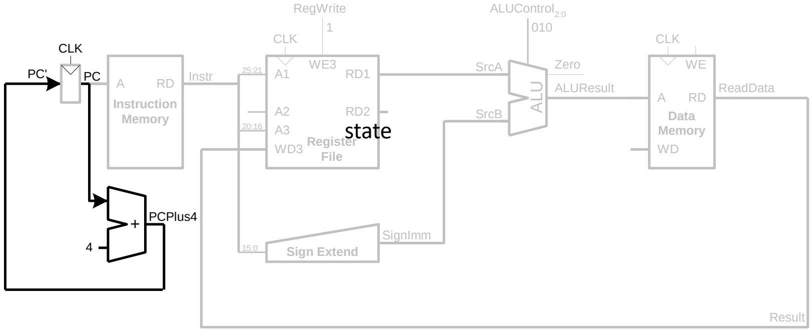
Step: 5: Read data from memory and write it back to register file



Step: 6: Determine Address of next Instruction

- While instruction is being executed, processor must compute address of next instruction, *PC'*.
- Because instructions are 32 bits= 4 Bytes, next instruction is at *PC+* 4.
- Another adder used to increment the *PC* by 4.
- The new address is written into program counter on next rising edge of the clock.
- This completes the datapath for `lw` instruction.

Step: 6: Determine Address of next Instruction



35

Datapath for the `sw` instruction

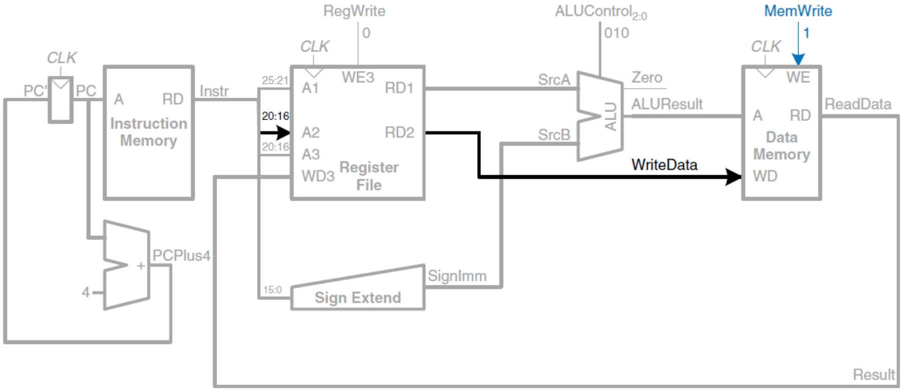
- `sw` instruction reads a base address from port 1 of register and sign-extends an immediate.
- ALU adds base address to immediate to find effective memory address.
- All of these functions are already supported by the datapath.
- The sw instruction also reads a second register from register file and writes it to data memory.
- The register is specified in the `rt` field, $Instr_{20:16}$.
- These bits of the instruction are connected to the second register file read port, *A2*.

36

Datapath for the **sw** instruction

- The register value is read onto the *RD2* port.
- It is connected to the write data port of the data memory.
- The write enable port of the data memory, *WE*, is controlled by *MemWrite*.
- For a sw instruction, `MemWrite=1`, to write the data to memory; `ALUControl=010`, to add the base address and offset;
- `RegWrite=0`, because nothing should be written to the register file.

Write Data to Memory for **sw** Instruction



Datapath for `R-type` instructions

- Extend datapath to handle `R-type instructions add, sub, and, or, slt`.
- Instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file.
- They differ only in the specific ALU operation.
- Hence, they can all be handled with the same hardware, using different *ALUControl* signals.
- Enhanced datapath handling R-type instructions.
- The register file reads two registers.
- The ALU performs an operation on these two registers.
- Add a multiplexer to choose *SrcB* from either the register file *RD2* port or *`SignImm`*.

Datapath for `R-type` instructions

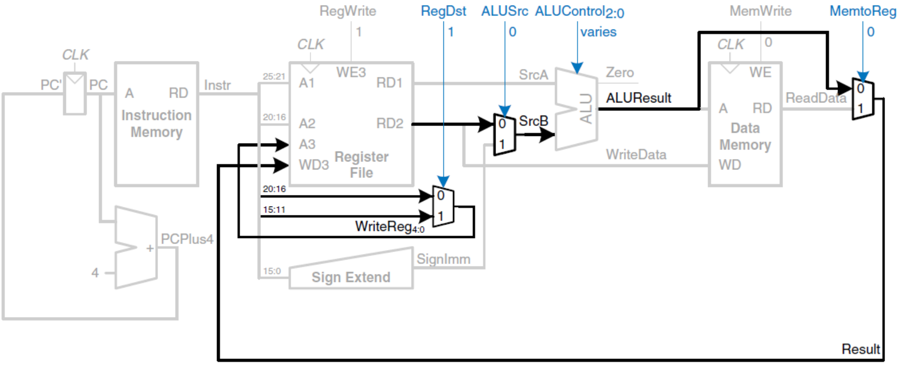
- The MUX is controlled by a new signal, *`ALUSrc`*.
- *`ALUSrc`* `is 0` for R-type instructions to choose *SrcB* from the register file;
- it is 1 for `lw` and `sw` to choose *`SignImm`*.
- `R-type` instructions write the *`ALUResult`* to register file.
- Add another MUX to choose between *`ReadData`* and *`ALUResult`*.
- Call its output *`Result`*.
- This MUX is controlled by another new signal, *`MemtoReg`*.
- *`MemtoReg`* is 0 for R-type instructions to choose *`Result`* from the *`ALUResult`*;
- it is 1 for `lw` to choose *`ReadData`*.

Datapath for `R-type` instructions

- For `R-type` instructions, the register is specified by the `rd` field, $Instr_{15:11}$.
- Add a third MUX to choose *WriteReg* from the appropriate field of the instruction.
- The MUX is controlled by *RegDst*.
- *RegDst* is 1 for `R-type` instructions to choose *WriteReg* from `rd` field, $Instr_{15:11}$;
- it is 0 for `lw` to choose `rt` field, $Instr_{20:16}$.

41

Datapath enhancement for `R-type` Instructions

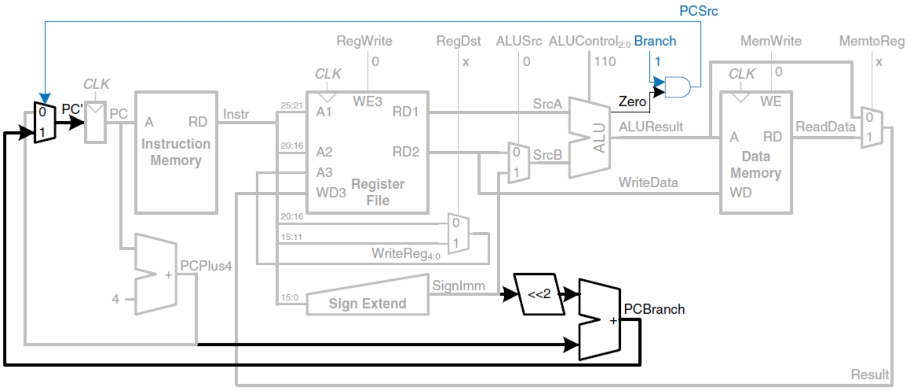


42

Datapath for beq instruction

- Extend datapath to handle beq.
- beq compares two registers.
- If they are equal, it takes the branch by adding branch offset to program counter.
- Offset is a positive or negative number, stored in imm field of instruction, $Instr_{31:26}$.
- The offset indicates the number of instructions to branch past.
- Hence, immediate must be sign-extended and multiplied by 4 to get new program counter value:
- $PC' = PC+ 4 + SignImm \times 4.$

Datapath enhancement for beq Instruction



Datapath for `beq` instruction

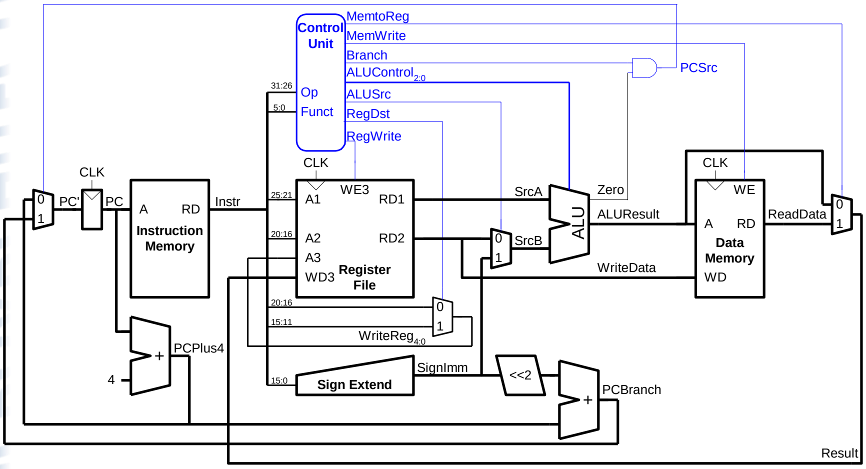
- The next *PC* value for a taken branch, *PCBranch*, is computed by shifting `SignImm` left by 2 bits, then adding it to `PCPlus4`.
- The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires.
- The two registers are compared by computing `SrcA – SrcB` using the ALU.
- If `ALUResult` is 0, as indicated by the *Zero* flag from the ALU, the registers are equal.
- We add a MUX to choose *PC* from either `PCPlus4` or `PCBranch`.

Datapath for beq instruction

- `PCBranch` is selected if the instruction is a branch and the *Zero* flag is asserted.
- Hence, *Branch* is 1 for beq and 0 for other instructions.
- For beq, *ALUControl* 110, so the ALU performs a subtraction.
- *ALUSrc* 0 to choose *SrcB* from the register file.
- *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory.

MICROARCHITECTURE

Single-Cycle Processor

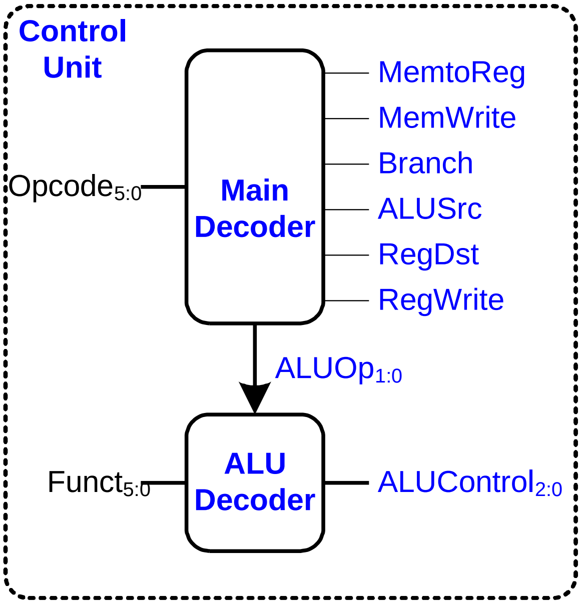


Chapter 7 <47>

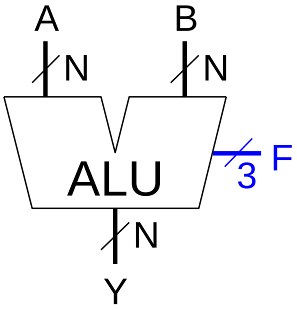
Test your understanding!

1. Analyze the Data Path of the MIPS single cycle processor for the `lw` instruction with the aid of neatly labelled diagrams.
2. Construct the Data Path of the MIPS single cycle processor for the `sw` instruction with the aid of neatly labelled diagram.
3. Construct the Data Path of the MIPS single cycle processor for the `R-type` instruction with the aid of neatly labelled diagram.
4. Construct the Data Path of the MIPS single cycle processor for the `beq` instruction with the aid of neatly labelled diagram.

Single-Cycle Control

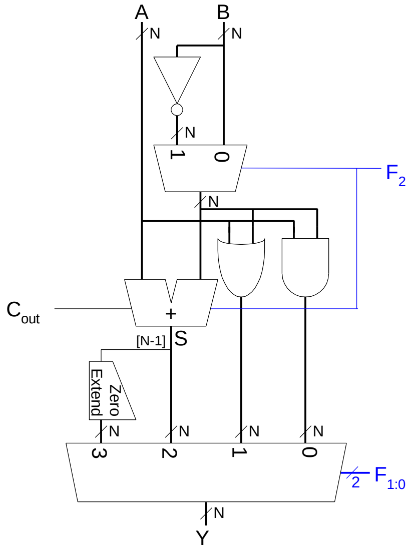


Review of ALU…



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Review of ALU Internals…



51

Control Unit- ALU Decoder

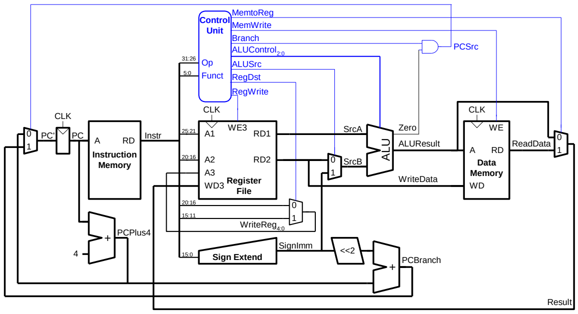
$\text{ALUOp}_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

$\text{ALUOp}_{1:0}$	Funct	$\text{ALUControl}_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

52

Control Unit Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	0	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Test your understanding!

1. Construct the Control Unit of the MIPS Signle Cycle Processor.
2. Apply the Control Unit Main Decoder for the, `lw, sw, R-type` and `beq` instructions of MIPS.

Performance Analysis

- `Program Execution Time`

  `=(#instructions)x (cycles/instruction) x(seconds/cycle)`

  `= (# instructions) x CPI x` T_C
- T_C limited by critical path (`lw`)

Performance Analysis

- Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1.
- Critical path for `lw` instruction starts with PC loading a new address on rising edge of the clock.
- Instruction memory reads next instruction.
- The register file reads *SrcA*.
- While the register file is reading, the immediate field is sign-extended and selected at the *ALUSrc* MUX to determine *SrcB*.
- The ALU adds *SrcA* and *SrcB* to find the effective address.
- The data memory reads from this address.
- The *MemtoReg* MUX selects *ReadData*.
- Finally, *Result* must setup at the register file before the next rising clock edge, so that it can be properly written.

Single Cycle Performance

- Hence, the cycle time is:
- Single-cycle critical path:

  $T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$

- Typically, limiting paths are:
  - ❑Memory, ALU, Register file (accesses are slower…)
  - ❑$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Build a single-cycle MIPS processor in a 65 nm CMOS manufacturing process. The logic elements have the delays given in Table. Compute the execution time for a program with 100 billion instructions. Assume that CPI=1.

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

T_c = ?

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

$$
\begin{aligned}
T_c &= t_{pcq\_PC} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}} \\
&= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\
&= 925 \text{ ps}
\end{aligned}
$$

Single-Cycle Performance Example

Program with 100 billion instructions:

$$
\begin{aligned}
\textbf{Execution Time} &= (\#\text{ instructions}) \text{ x CPI x } T_C \\
&= (100 \times 10^{9})(1)(925 \times 10^{-12} \text{ s}) \\
&= \textbf{92.5 seconds}
\end{aligned}
$$

Test your understanding!

