IMPORTANT QUESTIONS & ANSWERS

Department of Computer Science and Engineering

SUBJECT CODE: CS8491

SUBJECT NAME: COMPUTER ARCHITECTURE

Regulation: 2017 **Semester and Year: IV and II**

ANNA UNIVERSITY, CHENNAI-25
SYLLABUS COPY - REGULATION 2017

CS8491	**COMPUTER ARCHITECTURE**	**L T P C**
		3 0 0 3

UNIT I OVERVIEW & INSTRUCTIONS 9

Eight ideas – Components of a computer system – Technology – Performance – Power wall – Uniprocessors to multiprocessors; Instructions – operations and operands – representing instructions – Logical operations – control operations – Addressing and addressing modes.

UNIT II ARITHMETIC OPERATIONS 7

ALU - Addition and subtraction – Multiplication – Division – Floating Point operations – Subword parallelism.

UNIT III PROCESSOR AND CONTROL UNIT 11

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

UNIT IV PARALLELISM 9

Instruction-level-parallelism – Parallel processing challenges – Flynn's classification – Hardware multithreading – Multicore processors

UNIT V MEMORY AND I/O SYSTEMS 9

Memory hierarchy - Memory technologies – Cache basics – Measuring and improving cache performance - Virtual memory, TLBs - Input/output system, programmed I/O, DMA and interrupts, I/OM processors.

TOTAL: 45 PERIODS

TEXT BOOK:

1. David A. Patterson and John L. Hennessey, “Computer Organization and Design’, Fifth edition, Morgan Kauffman / Elsevier, 2014.

REFERENCES:

1. V.Carl Hamacher, Zvonko G. Varanesic and Safat G. Zaky, “Computer Organisation“, VI edition, Mc Graw-Hill Inc, 2012.
2. William Stallings “Computer Organization and Architecture”, Seventh Edition, Pearson Education, 2006.
3. Vincent P. Heuring, Harry F. Jordan, “Computer System Architecture”, Second Edition, Pearson Education, 2005.
4. Govindarajalu, “Computer Architecture and Organization, Design Principles and Applications", first edition, Tata Mc Graw Hill, New Delhi, 2005.
5. John P. Hayes, “Computer Architecture and Organization”, Third Edition, Tata Mc Graw Hill, 1998.
6. http://nptel.ac.in/.

TABLE OF CONTENT

AIM AND OBJECTIVE OF THE SUBJECT

Aim:

To discuss the basic structure of a digital computer and to study in detail the organization of the Control unit, the Arithmetic and Logical unit, the Memory unit and the I/O unit.

Objectives:

- To make students understand the basic structure and operation of digital computer.
- To understand the hardware-software interface.
- To familiarize the students with arithmetic and logic unit and implementation of fixed point and floating-point arithmetic operations.
- To expose the students to the concept of pipelining.
- To familiarize the students with hierarchical memory system including cache memories and virtual memory.
- To expose the students with different ways of communicating with I/O devices and standard I/O interfaces

DETAILED LESSON PLAN

Text Books:

T1: David A. Patterson and John L.Hennessey, "Computer Organization and Design" Morgan Kauffman/Elsevier, Fifth Edition, 2014

Reference Books:

R1: V.Carl Hamachar, Zvonko G.Varanesic and Safar G.Zaky, "Computer Organization", VI edition, McGraw-Hill Inc, 2012

R2: William Stallings, "Computer Organization and Architecture", Seventh Edition. Pearson Education, 2006

R3: Vincent P.Heuring, Harry F.Jordan, "Computer System Architecture", Second Edition, Pearson Education, 2005

R4: John P.Hayes," Computer Architecture and Organization ", third Edition, Tata McGraw Hill, 1998

S. No.	Unit No.	Topic / Portions to be Covered	Hours Required	Cumulative Hours	Books Referred
UNIT I OVERVIEW AND INSTRUCTIONS					
1	1	Eight Ideas, Components	1	1	T1
2	1	Technology, Performance,	1	2	R1
3	1	Power Wall, Uniprocessor to multiprocessors	1	3	R1
4	1	Instructions Operations and operands. Representing Instructions	2	5	R1
5	1	Logical operations	1	6	T1,R1
6	1	Control operations	1	7	T1,R1
7	1	Addressing and Addressing modes	2	9	T1,R1
UNIT II ARITHMETIC OPERATIONS					
8	2	ALU	1	10	T1,R1
9	2	Addition and Subtraction	2	12	T1,R1
10	2	Multiplication	1	13	T1,R1
11	2	Division	1	14	T1
12	2	Floating Point operations	1	15	T1,R1