1. Analyse the Performance of the MIPS Single Cycle Processor.
2. From first principles, estimate the PET of a MIPS Single Cycle Processor, using the same delay table as before, for a given program with:

(i) 10 lakh instructions

(ii) 10 crore instructions

Multi-Cycle Microarchitecture

Module IV (Cont…)

Problems with Single-cycle Processor

- The single-cycle processor has three primary drawbacks.
  - **1**. It requires a clock cycle long enough to support slowest instruction (`lw`), even though most instructions are faster.
  - **2**. It requires three adders (one in the ALU and two for the PC logic), adders are relatively expensive circuits, especially if they must be fast.
  - **3**. It has separate Instruction and Data Memories, which may not be realistic.
- Most computers have a single large memory that holds both instructions and data and that can be read and written.

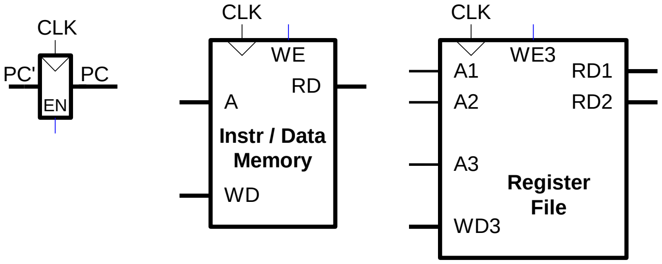
Multi-cycle Processor

- Multicycle processor breaks an instruction into multiple shorter steps.
- In each short step, the processor can read or write the memory or register file or use the ALU.
- Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones.
- The processor needs only one adder; this adder is reused for different purposes on various steps.
- And the processor uses a combined memory for instructions and data.
- The instruction is fetched from memory on the first step, and data may be read or written on later steps.

Design of Multi-cycle Processor

- First, construct a Datapath by connecting the architectural state elements and memories with combinational logic.
- But, this time, also add non-architectural state elements to hold intermediate results between the steps.
- Then design the Controller.
- The controller produces different signals on different steps during execution of a single instruction, so it is now a finite state machine rather than combinational logic.
- Examine how to add new instructions to the processor.
- Finally, analyze the performance of the multicycle processor and compare it to the single-cycle processor.

Replace Instruction and Data memories with a single unified memory – more realistic

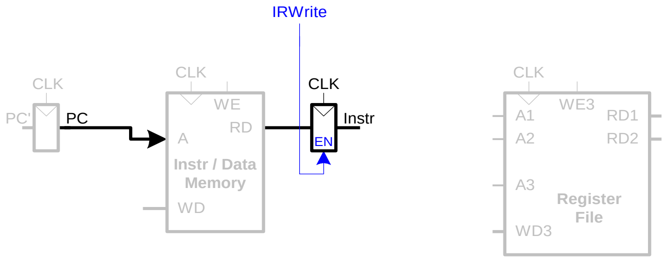


Multi-cycle Datapath

Test your understanding!

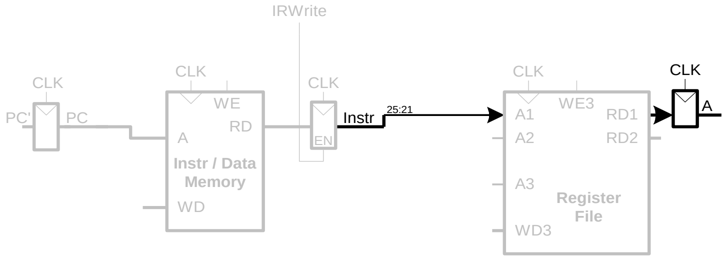
1. Explain the drawbacks of single cycle micro-architecture.
2. How does multi cycle micro-architecture work?
3. Explain the design of multi cycle micro-architecture with use of modified state elements.

Step-1: Fetch Instruction



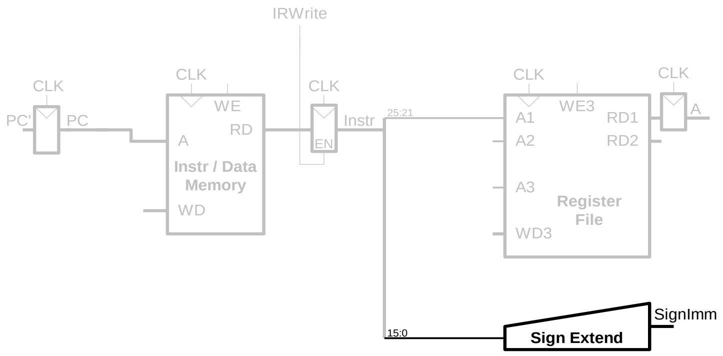
Multi-cycle Datapath

Step-2a: Read Source Operands from Reg File



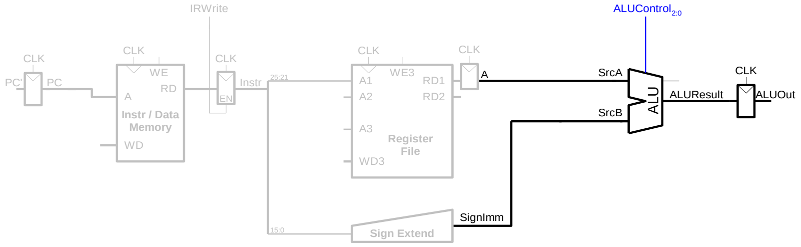
Multi-cycle Datapath: `lw` Instruction

Step-2b: Sign Extend the Immediate



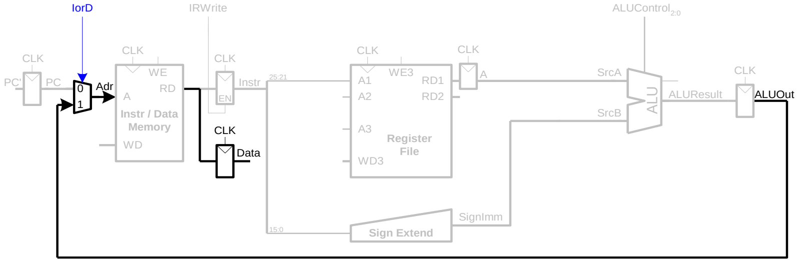
Multi-cycle Datapath: `lw` Instruction

Step-3: Compute the Memory Address



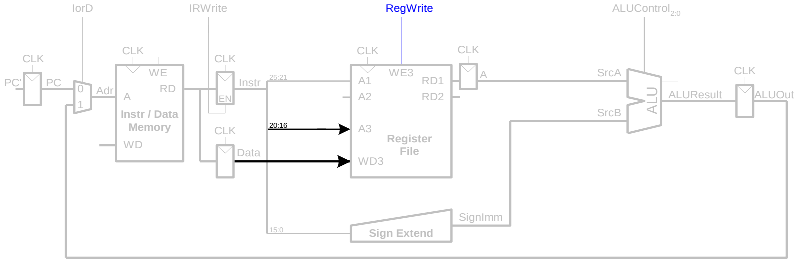
Multi-cycle Datapath: `lw` Address

Step-4: Read Data from Memory



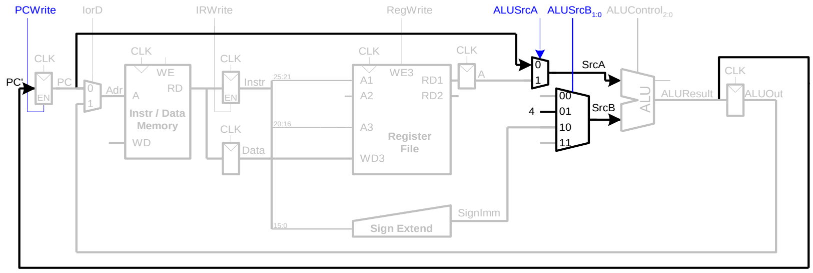
Multi-cycle Datapath: `lw` Memory Read

Step-5: Write Data back to Register File



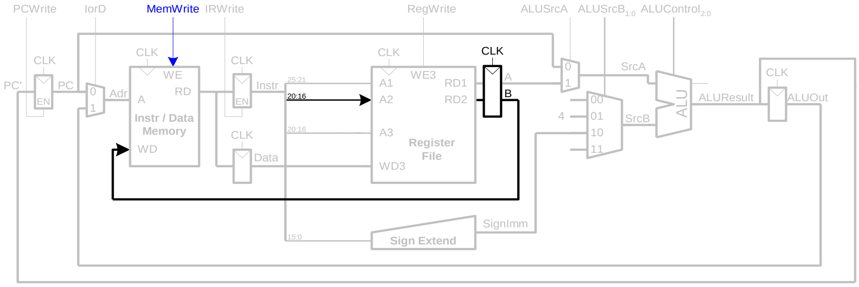
Multi-cycle Datapath: `lw` Write Register

Step-6: Increment PC



13

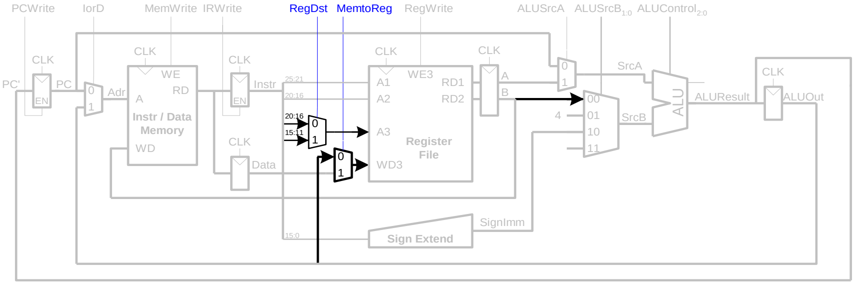
Write Data in rt to Memory



Multi-cycle Datapath: sw Instruction

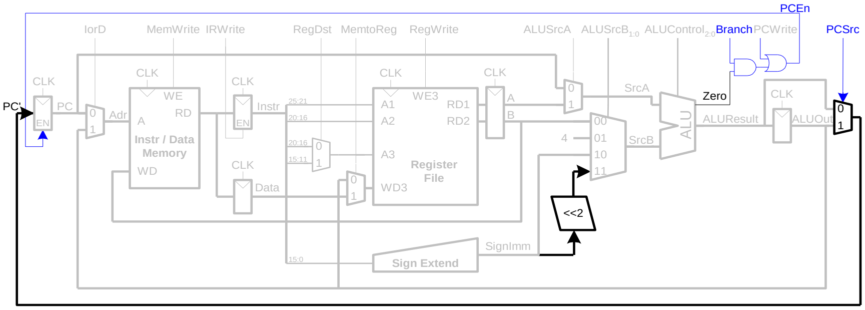
14

- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



Multi-cycle Datapath: R-type Instruction

- `rs == rt`? → Check the condition
- BTA = (sign-extended immediate << 2) + (PC+4)

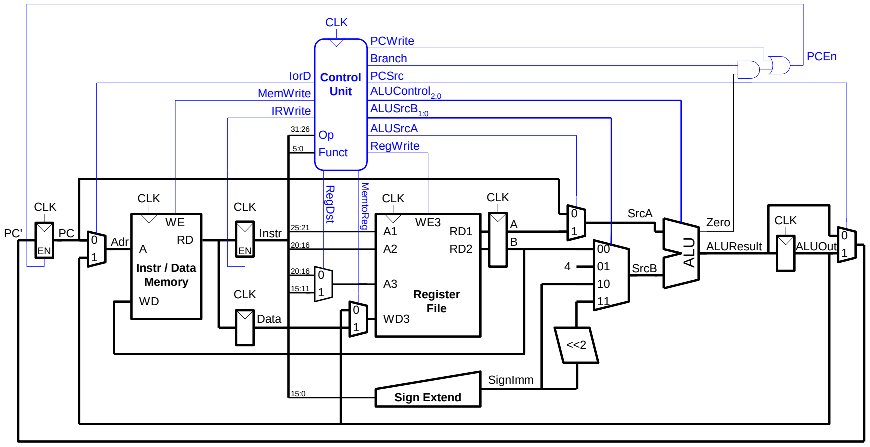


Multi-cycle Datapath: `beq` Instruction

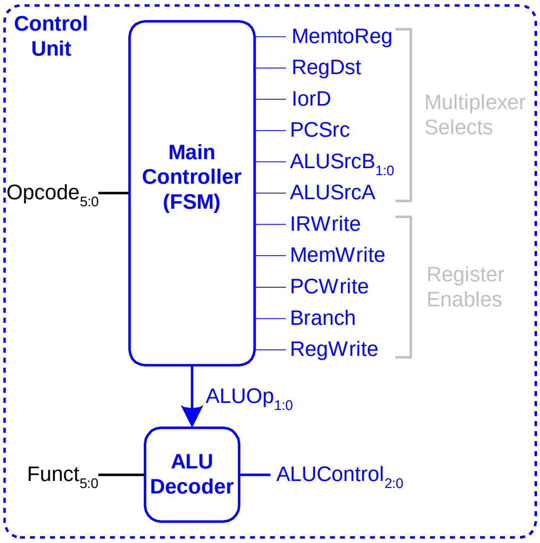
Test your understanding!

1. Analyze the Data Path of the MIPS multi cycle processor for the `lw` instruction with the aid of neatly labelled diagrams.
2. Construct the Data Path of the MIPS multi cycle processor for the `sw` instruction with the aid of neatly labelled diagram.
3. Construct the Data Path of the MIPS multi cycle processor for the `R-type` instruction with the aid of neatly labelled diagram.
4. Construct the Data Path of the MIPS multi cycle processor for the `beq` instruction with the aid of neatly labelled diagram.

Multi-cycle Processor



Multi-cycle Control



Test your understanding!

1. Construct the Control Unit of the MIPS Multi Cycle Processor.
2. Apply the Control Unit Main Decoder for the, `lw, sw, R-type` and `beq` instructions of the MIPS Multi Cycle Processor.

Multi-cycle Microarchitecture Performance Analysis

- The Execution Time of an instruction depends on both the number of cycles it uses and the cycle time.
- Single-cycle processor performed all instructions in one cycle, but Multi-cycle processor uses varying numbers of cycles for various instructions.
- However, the multi-cycle processor does less work in a single cycle and, so, has a shorter cycle time.
- The multicycle processor requires three cycles for `beq` and `j` instructions, four cycles for `sw`, `addi`, and `R-type` instructions, and five cycles for `lw` instructions.
- The CPI depends on the relative likelihood that each instruction is used.

Example: **MULTI-CYCLE PROCESSOR CPI(M)**

The **SPECINT2000 benchmark** consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this benchmark or CPI(M).

Sol:

- The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used.
- For this benchmark, Average CPI = (0.11+ 0.02)(3) + (0.52+ 0.10)(4) + (0.25)(5) = 4.12.
- This is better than the worst-case CPI of **5**, which would be required if all instructions took the same time. (Single-Cycle)
- Therefore, CPI(M)= 4.12

Performance Analysis

- The design of the multicycle processor is such that each cycle involved one ALU operation, memory access, or register file access.
- Assume the register file is faster than the memory and that writing memory is faster than reading memory.
- The Cycle Time is given by:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

- The numerical values of these times will depend on the specific implementation technology.

Example **PROCESSOR PERFORMANCE COMPARISON**

Is it better building the multicycle processor instead of the single-cycle processor? For both designs, a 65 nm CMOS manufacturing process with the delays given in Table. Compare each processor's execution time for 100 billion instructions from the **SPECINT2000 benchmark**.

MICROARCHITECTURE

Multi-cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$T_c = ?$

Multi-cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

$$\begin{aligned} \boldsymbol{T_c} &= t_{pcq\_PC} + t_{\mathrm{mux}} + \max(t_{\mathrm{ALU}} + t_{\mathrm{mux}}, t_{\mathrm{mem}}) + t_{\mathrm{setup}} \\ &= t_{pcq\_PC} + t_{\mathrm{mux}} + t_{\mathrm{mem}} + t_{\mathrm{setup}} \\ &= [30 + 25 + 250 + 20]\ \mathrm{ps} \\ &= \mathbf{325\ ps} \end{aligned}$$

Multi-cycle Performance Example

Program with **100 billion instructions**

Execution Time = ?

Program with 100 billion instructions

Execution Time = (# instructions) × CPI(M) × T_c

$$= (100 \times 10^{9})(4.12)(325 \times 10^{-12})$$

$$= \mathbf{133.9\ seconds}$$

This is **slower** than the single-cycle processor (92.5 seconds). Why?

Comparison

- A multi-cycle processor avoids making all instructions take as long as the slowest one.
- But, this example shows that the multicycle processor is slower than the single-cycle processor .
- The fundamental problem is that even though the slowest instruction, `lw`, was broken into five steps, the multi-cycle processor cycle time was not nearly improved fivefold.
- Not all of the steps are exactly the same length, and also the 50-ps sequencing overhead of register clk-to-Q and setup time must now be paid on every step, not just once for entire instruction.

Comparison

- Compared with the single-cycle processor, the multicycle processor is likely to be less expensive because it eliminates two adders and combines the instruction and data memories into a single unit.
- It does, however, require five non-architectural registers and additional multiplexers.

Test your understanding!

1. Analyse the Performance of the MIPS Multi Cycle Processor.

2. Processor performance comparison using **SPECINT2000 benchmark** with same table of delays for :

(i) 10 lakh instructions

(ii) 10 crore instructions

3. Compare and Contrast the performance of Multi Cycle processor with Single Cycle processor, highlighting the pros and cons.

INPUT/OUTPUT (I/O) System

Module V

Based on :

Carl Hamacher "Computer Organization", MGH

Introduction to I/O

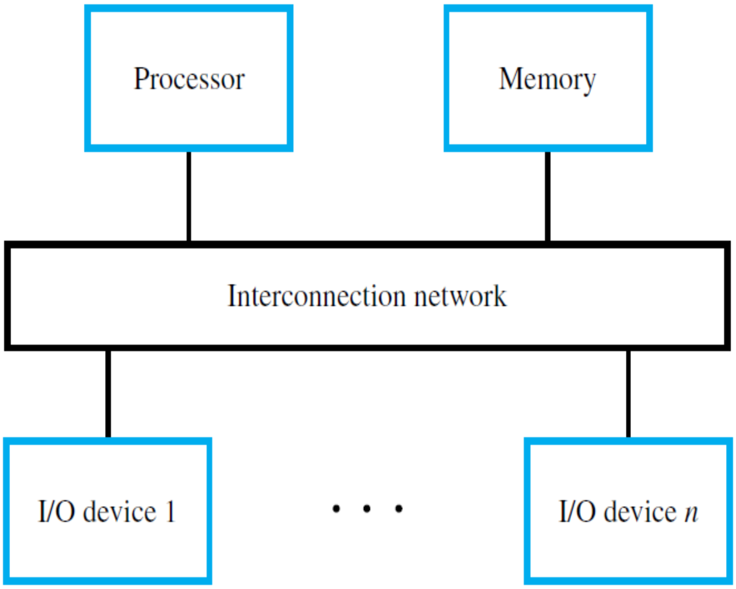
- Input to a Computer may come from Keyboard, Mouse, a Touch panel, a Sensor switch, a Digital camera, a Microphone, or a Fire alarm.
- Output may be to a Printer, Sound signal sent to a Speaker, or a digitally coded command that changes the speed of a Motor, opens a Valve, or causes a Robot to move in a specified manner.
- Computers should have the ability to exchange Digital and Analog information with a wide range of devices in many different environments.
- Input/Output (I/O) capability of computers allows for basic I/O operations.

Accessing I/O Devices

- The components of a computer system communicate with each other through an **Interconnection network**.

- The **Interconnection network** consists of circuits needed to transfer information between the Processor, the Memory unit, and a number of I/O devices.

The Interconnection Network



Memory-Mapped I/O

- Each I/O device must appear to the Processor as consisting of some addressable locations, just like the Memory.
- Some addresses in the address space of the Processor are assigned to these I/O locations, rather than to the main Memory.
- These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of Registers known as I/O Registers.

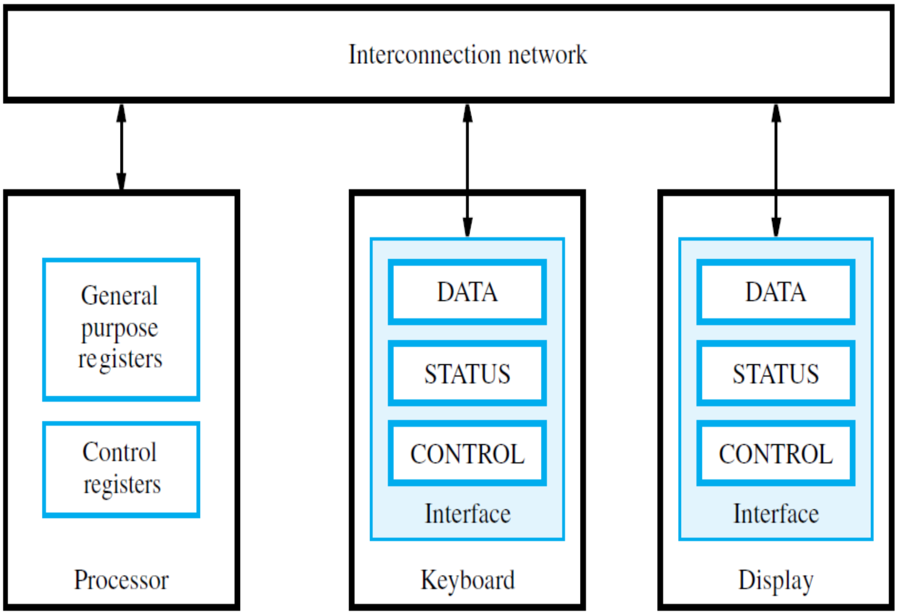
Memory-Mapped I/O

- Since the I/O devices and the Memory share the same address space, this arrangement is called **Memory-Mapped I/O**.
- It is used in most Computers.
- With Memory-Mapped I/O, any machine instruction that can access Memory can be used to transfer data to or from an I/O device.

I/O Device Interface

- One Register may serve as a buffer for data transfers
- Another may hold information about the current status of the device,
- Another may store the information that controls the operational behavior of the device.
- These *data*, *status*, and *control* Registers are accessed by program instructions as if they were Memory locations.
- Typical transfers of information are between I/O Registers and the Registers in the Processor.

The Connections for Processor, Keyboard and Display



Check your understanding!

1. State the various types of Input and Output devices that need to be used in a modern General purpose or Embedded Computer, and provide details how I/O Capability is provided?
2. Explain how accessing I/O devices is made possible in modern Computer, with the aid of a generic sketch.
3. Explain the relevance of Memory Mapped I/O.
4. State the need for I/O Device Interface and illustrate with the aid of a diagram the connections between CPU and I/O devices.

Modes of Data Transfer

1. Programmed I/O

2. Interrupt driven I/O
3. Direct Memory Access (DMA)

Program-Controlled I/O

- Consider a task that reads characters typed on a keyboard, stores these data in the Memory, and displays the same characters on a display screen.
- A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action.
- This method is known as **Program-Controlled I/O**.

Program-Controlled I/O

- It is necessary to ensure that the task happens at the right time.
- An input character must be read in response to a key being pressed.
- For output, a character must be sent to the display only when the display device is able to accept it.
- The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second.

Program-Controlled I/O

- The rate of output transfers from the computer to the display is much higher.
- It is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second.
- However, this is still much slower than the speed of a Processor that can execute billions of instructions per second.
- The difference in speed between the Processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

Program-Controlled I/O

- Program-controlled I/O requires continuous involvement of Processor in I/O activities.
- Almost all of the execution time is spent in wait loops, while the Processor waits for a key to be pressed or for the display to become available.
- Wasting the Processor execution time in this manner leads to poor efficiency.

- It can be avoided by using the concept of Interrupts.

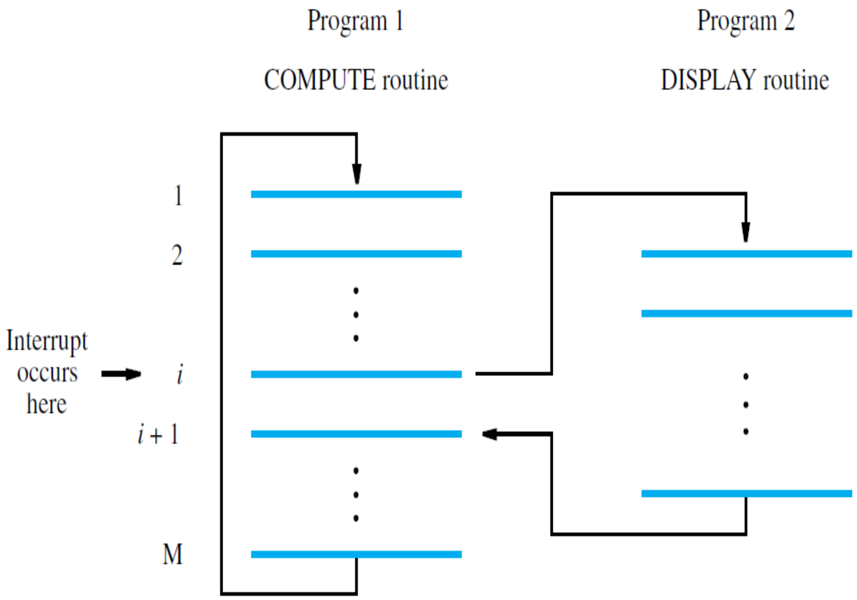
Interrupts

- In P I/O , program enters a wait loop in which it repeatedly tests device status.
- During this period, Processor is not performing any useful computation.
- Other tasks can be performed while waiting for an I/O device to become ready.
- Arrange for I/O device to alert the Processor when it becomes ready.
- It can do so by sending a hardware signal called an **Interrupt Request** to the Processor.
- Since the Processor is no longer required to continuously poll the status of I/O devices, it can use the waiting period to perform other useful tasks.
- Interrupts allow wait periods to be eliminated.

Interrupt driven I/O

- The program routine executed in response to an interrupt request is called the **Interrupt-Service Routine (ISR)**.
- Let an interrupt request arrives during execution of instruction *i*.
- The Processor first completes execution of instruction *i*.
- Then, it loads PC with address of first instruction of ISR.
- After execution of the interrupt-service routine, the Processor returns to instruction *i* + 1.
- When an interrupt occurs, current contents of PC, which point to instruction *i* + 1, must be put in temporary storage.
- A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction *i* + 1.
- The Return address must be saved on the **Stack**.

Transfer of control through use of Interrupts



Interrupt driven I/O

- Processor must inform the device that its request has been recognized so that it may remove its Interrupt-Request signal.
- This is done by a special control signal, called **Interrupt Acknowledge**, which is sent to device thro' the Interconnection network.
- An alternative is to have the transfer of data between the Processor and the I/O device interface accomplish the same purpose.
- The execution of an instruction in the Interrupt-Service Routine (ISR) that accesses the status or data Register in the device interface implicitly informs the device that its Interrupt Request has been recognized.

Direct Memory Access

- Blocks of data are often transferred between the main Memory and I/O devices such as disks without program-controlled intervention by the Processor.
- A special control unit is provided to manage the transfer, without continuous intervention by the Processor.
- This approach is called **Direct Memory Access**, or **DMA**.
- The unit that controls DMA transfers is referred to as a **DMA Controller**.

Direct Memory Access

- It may be part of the I/O device interface, or it may be a separate unit shared by a number of I/O devices.
- The DMA Controller performs the functions that would normally be carried out by the Processor when accessing the main Memory.
- For each word transferred, it provides the Memory address and generates all the control signals needed.
- It increments the Memory address for successive words and keeps track of the number of transfers.

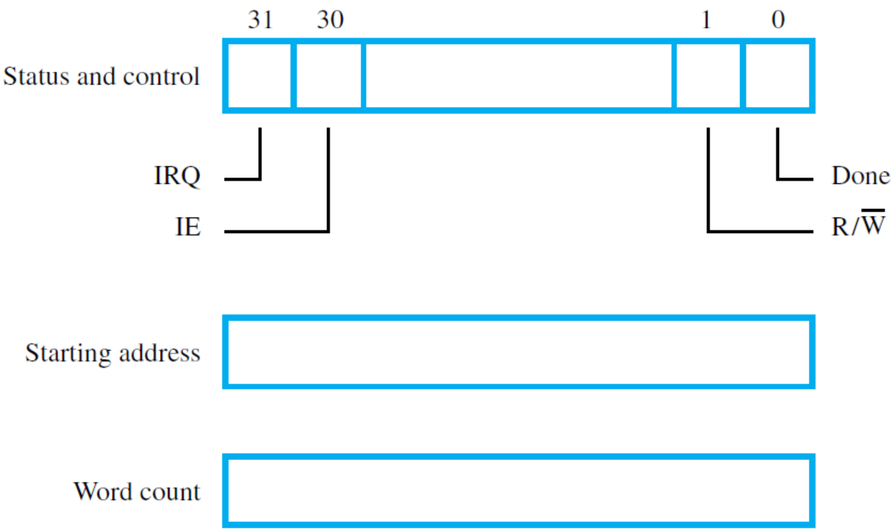
DMA Operation

- Operation of DMA Controller must be under the control of a Program executed by the Processor, usually an OS routine.
- To initiate transfer of a block of words, Processor sends to DMA Controller starting address, number of words in the block, and direction of the transfer.
- The DMA Controller then proceeds to perform the requested operation.
- When the entire block has been transferred, it informs the Processor by raising an Interrupt.

DMA Controller Registers

- Two Registers are used for storing the Starting Address and the Word Count.
- The third Register contains Status and Control Flags.
- The R/W bit determines the direction of the transfer.
- When this bit is set to 1 by a program instruction, Controller performs a Read operation, that is, it transfers data from the Memory to the I/O device.
- Otherwise, it performs a Write operation.
- Additional information is also transferred as may be required by the I/O device.

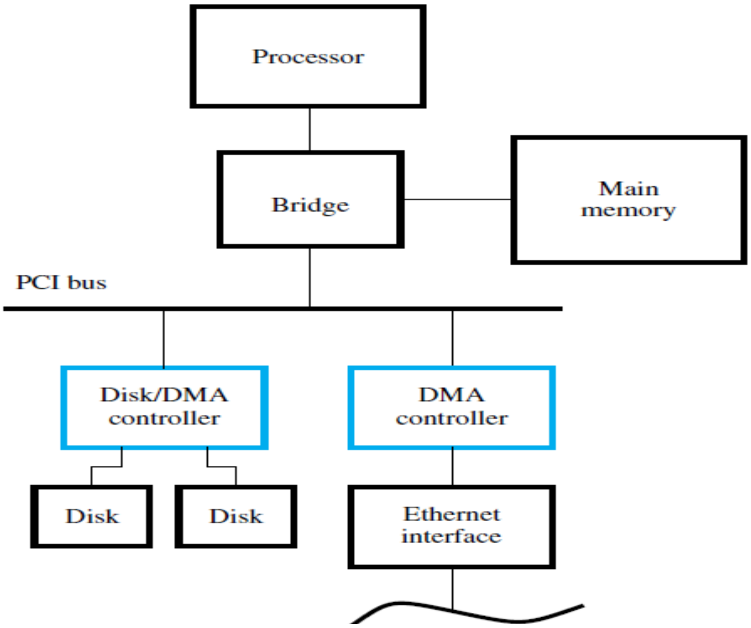
Typical Registers in DMA Controller



DMA

- When the Controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1.
- Bit 30 is the Interrupt-enable flag, IE.
- When this flag is set to 1, it causes the controller to raise an Interrupt after it has completed transferring a block of data.
- Finally, the Controller sets the IRQ bit to 1 when it has requested an interrupt.

Use of DMA controllers in Computer



DMA controllers in Computer

- One DMA controller connects a high-speed Ethernet to the computer's I/O bus (ex. PCI bus).
- The disk controller, which controls two disks, also has DMA capability and provides two DMA channels.
- It can perform two independent DMA operations, as if each disk had its own DMA controller.
- The Registers needed to store the Memory address, the word count, and so on, are duplicated, so that one set can be used with each disk.

DMA Transfer

- To start a DMA transfer of a block of data from the main Memory to one of the disks, an OS routine writes the address and word count information into the Registers of the disk controller.
- The DMA controller proceeds independently to implement the specified operation.
- When the transfer is completed, this fact is recorded in the Status and Control Register of the DMA channel by setting the Done bit.

DMA Transfer

- At the same time, if the IE bit is set, the Controller sends an **Interrupt Request** to the Processor and sets the IRQ bit.
- The Status Register may also be used to record other information, such as whether the transfer took place correctly or errors occurred.

Check your understanding!

1. State the modes of I/O Data Transfer.
2. Explain Programmed controlled I/O mechanism.
3. Explain Interrupt Driven I/O mechanism with the aid of a diagram.
4. What is DMA? How does it use Interrupts?
5. Explain the DMA Controller Registers with the aid of diagrams.
6. Describe the DMA transfer with the aid of diagram and give details of the role of DMA Controller.

Interface Circuits

- The I/O interface of a device consists of the circuitry needed to connect that device to the bus.
- On one side of the interface are the bus lines for address, data, and control.
- On the other side are the connections needed to transfer data between the interface and the I/O device.
- This side is called a **Port**, and it can be either a **Parallel Port** or a **Serial Port**.

Interface Circuits

- A **Parallel Port** transfers multiple bits of data simultaneously to or from the device.
- A **Serial Port** sends and receives data one bit at a time.
- Communication with the Processor is the same for both formats.
- The conversion from a Parallel to a Serial format and vice versa takes place inside the Interface circuit.

Functions of an I/O interface

1. Provides a Register for temporary storage of data.

2. Includes a Status Register containing status info that can be accessed by Processor.

3. Includes a Control Register that holds info governing behavior of Interface.

4. Contains Address-decoding Circuitry to determine when it is being addressed by Processor.

5. Generates the required Timing signals.

6. Performs any Format conversion that may be necessary to transfer data between Processor and I/O device, such as parallel-to-serial conversion, in the case of a serial port.

Interconnection Standards

- A typical Desktop or Notebook Computer has several Ports that can be used to connect I/O devices, such as a Mouse, a Memory key, or a Disk drive.
- Standard interfaces have been developed to enable I/O devices to use interfaces that are independent of any particular Processor.
- For ex, a pen drive can be used with any computer that has a USB port.
- IEEE (Institute of Electrical and Electronics Engineers) develops these standards further and publishes them as IEEE Standards.

Universal Serial Bus (USB)

- Universal Serial Bus (USB) is the most widely used interconnection standard.
- A large variety of devices are available with a USB connector, including Mice, Memory keys, Disk drives, Printers, Cameras, and many more.
- The commercial success of the USB is due to its simplicity and low cost.

Universal Serial Bus (USB)

- The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s).
- Later, USB 2, called High-Speed USB, was introduced.
- It enables data transfers at speeds up to 480 Megabits/s.
- As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Super-speed) was developed. It supports data transfer rates up to 5 Gigabits/s.

Objectives of the USB

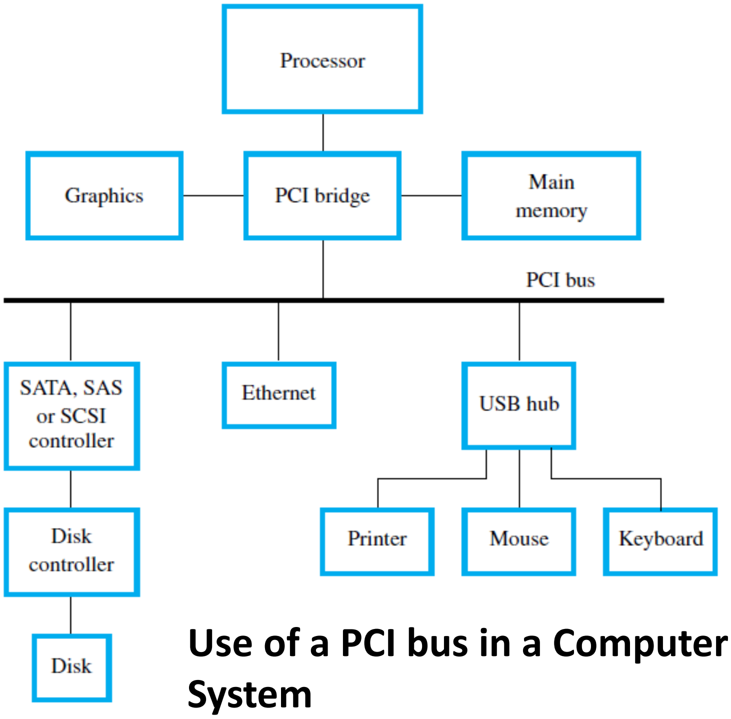
- Provide a simple, low-cost, and easy to use interconnection system.
- Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications.
- Enhance user convenience through a “plug-and-play” mode of operation.

PCI Bus

- The PCI (Peripheral Component Interconnect) bus was developed as a low-cost, Processor-independent bus.
- It is on the motherboard of a computer and used to connect I/O interfaces for a wide variety of devices.
- A device connected to the PCI bus appears to the Processor as if it is connected directly to the Processor bus.
- Its interface Registers are assigned addresses in the address space of the Processor.

Bus Structure

- The PCI bus is connected to the Processor bus via a controller called a Bridge.
- The Bridge has a special port for connecting the computer's Main Memory.
- It may also have another special high speed port for connecting graphics devices.
- The Bridge translates and relays commands and responses from one bus to the other and transfers data between them.



Use of a PCI bus in a Computer System

Advantages and Benefits

- The PCI bus has gained great popularity, particularly in the PC world.
- It is also used in many other computers, to benefit from the wide range of I/O devices for which a PCI interface is available.
- Both a 32-bit and a 64-bit configuration are available, using either a 33-MHz or 66-MHz clock.
- A high-performance variant known as PCI-X is also available.
- It is a 64-bit bus that runs at 133 MHz.
- Yet higher performance versions of PCI-X run at speeds up to 533 MHz.

SCSI Bus

- The acronym SCSI stands for Small Computer System Interface .
- It refers to a standard bus defined by the American National Standards Institute (ANSI).
- The SCSI bus may be used to connect a variety of devices to a computer.
- It is particularly well-suited for use with disk drives.
- It is often found in installations such as institutional databases or email systems where many disks drives are used.
- In the original specifications of the SCSI standard, devices are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates of up to 5 Megabytes/s.

SCSI Bus

- The standard has undergone many revisions, and its data transfer capability has increased rapidly.
- SCSI-2 and SCSI-3 have been defined, and each has several options.
- Data are transferred either 8 bits or 16 bits in parallel, using clock speeds of up to 80 MHz.
- There are also several options for the electrical signaling scheme used.
- The bus may use single-ended transmission, where each signal uses one wire, with a common ground return for all signals.
- In another option, differential signaling is used, with a pair of wires for each signal.

Data Transfer over SCSI bus

- Devices connected to SCSI bus are not part of address space of Processor in the same way as devices connected to the Processor bus or to the PCI bus.
- A SCSI bus may be connected directly to Processor bus, or more likely to another standard I/O bus such as PCI, through a **SCSI controller**.
- Data and commands are transferred in the form of multi-byte messages called packets.
- To send commands or data to a device, Processor assembles the info in Memory then instructs **SCSI controlle**r to transfer it to the device.
- Similarly, when data are read from a device, **SCSI controller** transfers the data to Memory and then informs Processor by raising an Interrupt.

Example: Read operation.

- Assume that the Processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous.
- The Processor sends a command to the SCSI controller, which causes the following sequence of events to take place: →

1. The SCSI controller contends for control of the SCSI bus.

2. When it wins the arbitration process, the SCSI controller sends a command to the disk controller, specifying the required Read operation.

3. The disk controller cannot start to transfer data immediately. It must first move the read head of the disk to the required sector. Hence, it sends a message to the SCSI controller indicating that it will temporarily suspend the connection between them. The SCSI bus is now free to be used by other devices.

Example: Read operation (Cont…)

4. The disk controller sends a command to the disk drive to move the read head to the first sector involved in the requested Read operation. It reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data, it requests control of the bus. After it wins arbitration, it re-establishes the connection with the SCSI controller, sends the contents of the data buffer, then suspends the connection again.

5. The process is repeated to read and transfer the contents of the second disk sector.

6. The SCSI controller transfers the requested data to the main Memory and sends an interrupt to the Processor indicating that the data are now available.

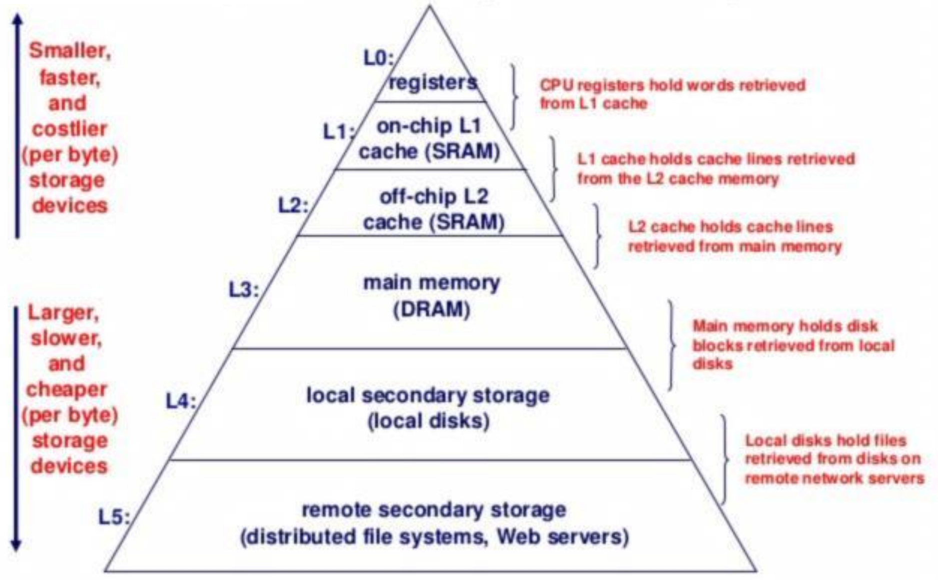
Check your understanding!

1. What is a Port and what are the types? Explain.
2. Explain functions of I/O Interface.
3. What is the need for a standard for I/O device connections?
4. Describe the objectives of USB and provide technical details of the standard.
5. Describe PCI Bus with the aid of diagram and give its advantages and benefits.
6. Investigate the SCSI Bus and how its serves the purpose of efficient data transfer, with details of a typical read operation.

Memory System

Module V Part-B

Pyramid Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
registers
L1:
on-chip L1 cache (SRAM)
L2:
off-chip L2 cache (SRAM)
L3:
main memory (DRAM)
L4:
local secondary storage (local disks)
L5:
remote secondary storage (distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from the L2 cache memory

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

Memory Hierarchy

- Computer Memory comprise of Primary Memory and Secondary Memory with different types.
- All the different types of Memory units are employed effectively in a Computer System.
- The entire Computer Memory can be viewed as the **Memory Hierarchy**.
- The fastest access is to data held in **Processor Registers** (Level L0).
- So the **Processor Registers** are at the top of Hierarchy in terms of speed of access.
- But, the Registers provide only a very small portion of the required Memory.

Memory Hierarchy

- At the next level of Hierarchy is a relatively small amount of Memory directly on Processor chip.
- This Memory, called a **Processor Cache**, holds copies of instructions and data stored in a much larger Memory that is provided externally.
- There are often two or more levels of **Cache**.
- A Primary Cache is always located on Processor chip.
- This Cache is small and its access time is comparable to that of Processor Registers.
- The primary Cache is referred to as the **Level 1 (L1) Cache**.
- A larger, and hence somewhat slower, Secondary Cache is placed between the Primary Cache and the rest of the Memory.
- It is referred to as the **Level 2 (L2) Cache**.
- Often, the L2 Cache is not on the Processor chip.

Memory Hierarchy

- The next level in the Hierarchy is the **Main Memory**.
- This is a large Memory implemented using Dynamic Memory components, typically assembled in Memory modules such as **SIMM**s and **DIMM**s (**D**ual **I**n-line **Memory M**odule).
- The Main Memory is much larger but significantly slower than Cache memories.
- In a Computer with a Processor clock of 2 GHz or higher, access time for Main Memory can be as much as 100 times longer than access time for the L1 Cache.

Memory Hierarchy

- Disk devices and Tape devices and optical storage provide a very large amount of inexpensive Memory, and they are widely used as local Secondary Storage in computer systems. (Level-L5)
- They are very slow compared to the Main Memory.
- Remote Secondary Storage include Distributed File Systems and Web Servers. (Level-L5)
- They represent the bottom level in the Memory Hierarchy.
- Cost per bit of storage is the least at the bottom and very high at the top!
- Mostly Electromechanical devices and effectively provide an “Ocean” of storage that project the small “Pool” of top most layer!!

Characteristics of Memory

- An ideal Memory would be fast, large, and inexpensive.
- A very fast Memory can be implemented using Static RAM chips.
- But, these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than Dynamic RAM cells.
- Although Dynamic Memory units with Gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data.
- A solution is provided by using Secondary Storage, mainly Magnetic disks, to provide the required Memory space.

Characteristics of Memory (Cont…)

- Disks are available at a reasonable cost, and they are used extensively in computer systems.
- However, they are much slower than semiconductor Memory units.
- A very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main Memory can be built with dynamic RAM technology.
- This leaves the more expensive and much faster Static RAM technology to be used in smaller units where speed is of the essence, such as in Cache Memories.
- During program execution, the speed of Memory access is of utmost importance.
- The key to managing the operation of the Hierarchical Memory system is to bring the instructions and data that are about to be used as close to the processor as possible.
- This is the main purpose of using Cache Memories.

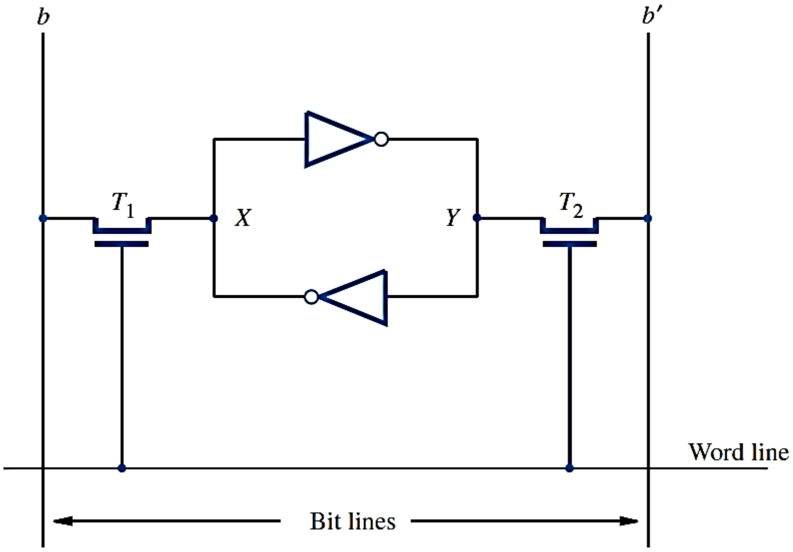
Semiconductor RAM Memories

- Semiconductor Random-Access Memories (RAMs) are available in a wide range of speeds.
- Their cycle times range from 100 ns to less than 10 ns.

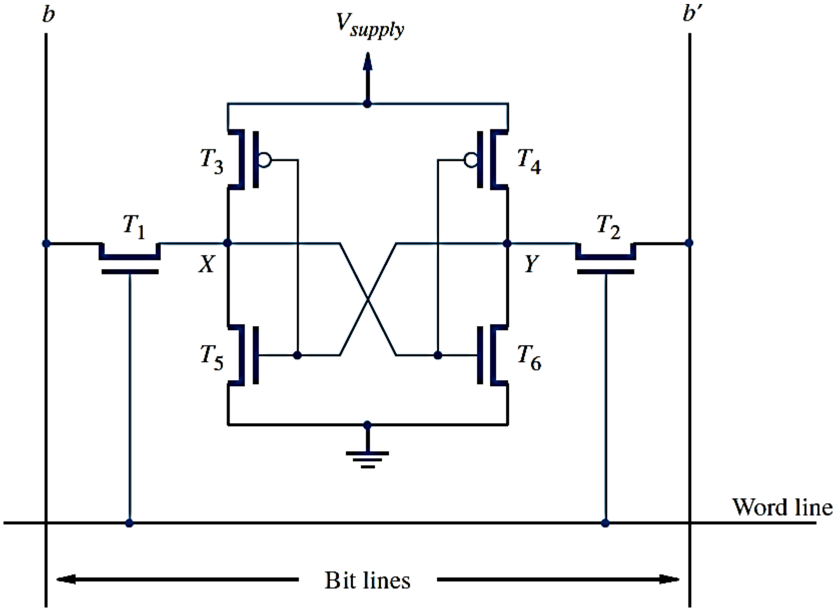
Static Memories

- Memories that consist of circuits capable of retaining their state as long as power is applied are known as *Static Memories*.
- *Static RAM* (SRAM) cell may be implemented as two inverters cross-connected to form a latch.
- The latch is connected to two bit lines by transistors *T*1 and *T*2.
- These transistors act as switches that can be opened or closed under control of the word line.
- When the word line is at ground level, the transistors are turned off and the latch retains its state.
- For example, if the logic value at point *X* is 1 and at point *Y* is 0, this state is maintained as long as the signal on the word line is at ground level.
- Assume that this state represents the value 1.

Static RAM Cell



Static RAM Cell using CMOS Transistors



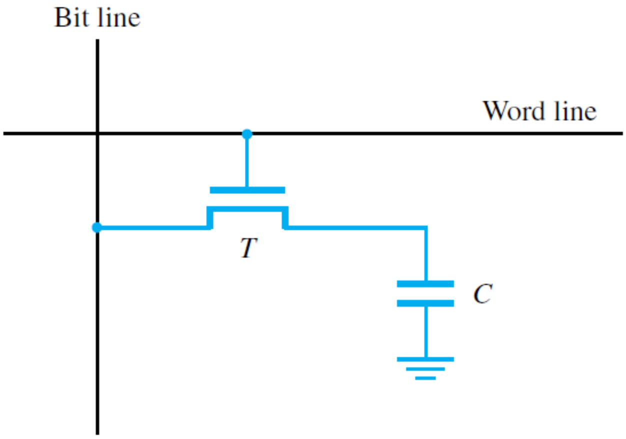
Static RAM Cell using CMOS Transistors

- Continuous power is needed for cell to retain its state.
- If power is cut off, cell's contents are lost.
- On power ON, latch settles into a new stable state.
- Hence, SRAMs are said to be *volatile* memories because their contents are lost when power is cut.
- A major advantage of CMOS SRAMs is their very low power consumption, because current flows in the cell only when the cell is being accessed.
- Otherwise, *T*1, *T*2, and one transistor in each inverter are turned off, ensuring that there is no continuous electrical path between V_{supply} and ground.
- Static RAMs can be accessed very quickly.
- Access times on the order of a few nanoseconds are found in commercially available chips.
- SRAMs are used in applications where speed is of critical concern.

Dynamic RAM

- Static RAMs are fast, but their cells require several transistors.
- Less expensive and higher density RAMs can be implemented with simpler cells.
- But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations.
- Memories that use such cells are called *dynamic RAMs* (DRAMs).
- Information is stored in a dynamic Memory cell in the form of a charge on a capacitor, but this charge can be maintained for only tens of milliseconds.
- Since the cell is required to store information for a much longer time, its contents must be periodically *refreshed* by restoring the capacitor charge to its full value.
- This occurs when the contents of the cell are read or when new information is written into it.

Dynamic RAM Cell



Dynamic RAM

- An example of a dynamic Memory cell consists of a capacitor, *C*, and a transistor, *T*.
- To store information in this cell, transistor *T* is turned on and an appropriate voltage is applied to the bit line.
- This causes a known amount of charge to be stored in the capacitor.
- After the transistor is turned off, the charge remains stored in the capacitor, but not for long.
- The capacitor begins to discharge.
- This is because the transistor continues to conduct a tiny amount of current, measured in pA, after it is turned off.

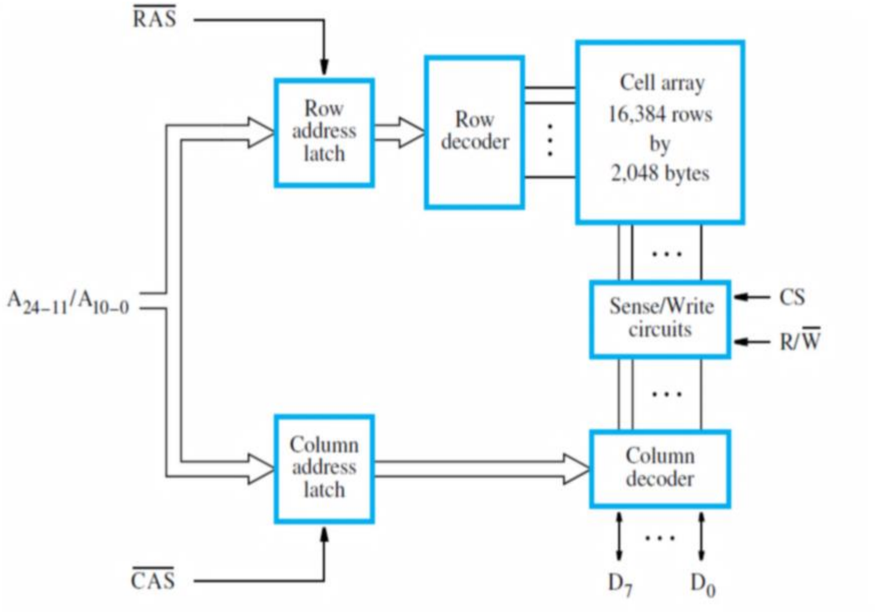
Dynamic RAM

- Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge in the capacitor drops below some threshold value.
- During a Read operation, the transistor in a selected cell is turned on.
- A sense amplifier connected to the bit line detects whether the charge stored in the capacitor is above or below the threshold value.
- If the charge is above the threshold, the sense amplifier drives the bit line to the full voltage representing the logic value 1.

Dynamic RAM

- As a result, the capacitor is recharged to the full charge corresponding to the logic value 1.
- If the sense amplifier detects that the charge in the capacitor is below the threshold value, it pulls the bit line to ground level to discharge the capacitor fully.
- Thus, reading the contents of a cell automatically refreshes its contents.
- Since the word line is common to all cells in a row, all cells in a selected row are read and refreshed at the same time.

Internal organization of a 32M × 8 dynamic Memory chip.



32M × 8 Dynamic Memory chip

- A 256-Megabit DRAM chip, configured as 32M × 8, is shown.
- The cells are organized in the form of a 16K × 16K array.
- The 16,384 cells in each row are divided into 2,048 groups of 8, forming 2,048 bytes of data.
- Therefore, 14 address bits are needed to select a row, and another 11 bits are needed to specify a group of 8 bits in the selected row.
- In total, a 25-bit address is needed to access a byte in this Memory.
- The high-order 14 bits and the low-order 11 bits of the address constitute the row and column addresses of a byte, respectively.
- To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 14 pins.
- During a Read or a Write operation, the row address is applied first.
- It is loaded into the row address latch in response to a signal pulse on an input control line called the Row Address Strobe (RAS).
- This causes a Read operation to be initiated, in which all cells in the selected row are read and refreshed.

32M × 8 Dynamic Memory chip

- Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of a second control line called the Column Address Strobe (CAS).
- The information in this latch is decoded and the appropriategroup of 8 Sense/Write circuits is selected.
- If the R/W control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7−0.
- For a Write operation, the information on theD7−0 lines is transferred to the selected circuits, then used to overwrite the contents of the selected cells in the corresponding 8 columns.
- Note that in commercial DRAM chips, the RAS and CAS control signals are active when low.
- Hence, addresses are latched when these signals change from high to low.
- The signals are shown in diagrams as RAS and CAS to indicate this fact.

Asynchronous DRAMs

- The timing of the operation of the DRAM described above is controlled by the RAS and CAS signals.
- These signals are generated by a Memory controller circuit external to the chip when the processor issues a Read or aWrite command.
- During a Read operation, the output data are transferred to the processor after a delay equivalent to the Memory’s access time.
- Such memories are referred to as **Asynchronous DRAMs.**

Synchronous DRAMs

- In the early 1990s, developments in Memory technology resulted in DRAMs whose operation is synchronized with a clock signal.
- Such memories are known as *synchronous DRAMs* (SDRAMs).
- The cell array is the same as in asynchronous DRAMs.
- The distinguishing feature of an SDRAM is the use of a clock signal, the availability of which makes it possible to incorporate control circuitry on the chip that provides many useful features.
- For example, SDRAMs have built-in refresh circuitry, with a refresh counter to provide the addresses of the rows to be selected for refreshing.
- As a result, the dynamic nature of these Memory chips is almost invisible to the user.

Synchronous DRAMs

- Synchronous DRAM scan deliver data at a very high rate, because all the control signals needed are generated inside the chip.
- The initial commercial SDRAMs in the 1990s were designed for clock speeds of up to 133 MHz.
- As technology evolved, much faster SDRAM chips were developed.
- Today's SDRAMs operate with clock speeds that can exceed 1 GHz.
- Chips are manufactured in different organizations, to provide flexibility in designing Memory systems.
- For example, a 1-Gbit chip may be organized as 256M × 4, or 128M × 8.

SIMMs and DIMMs

- Packaging considerations have led to the development of assemblies known as Memory modules.
- Each such module houses many Memory chips, typically in the range 16 to 32, on a small board that plugs into a socket on the computer’s motherboard.
- Memory modules are commonly called **SIMMs** (Single In-line Memory Modules) or **DIMMs** (Dual In-line Memory Modules), depending on the configuration of the pins.
- Modules of different sizes are designed to use the same socket.
- For example, 128M × 64, 256M × 64, and 512M × 64 bit DIMMs all use the same 240-pin socket.
- Thus, total Memory capacity is easily expanded by replacing a smaller module with a larger one, using the same socket.

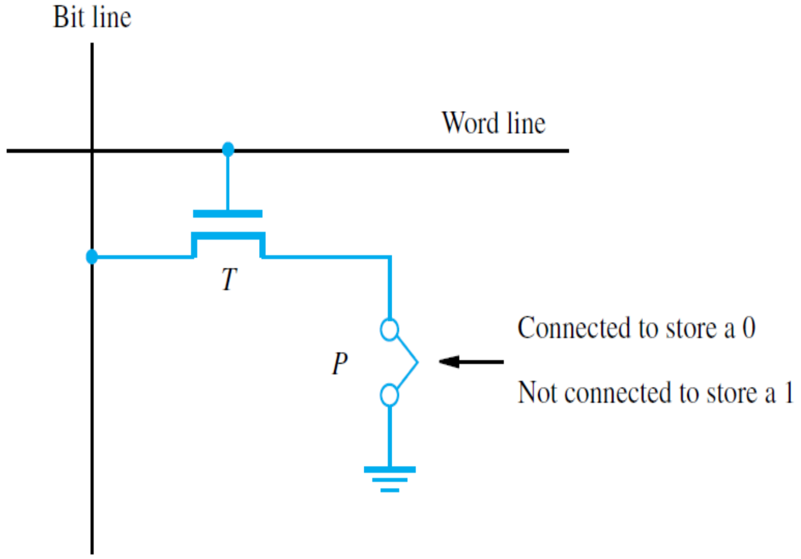
Check your understanding!

1. Investigate Memory Hierarchy with the aid of a diagram and provide details of the different layers.
2. Explain the characteristics of Memory.
3. Describe Static Memories with the aid of diagrams.
4. Describe Dynamic RAM with the aid of a diagram.
5. Provide the internal organization of a Dynamic Memory Chip with a diagram.
6. Compare and Contrast Asynchronous and Synchronous DRAMs.
7. Explain the relevance of SIMMs and DIMMs in modern Computer Systems.

Read-only Memories

- Both Static and dynamic RAM chips are Volatile Memory, which means that they retain information only while power is turned on.
- There are many applications requiring Memory devices that retain the stored information when power is turned off.
- So there is need to store a small program in such a Memory, to be used to start the bootstrap process of loading the operating system from a hard disk into the main Memory.
- Many embedded applications do not use a hard disk and require Nonvolatile Memories to store their software.
- Generally, their contents can be read in the same way as for volatile Memories.
- But, a special writing process is needed to place the information into a Nonvolatile Memory.
- Since its normal operation involves only reading the stored data, a Memory of this type is called a **Read-only Memory (ROM).**

A ROM Cell



Read- Only Memory (ROM)

- Information can be written into it only once at the time of manufacture.
- A logic value 0 is stored in the cell if the transistor is connected to ground at point *P*; otherwise, a 1 is stored.
- The bit line is connected thro’ a resistor to power supply.
- To read state of cell, word line is activated to close the transistor switch.
- As a result, the voltage on the bit line drops to near zero, if there is a connection between the transistor and ground.
- If there is no connection to ground, the bit line remains at the high voltage level, indicating a 1.
- A sense circuit at the end of the bit line generates the proper output value.
- The state of the connection to ground in each cell is determined when the chip is manufactured, using a mask with a pattern that represents the information to be stored.

PROM

- Some ROM designs allow the data to be loaded by the user, thus providing a *programmable ROM* (PROM).
- Programmability is achieved by inserting a fuse at point *P*.
- Before it is programmed, the memory contains all 0s.
- The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses.
- This process is irreversible.
- PROMs provide flexibility and convenience not available with ROMs.
- The cost of preparing the masks needed for storing a particular information pattern makes ROMs cost effective only in large volumes.
- Memory chips can be programmed directly by the user.

EPROM

- It allows the stored data to be erased and new data to be written into it.
- Such an **Erasable, Reprogrammable ROM** is usually called an **EPROM**.
- It provides considerable flexibility during development phase of digital systems.
- Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs or PROMs while software is being developed.
- An EPROM cell has a structure similar to the ROM cell.

EPROM

- However, the connection to ground at point *P* is made through a special transistor.
- The transistor is normally turned off, creating an open switch.
- It can be turned on by injecting charge into it that becomes trapped inside.
- Thus, an EPROM cell can be used to construct a memory in the same way as the ROM cell.
- Erasure requires dissipating the charge trapped in the transistors that form the memory cells.
- This can be done by exposing the chip to ultraviolet light, which erases the entire contents of the chip.
- To make this possible, EPROM chips are mounted in packages that have transparent windows.

EEPROM

- An EPROM must be physically removed from the circuit for reprogramming.
- Also, the stored information cannot be erased selectively.
- The entire contents of the chip are erased when exposed to ultraviolet light.
- Another type of erasable PROM can be programmed, erased, and reprogrammed electrically.
- Such a chip is called an **Electrically Erasable PROM**, or **EEPROM**.
- It does not have to be removed for erasure.
- Moreover, it is possible to erase the cell contents selectively.
- One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity.
- They have replaced EPROMs in practice due to their advantages.

Flash

- An approach similar to EEPROM technology has given rise to *flash memory* devices.
- A flash cell is based on a single transistor controlled by trapped charge, much like an EEPROM cell.
- Also like an EEPROM, it is possible to read the contents of a single cell.
- The key difference is that, in a flash device, it is only possible to write an entire block of cells.
- Prior to writing, the previous contents of the block are erased.
- Flash devices have greater density, which leads to higher capacity and a lower cost per bit.
- They require a single power supply voltage, and consume less power in their operation.

Flash

- The low power consumption of flash memories makes them attractive for use in portable, battery-powered equipment.
- Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players.
- In hand-held computers and cell phones, a flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive.
- A flash memory is used in digital cameras to store picture data.
- In MP3 players, flash memories store the data that represent sound.

Flash Cards

- One way of constructing a larger module is to mount flash chips on a small card.
- Such flash cards have a standard interface that makes them usable in a variety of products.
- A card is simply plugged into a conveniently accessible slot.
- Flash cards with a USB interface are widely used and are commonly known as memory keys.
- They come in a variety of memory sizes.
- Larger cards may hold as much as 32 GBytes.
- A minute of music can be stored in about 1 MByte of memory, using the MP3 encoding format.
- Hence, a 32-GByte flash card can store approximately 500 hours of music.

Flash Drives

- Larger flash memory modules have been developed to replace hard disk drives, and hence are called flash drives.
- They are designed to fully emulate hard disks, to the point that they can be fitted into standard disk drive bays.
- However, the storage capacity of flash drives is significantly lower.
- Currently, the capacity of flash drives is on the order of 64 to 128 GBytes.

Flash Drives

- In contrast, hard disks have capacities exceeding a Terabyte.
- Also, disk drives have a very low cost per bit.
- The fact that flash drives are solid state electronic devices with no moving parts provides important advantages over disk drives.
- They have shorter access times, which result in a faster response.
- They are insensitive to vibration and they have lower power consumption, which makes them attractive for portable, battery-driven applications.

Check your understanding!

1. Compare and Contrast RAM and ROM by providing their inherent characteristics and differences.
2. With the aid of a diagram, explain the ROM Cell.
3. Explain PROM, EPROM and EEPROM.
4. Describe Flash memory and Flash Cards.
5. Compare and Contrast Flash Drives and Hard Disk Drives.

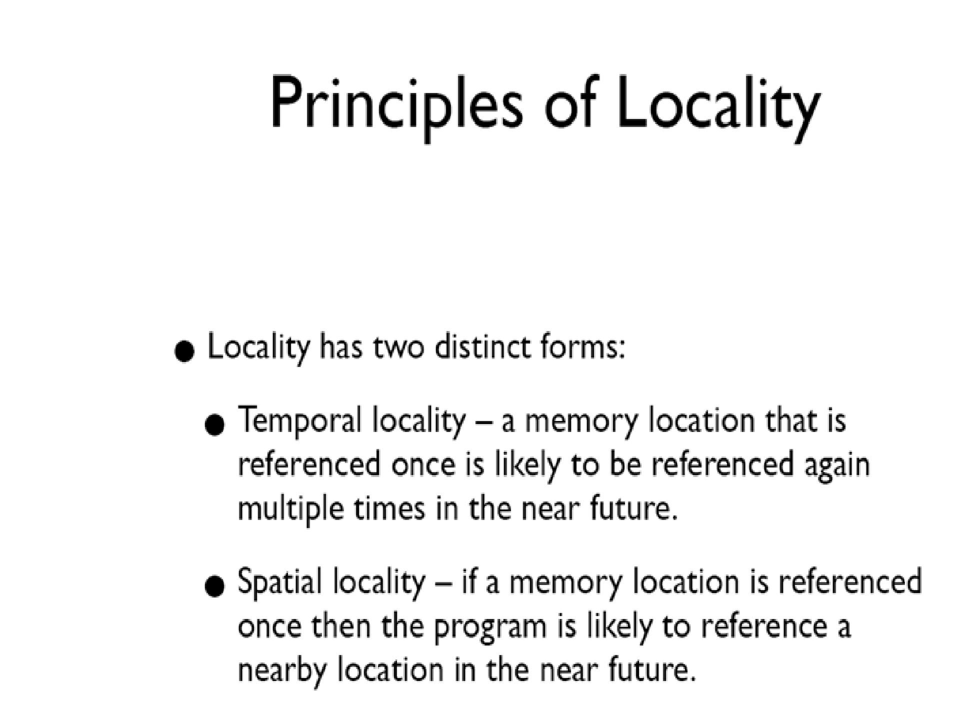
Cache Memory
AND
Virtual Memory

MODULE VI

EC206 CO

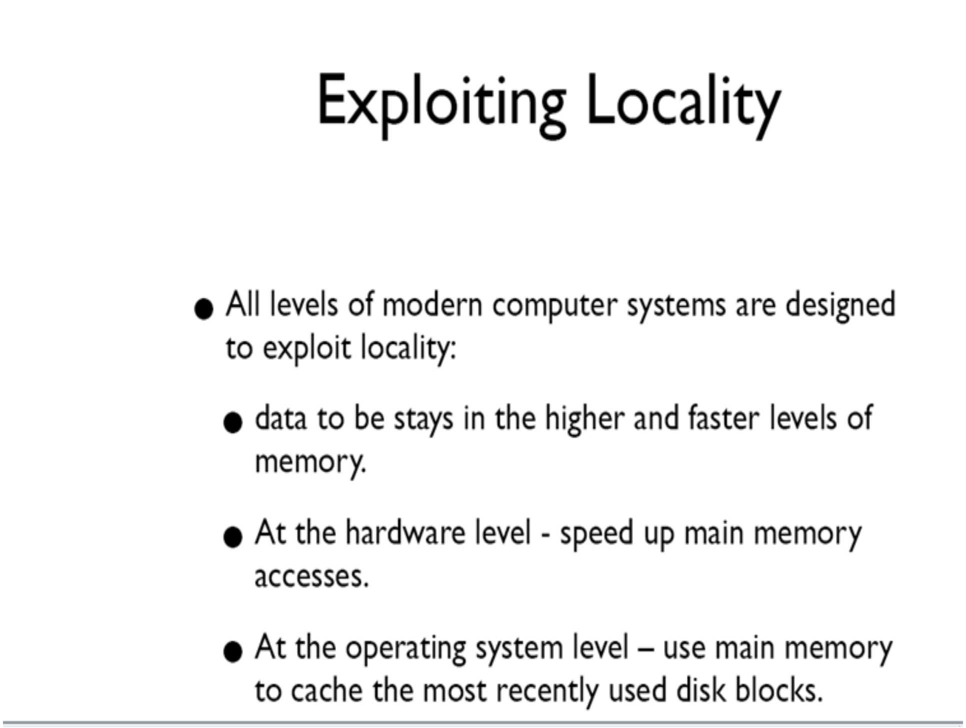
Principles of Locality

- Memory hierarchies work because well written programs tend to exhibit good locality.
  - They tend to reference data items that are near other recently referenced data items.
  - Or that they were recently referenced themselves.
  - Known as the Principal of Locality.



Principles of Locality

- Locality has two distinct forms:
  - Temporal locality – a memory location that is referenced once is likely to be referenced again multiple times in the near future.
  - Spatial locality – if a memory location is referenced once then the program is likely to reference a nearby location in the near future.

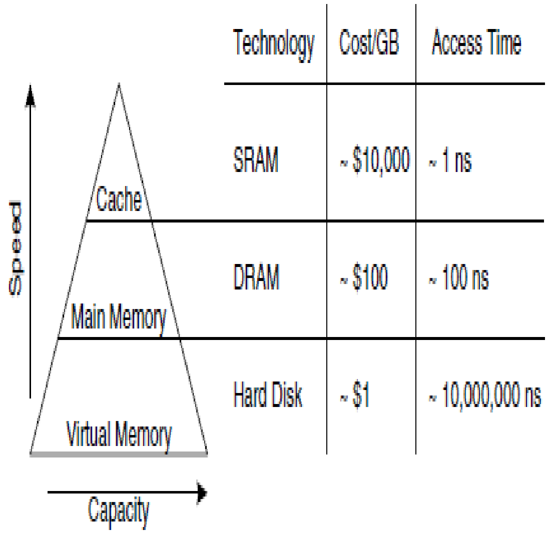


Exploiting Locality

- All levels of modern computer systems are designed to exploit locality:
  - data to be stays in the higher and faster levels of memory.
  - At the hardware level - speed up main memory accesses.
  - At the operating system level – use main memory to cache the most recently used disk blocks.

Exploiting Locality

- By implication:
  - Memory devices at the next level of the hierarchy can be slower and thus larger and cheaper per bit.
- The overall effect is:
  - A large pool of memory that costs as much as the cheap storage (cents/bit) near the bottom of hierarchy.
  - A memory system that serves data to the CPU at the rate of the fast memory devices near the top of the hierarchy.



Technology	Cost/GB	Access Time
SRAM	~ $10,000	~ 1 ns
DRAM	~ $100	~ 100 ns
Hard Disk	~ $1	~ 10,000,000 ns

Memory hierarchy components, with typical characteristics in 2006

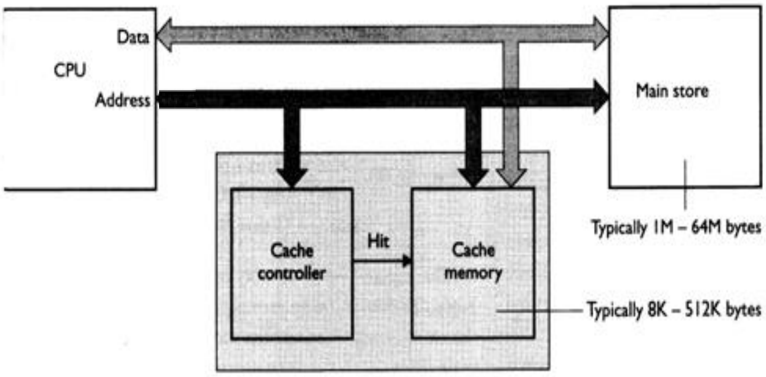
Locality of Reference

- Property of well written Computer Programs.
- Analysis of Programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly.
- These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.
- Many instructions in localized areas of Program are executed repeatedly during some time period.
- This behavior manifests itself in two ways:
- Temporal and Spatial.
- Temporal means that a recently executed instruction is likely to be executed again very soon.
- The Spatial aspect means that instructions close to a recently executed instruction are also likely to be executed soon.

Cache Memories

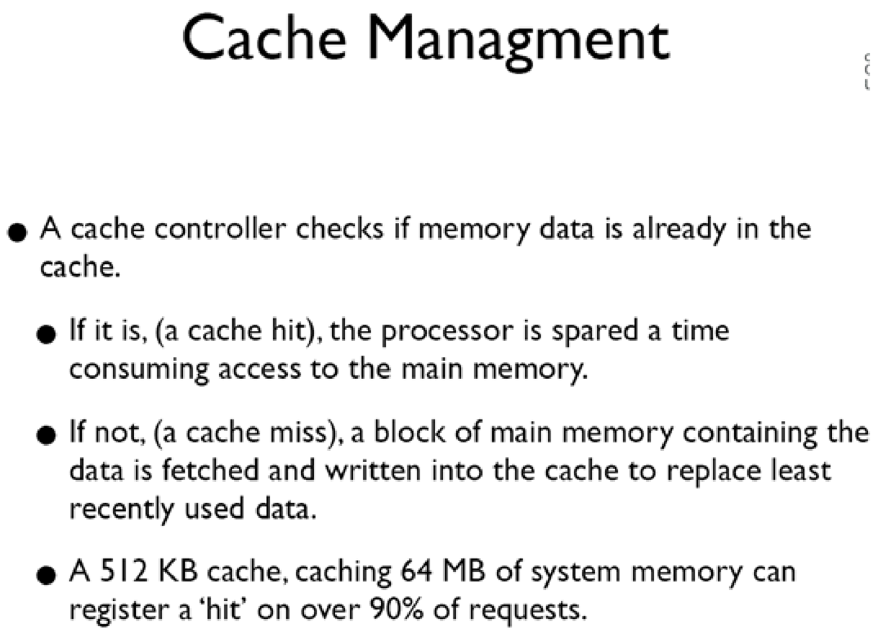
- Cache is a small and very fast Memory, placed between the Processor and the Main Memory.
- **Temporal locality** suggests that whenever a**n** instruction or data, is first needed, this item should be brought into Cache, as it is likely to be needed again soon.
- **Spatial locality** suggests that instead of fetching just one item from Main Memory to Cache, it is useful to fetch several items that are located at adjacent addresses as well.
- The term Cache Block refers to a set of contiguous address locations of some size.
- Another term that is often used to refer to a Cache block is a Cache Line.

Cache Organisation



Concept of Cache Memory

- When Processor issues a Read request, contents of a block of Memory words with location specified are transferred into Cache.
- Later, when Program references any of the locations in this block, the desired contents are read directly from the Cache.
- Usually, Cache Memory can store a reasonable no: of blocks at any given time, but this number is small compared to total no: of blocks in Main Memory.



Cache Managment

- A cache controller checks if memory data is already in the cache.
  - If it is, (a cache hit), the processor is spared a time consuming access to the main memory.
  - If not, (a cache miss), a block of main memory containing the data is fetched and written into the cache to replace least recently used data.
  - A 512 KB cache, caching 64 MB of system memory can register a 'hit' on over 90% of requests.

Cache Hits

- Processor issues Read and Write requests using addresses that refer to locations in Memory.
- The Cache controller determines whether requested word currently exists in Cache.
- If it does, the Read or Write operation is performed on the appropriate Cache location.
- A read or write Cache Hit is said to have occurred.
- The Main Memory is not involved when there is a Cache hit in a Read operation.

Write-through

- For a Write operation, the system can proceed in one of two ways.
- In the first technique, called the Write-through protocol, both the Cache location and the Main Memory location are updated.

Write-back, or Copy-back

- The second technique is to update only Cache location and to mark the block containing it with an associated flag bit, called the dirty or modified bit.
- Main Memory location of word is updated later, when block containing this marked word is removed from Cache to make room for a new block.
- This technique is known as the write-back, or copy-back.

Comparison

- The write-through protocol is simpler than write-back protocol, but it results in unnecessary Write operations in Main Memory when a given Cache word is updated several times during its Cache residency.
- The write-back protocol also involves unnecessary Write operations, as all words of block are eventually written back, even if only a single word has been changed while block was in Cache.
- The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to Memory chips.

Cache Miss

- A Read operation for a word that is not in the Cache constitutes a Read Cache Miss.
- It causes the block of words containing the requested word to be copied from Main Memory into Cache.
- After the entire block is loaded into Cache, the particular word requested is forwarded to Processor.
- When a Write Cache Miss occurs using the write-through protocol, information is written directly into Main Memory.
- For write-back protocol, block containing the addressed word is first brought into Cache, and then desired word in Cache is overwritten with the new information.

Memory SYSTEM PERFORMANCE ANALYSIS

- Memory system performance metrics are miss rate or hit rate and average Memory access time.
- Miss and hit rates are calculated as:

$$\textit{Miss Rate} = \frac{\textit{Number of misses}}{\textit{Number of total memory accesses}} = 1 - \textit{Hit Rate}$$

$$\textit{Hit Rate} = \frac{\textit{Number of hits}}{\textit{Number of total memory accesses}} = 1 - \textit{Miss Rate}$$

Ex: 1 **CALCULATE CACHE PERFORMANCE**

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache.

The other 750 data values are supplied to the Processor by Main Memory or disk Memory.

What are the miss and hit rates for the cache?

Sol:

The miss rate is 750/2000 = 0.375 = **37.5%.**

The hit rate is 1250/2000= 0.625 1- 0.375= **62.5%.**

Average Memory Access Time (AMAT)

- It is average time a Processor must wait for Memory per load or store instruction.
- The Processor first looks for data in the cache.
- If the cache misses, Processor then looks in Main Memory.
- If Main Memory misses, Processor accesses VirtualMemory on hard disk.
- Thus, *AMAT* is calculated as:

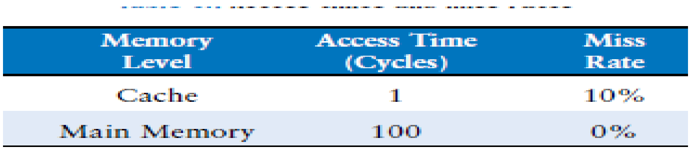
$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM})$$

- where t_{cache}, t_{MM}, and t_{VM} are the access times of cache, Main Memory, and VirtualMemory, and MR_{cache} and MR_{MM} are cache and Main Memory miss rates, respectively.

Ex: 2 **CALCULATING AMAT**

Suppose a computer system has a Memory organization with only two levels of hierarchy, a cache and Main Memory.

What is the average Memory access time given the access times and miss rates?



Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Sol:

The average Memory access time is 1 + 0.1(100)= **11 cycles**.

Ex 3 **IMPROVING ACCESS TIME**

An 11-cycle average Memory access time means that the Processor spends ten cycles waiting for data for every one cycle actually using that data.

What cache miss rate is needed to reduce the average Memory access time to 1.5 cycles? (Use previous ex data)

Sol:

- If the miss rate is *m*, the average access time is 1+ 100*m*.
- 1+ 100m =1.5 → m= 0.5/100 = 0.005.
- Cache miss rate = **0.5%**

Mapping and Replacement

- The correspondence between Main Memory blocks and those in Cache is specified by Mapping Method.
- When Cache is full and a Memory word that is not in Cache is referenced, Cache controller must decide which block should be removed to create space for new block that contains referenced word.
- The collection of rules for making this decision constitutes Cache's Replacement algorithms.

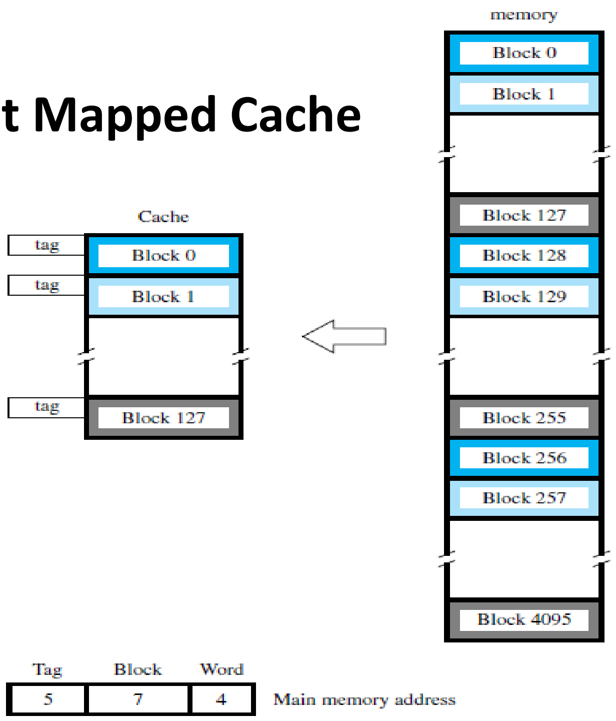
Mapping Methods

- There are three methods to determine where Memory blocks are placed in Cache.
- Consider a Cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that Main Memory is addressable by a 16-bit address.
- The Main Memory has 64K words, as 4K blocks of 16 words each.

Direct Mapping

- The simplest way to determine Cache locations in which to store Memory blocks is the Direct-mapping technique.
- In this technique, block *j* of Main Memory maps onto block *j* modulo 128 of Cache.
- Thus, whenever one of Main Memory blocks 0*,* 128*,* 256*, . . .* is loaded into Cache, it is stored in Cache block 0.
- Blocks 1*,* 129*,* 257*, . . .* are stored in Cache block 1, and so on.

Direct Mapped Cache

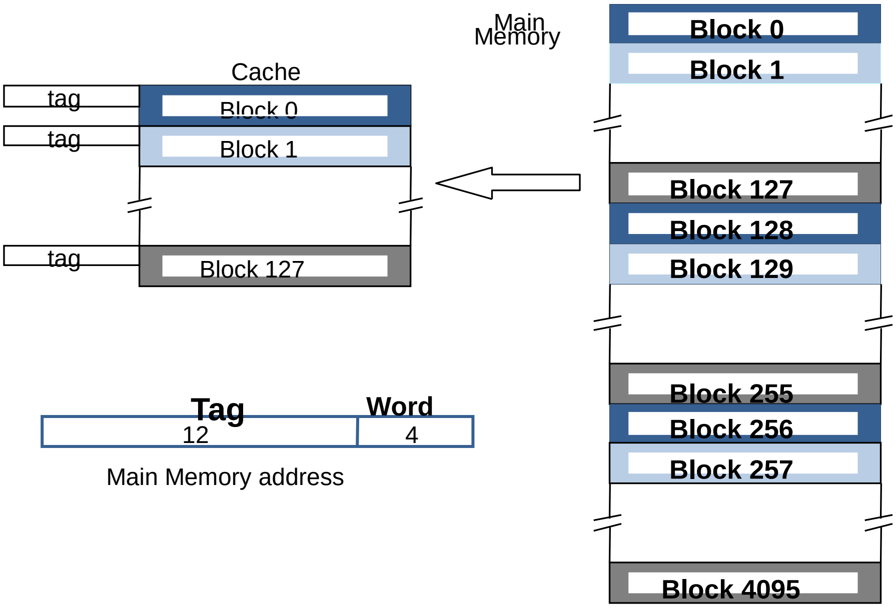


- Since more than one Memory block is mapped onto a given Cache block position, Contention may arise for that position even when the Cache is not full.
- For ex, instructions of a program may start in block 1 and continue in block 129, after a branch.
- As this program is executed, both of these blocks must be transferred to the block-1 position in Cache.
- Contention is resolved by allowing new block to overwrite currently resident block.

- With direct mapping, replacement algorithm is simple.
- Placement of a block in Cache is determined by its Memory address.
- The Memory address can be divided into three fields.
- The low-order 4 bits select one of 16 words in a block.
- When a new block enters the Cache, 7-bit Cache block field determines Cache position in which this block must be stored.
- The high-order 5 bits of the Memory address of the block are stored in 5 tag bits associated with its location in Cache.
- The tag bits identify which of 32 Main Memory blocks mapped into this Cache position is currently resident in Cache.

- As execution proceeds, 7-bit Cache block field of each address generated by Processor points to a particular block location in Cache.
- High-order 5 bits of address are compared with tag bits associated with that Cache location.
- If they match, then desired word is in that block of Cache.
- If there is no match, then block containing required word must first be read from Main Memory and loaded into Cache.
- The direct-mapping technique is easy to implement, but it is not very flexible.

Associative Mapping



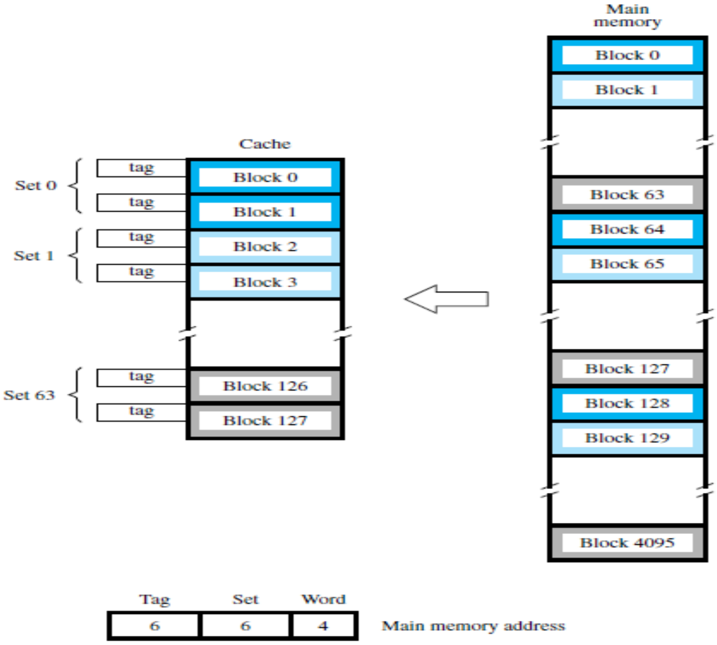
Associative Mapping

- Main Memory block can be placed into any cache position.
- Memory address is divided into two fields:

  - Low order 4 bits identify the word within a block.

  - High order 12 bits or tag bits identify a Memory block when it is resident in the cache.
- Flexible, and uses cache space efficiently.
- Replacement algorithms can be used to replace an existing block in the cache when the cache is full.
- Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.

Set Associative Mapping with 2 Blocks /Set



Set Associative Mapping

- Blocks of cache are grouped into sets.
- Mapping function allows a block of the Main Memory to reside in any block of a specific set.
- Divide the cache into 64 sets, with two blocks per set.
- Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.
- Memory address is divided into three fields:

  - 6 bit field determines the set number.

  - High order 6 bit fields are compared to tag fields of two blocks in a set.
- Set-associative mapping is combination of direct and associative mapping.
- Number of blocks per set is a design parameter.

  - One extreme is to have all the blocks in one set, requiring no set bits (fully associative mapping).

  - Other extreme is to have one block per set, is same as direct mapping.

Write Buffer

Write-through:

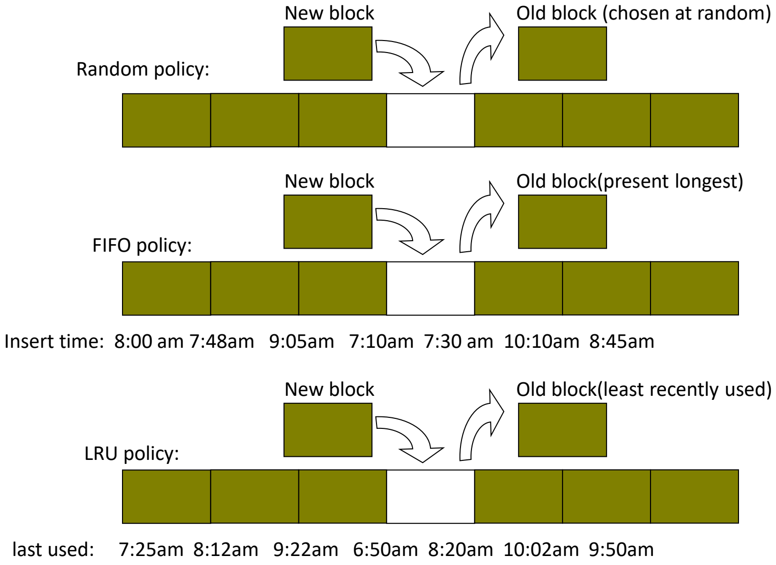
- Each write operation involves writing to the Main Memory.
- If the Processor has to wait for the write operation to be complete, it slows down the Processor.
- Processor does not depend on the results of the write operation.
- Write buffer can be included for temporary storage of write requests.
- Processor places each write request into the buffer and continues execution.
- If a subsequent Read request references data which is still in the write buffer, then this data is referenced in the write buffer.

Write-back:

- Block is written back to the Main Memory when it is replaced.
- If the Processor waits for this write to complete, before reading the new block, it is slowed down.
- Fast write buffer can hold the block to be written, and the new block can be read first.

Replacement Algorithms

- First-in, First-out(FIFO): Evict the block that has been in the cache the longest
- Least recently used (LRU): Evict the block whose last request occurred furthest in the past.
- Random: Choose a block at random to evict from the cache.



New block

Old block (chosen at random)

Random policy:

New block

Old block(present longest)

FIFO policy:

Insert time: 8:00 am 7:48am 9:05am 7:10am 7:30 am 10:10am 8:45am

New block

Old block(least recently used)

LRU policy:

last used: 7:25am 8:12am 9:22am 6:50am 8:20am 10:02am 9:50am

The Random, FIFO, and LRU Block replacement

Complexity of implement(Random)

- It only requires a random or pseudo-random number generator.
- Overhead is an O(1) additional amount of work per replacement.
- Makes no attempt to take advantage of any temporal or spatial localities.

Complexity of implement(FIFO)

- FIFO strategy just requires a queue Q to store references to blocks in cache
- Blocks are enqueued in Q
- Simply performs a dequeue operation on Q to determine which Block to evict.
- This policy requires O(1) additional work per block replacement
- Try to take advantage of temporal locality

Complexity of implement(LRU)

- Implementing the LRU strategy requires the use of a priority queue Q
- When insert a Block in Q or update its key, the Block is assigned the highest key in Q
- Each Block request and Block replacement is O(1) if Q is implemented with a sorted sequence based on a linked list.
- Because of the constant-time overhead and extra space for the priority Queue Q, make this policy less attractive from a practical point of view.

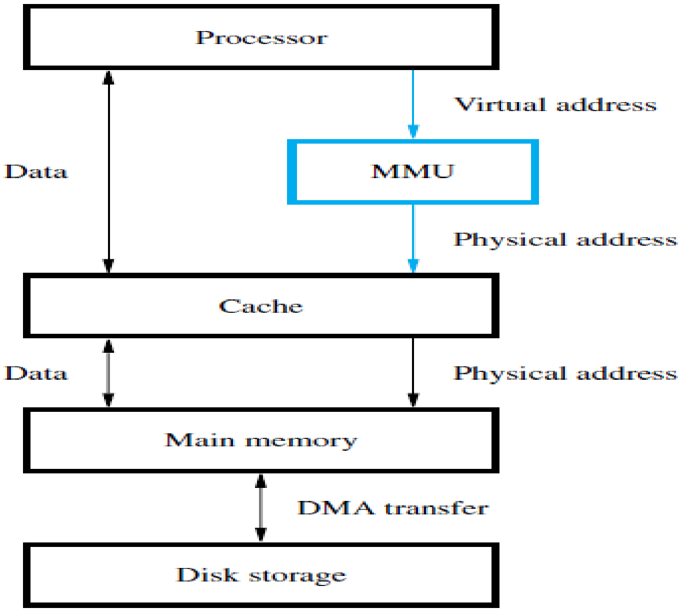
Virtual Memory

- In most Computers, the Physical Main Memory is not as large as Address Space of the Processor.
- A Program, if it does not completely fit into Main Memory, parts of it currently being executed are in Main Memory and remaining portion is stored in Secondary Storage such as Hard disk.
- When a new part of program is to be brought into Main Memory for execution and if the Memory is full, it must replace another part which is already is in Main Memory.
- As this Secondary Storage is not actually part of System Memory, so for CPU, the extended portion of Secondary Storage is Virtual Memory.

- Automatically move Program and Data Blocks into Physical Memory from Secondary Memory when they are required for execution.
- Virtual Memory is used to Logically extend the size of Main Memory.
- When Virtual Memory is used, the Address field is Virtual Address or Logical Address.
- A special Hardware unit knows as Memory Management Unit (MMU) translates Virtual Address into Physical Address.

- Processor makes reference to instructions and data in an address space that is independent of available Physical Main Memory space.
- The binary addresses that Processor issues for either instructions or data are called Virtual or Logical addresses.
- These addresses are translated into Physical addresses by a combination of hardware and software actions.
- If a Virtual address refers to a part of program or data space that is in Physical Memory, then contents of location in Main Memory are accessed immediately.
- Otherwise, contents of referenced address must be brought into a suitable location in Memory before they can be used.

Virtual Memory Organization



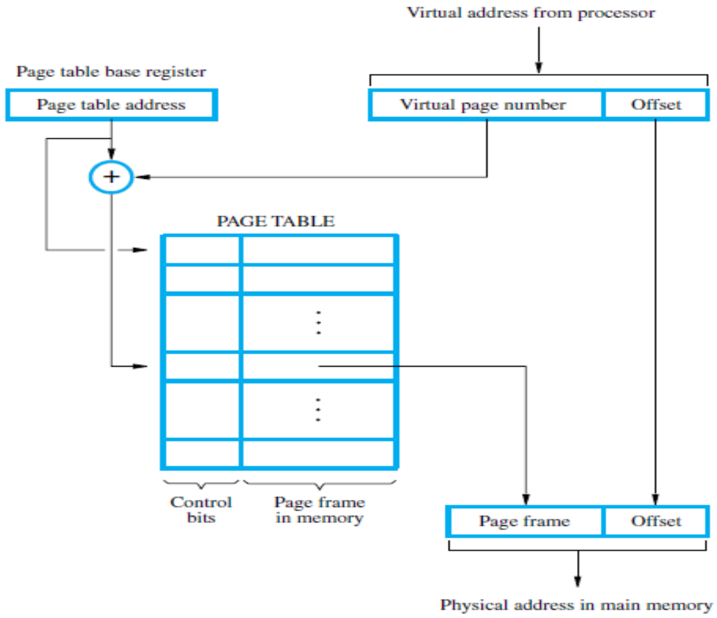
- A special hardware unit, called *Memory Management Unit* (MMU), keeps track of which parts of Virtual Address space are in Physical Memory.
- When desired data or instructions are in Main Memory, MMU translates Virtual Address into corresponding Physical Address.
- Then, requested Memory access proceeds in the usual manner.
- If data are not in Main Memory, MMU causes the OS to transfer data from disk to Memory.
- Such transfers are done using DMA scheme.

Address Translation

- To translate Virtual Addresses into Physical addresses, assume that all programs and data consist of fixed-length units called Pages.
- Page consists of a Block of words that occupy contiguous locations in the Main Memory.
- Pages range from 2K to 16K bytes in size.
- They constitute basic unit of info transferred between Main Memory and Disk whenever MMU determines that a transfer is required.
- Pages should not be too small, because access time of a disk is much longer (several mS) than access time of Main Memory (several nS).

- It takes a large amount of time to locate data on disk, but once located, data can be transferred at a rate of several MB per second.
- But if Pages are too large, it is possible that a large portion of a Page may not be used.
- Leads to waste of Memory space.
- The Virtual-Memory mechanism bridges size and speed gaps between Main Memory and Secondary Storage and is usually implemented in part by Software techniques.

Virtual-Memory Address Translation



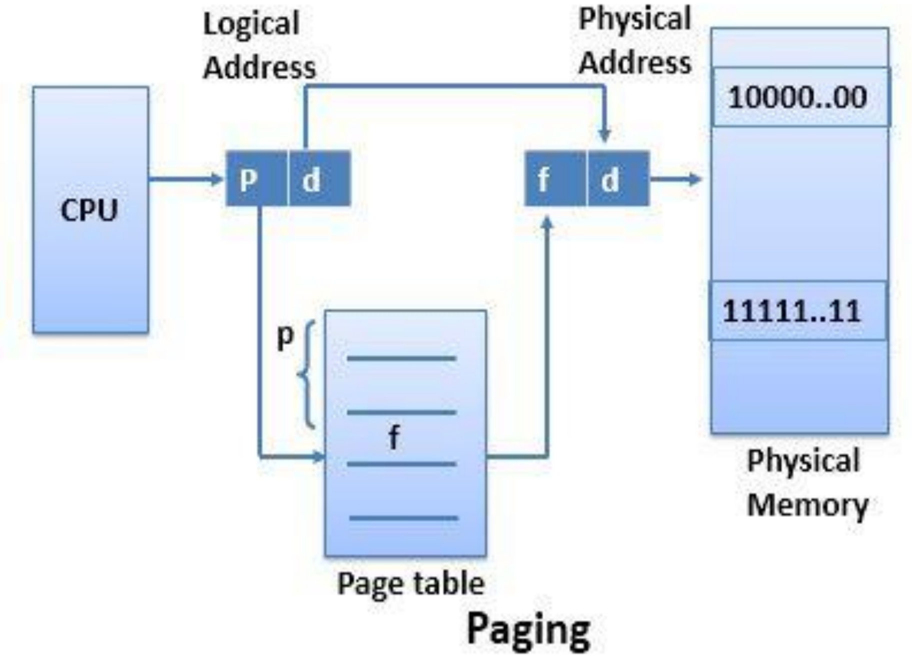
Paging

- This is a Virtual Memory address-translation method based on the concept of fixed-length Pages.
- Each Virtual Address generated by Processor is inferred as a Virtual Page Number (high-order bits) followed by an Offset (low-order bits) that specifies location of a Byte (or Word) within a Page.
- Information about Main Memory location of each Page is kept in a Page Table.

- This information includes Main Memory Address where Page is stored and current status of Page.
- An area in Main Memory that can hold one Page is called a Page Frame.
- Starting address of Page Table is kept in a Page Table Base Register.
- By adding Virtual Page number to contents of this register, address of corresponding entry in Page Table is obtained.
- Contents of this location give starting address of Page if that Page currently resides in Main Memory.

- Each entry in Page Table also includes some Control bits that describe Status of Page while it is in Main Memory.
- One bit indicates Validity of Page, that is, whether the Page is actually loaded in Main Memory.
- It allows OS to invalidate the Page without actually removing it.
- Another bit indicates whether the Page has been modified during its residency in the Memory.

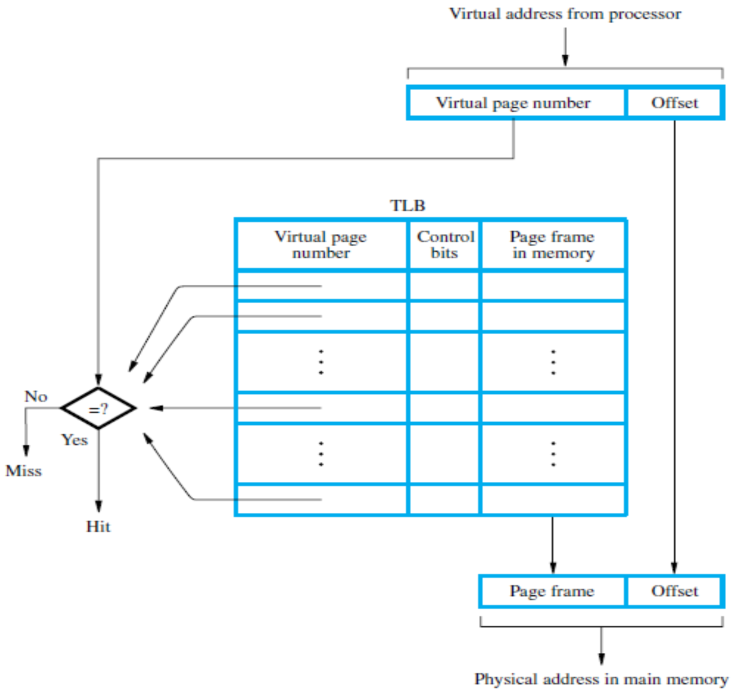
- This information is needed for the Page to be written back to disk before it is removed from Main Memory to make room for another Page.
- Other control bits indicate various restrictions that may be imposed on accessing the Page.
- For example, a program may be given full read and write permission, or it may be restricted to read accesses only.



Translation Lookaside Buffer

- Page Table information is used by MMU for every read and write access.
- Ideally, Page Table should be situated within MMU.
- Unfortunately, the Page Table may be rather large.
- Since MMU is normally implemented as part of Processor chip, it is impossible to include complete Table within MMU.
- Instead, a copy of only a small portion of Table is placed within MMU, and the complete Table is kept in Main Memory.
- The portion within MMU consists of entries corresponding to most recently accessed Pages.
- They are stored in a small Table, usually called *Translation Lookaside Buffer* (TLB).

- TLB functions as a cache for Page Table in Main Memory.
- Each entry in TLB includes a copy of information in corresponding entry in Page Table.
- In addition, it includes Virtual address of Page, which is needed to search TLB for a particular Page.
- A possible organization of a TLB uses the Associative-Mapping technique.
- Set-Associative mapped TLBs are also found in commercial products.



- Address translation proceeds as follows:
  - Given a Virtual address, MMU looks in TLB for the referenced Page.
  - If Page Table entry for this Page is found in TLB, Physical address is obtained immediately.
  - If there is a Miss in TLB, then required entry is obtained from Page Table in Main Memory and TLB is updated.

- Need to ensure contents of TLB are always same as contents of Page Tables in Memory.
- When OS changes contents of a Page Table, it must simultaneously invalidate corresponding entries in TLB.
- One of the control bits in TLB is provided for this purpose.
- When an entry is invalidated, TLB acquires new information from Page Table in Memory as part of the MMU's normal response to Access Misses.

Page Faults

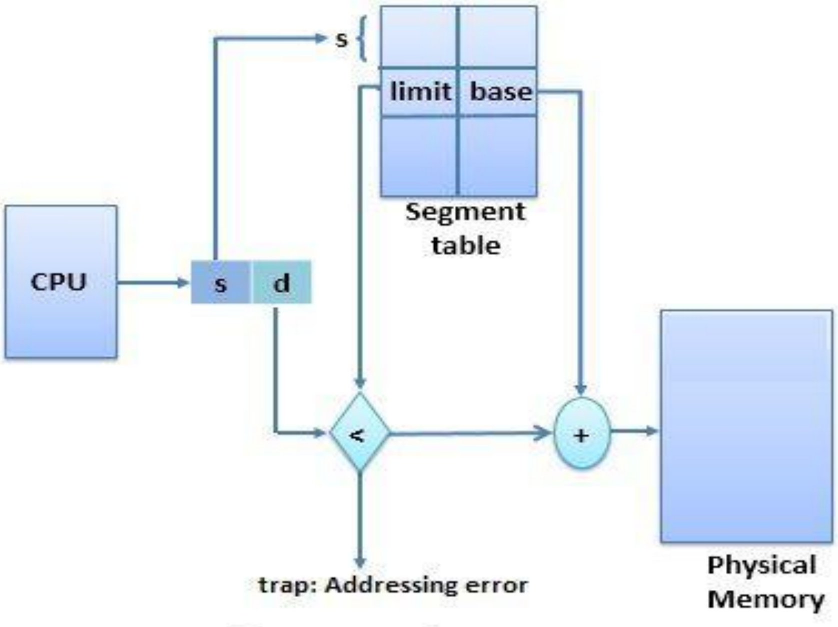
- When a program generates an access request to a Page that is not in the Main Memory, a Page Fault is said to have occurred.
- The entire Page must be brought from the disk into the Memory before access can proceed.
- When it detects a Page fault, the MMU asks the OS to intervene by raising an Exception (Interrupt).
- Processing of program that generated Page fault is interrupted, and control is transferred to the OS.
- OS copies requested Page from disk into Main Memory.
- Since this process involves a long delay, OS may begin execution of another program whose Pages are in Main Memory.

- When Page transfer is completed, execution of interrupted program is resumed.
- When MMU raises an interrupt to indicate a Page fault, the instruction that requested Memory access may have been partially executed.
- Need to ensure that interrupted program continues correctly when it resumes execution.
- There are two options:
  - Either execution of interrupted instruction continues from point of interruption, or instruction must be restarted.
  - Design of a particular Processor dictates which of these two options is used.

Segmentation

- Segmentation is also a Memory management scheme.
- It supports the user's view of the Memory.
- The process is divided into variable size Segments and loaded to the logical Memory address space.
- The Logical address space is the collection of variable size Segments.
- Each Segment has its name and length.
- For the execution, the Segments from Logical Memory space are loaded to the Physical Memory space.

- The address specified by the user contain two quantities, Segment name and Offset.
- The Segments are numbered and referred by the Segment number instead of Segment name.
- This Segment number is used as an index in the Segment table, and Offset value decides the length or limit of the Segment.
- The Segment number and the Offset together combine to generates the address of the Segment in the Physical Memory space.



Segmentation

Key Differences Between Paging and Segmentation

- The basic difference between Paging and Segmentation is that a Page is always of fixed block size whereas, a Segment is of variable size.
- Paging may lead to Internal Fragmentation as Page is of fixed block size, but it may happen that the process does not acquire the entire block size which will generate the internal fragment in Memory.
- Segmentation may lead to External Fragmentation as Memory is filled with variable sized blocks.

- In Paging the user only provides a single integer as the address which is divided by the hardware into a Page number and Offset.
- On the other hands, in Segmentation the user specifies address in two quantities i.e. Segment number and Offset.
- The size of Page is decided or specified by hardware.
- But, size of Segment is specified by user.
- In Paging, Page table maps the logical address to the Physical address, and it contains base address of each page stored in the frames of Physical Memory space.
- However, in Segmentation, Segment table maps Logical address to Physical address, and it contains Segment number and Offset .