S. No.	Unit No.	Topic / Portions to be Covered	Hours Required	Cumulative Hours	Books Referred
13	2	Subword parallelism	1	16	T1,R1
UNIT III PROCESSOR AND CONTROL UNIT					
14	3	Basic MIPS implementation	1	17	T1,R1
15	3	Building datapath	1	18	T1,R1
16	3	Control implementation scheme	2	20	T1,R1
17	3	Pipelining and Pipelined datapath	2	22	T1,R1
18	3	Handling Data hazards	2	24	T1,R1
19	3	Control Hazards	2	26	R1
20	3	Exceptions	1	27	R1
UNIT IV PARALLELISM					
21	4	Instruction Level parallelism	2	29	T1
22	4	Parallel Processing challenges	1	30	T1
23	4	Flynn classification	2	32	T1
24	4	MISD & MIMD	2	34	T1
25	4	Hardware multithreading	1	35	T1
26	4	Multicore Processors	1	36	T1
UNIT V MEMORY AND I/O SYSTEMS					
27	5	Memory Hierarchy & Memory Technologies	2	38	T1,R1
28	5	Flash and Disk Memory Technologies	1	39	T1,R1
29	5	Cache Basics	1	40	T1,R1
30	5	Measuring and improving cache performance	1	41	T1
31	5	Virtual memory, TLBs	1	42	T1
32	5	Input/ Output System & Programmed I/O	2	44	T1,R1
33	5	DMA and Interrupts	1	45	T1,R1
34	5	I/O Processors	1	46	T1,R1

UNIT I

UNIT I OVERVIEW & INSTRUCTIONS 9

Eight ideas – Components of a computer system – Technology – Performance – Power wall – Uniprocessors to multiprocessors; Instructions – operations and operands – representing instructions – Logical operations – control operations – Addressing and addressing modes.

PART A

1. State Amdahl's law? (Nov/Dec 2014)

Amdahl's law is used to find the execution time of a program after making the improvement. It can be represented as

Execution time after improvement = Execution time affected by improvement / Amount of improvement + Execution time unaffected.

Amdahl's law states that in parallelization, if P is the proportion of a system or program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speed up that can be achieved using N number of processors is

1/((1-P)+(P/N).

2. What is register indirect addressing mode? When it is used? (Nov/Dec 2013)

In this mode the instruction specifies a register in the CPU whose contents give the effective address of the operand in the memory. The selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

Uses: The use of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify

a memory address directly. Therefore EA = the address stored in the register R.

- Operand is in memory cell pointed by contents of register R

Example: Add (R2), R0

3. Write the CPU performance equation. (May/June 2014) or How CPU execution time for a program is calculated.(Nov/Dec 2016)

The Classic CPU Performance Equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time.

CPU time = Instruction count * CPI * Clock cycle time (or)

$$\text{CPU time} = \frac{\text{Instruction count} * \text{CPI}}{\text{Clock rate}}$$

CPI:

The term Clock Cycles per Instruction which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI.

CPI = CPU clock cycles/Instruction count.

4. What are the fields in an MIPS instruction? (April /May 2015)

MIPS fields are

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

Where,

op : Basic operation of the instruction, traditionally called the opcode.

rs : The first register source operand.

rt : The second register source operand.

rd : The register destination operand. It gets the result of the operation.

shamt : Shift amount.

funct : Function.

5. Define word length. (Nov/Dec 2011)

A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. The number of bits in a word (the word size, word width, or word length) is an important characteristic of any specific processor design or computer architecture.

The size of a word is reflected in many aspects of a computer's structure and operation; the majority of the registers in a processor are usually word sized and the largest piece of data that can be transferred to and from the working memory in a single operation is a word in many (not all) architectures.

6. What are the merits and demerits of single address instructions? (Nov/Dec 2011)

The machine will be slower in this case but not in all cases. Programs are now even shorter. The registers may be used for temporary results which are not needed immediately or for holding frequently used operands e.g. the end count in a "for" loop.

7. What do you mean by relative addressing mode? (Nov/Dec 2014, May/June 2012)

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. Effective address is defined as the memory address obtained from the computation dictated by the given addressing mode.

Example:

Bne $s0,$s1,Exit go to Exit if$s0≠$s1

In this example branch address is calculated by adding the PC value with the constant in the instruction.

8. Distinguish Pipelining from Parallelism. (April/May 2015)

In order to increase the instruction throughput, high performance processors make extensive use of a technique called pipelining. A pipelined processor doesn't

wait until the result from a previous operation has been written back into the register files or main memory - it fetches and starts to execute the next instruction as soon as it has fetched the first one and dispatched it to the instruction register. So a pipelined processor will start fetching the next instruction from memory as soon as it has latched the current instruction in the instruction register.

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism.

9. Define auto increment and auto decrement addressing mode. (April/May 2010)

Auto Increment mode:

The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list. The Auto increment mode is written as **(Ri) +.** As a companion for the Auto increment mode, another useful mode accesses the items of a list in the reverse order.

Auto Decrement mode:

The contents of a register specified in the instruction are first automatically decremented and is then used as the effective address of the operand. We denote the Auto decrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write **- (Ri).**

10. What is Instruction set architecture?(Nov/Dev 2016)

Instruction set architecture also called architecture. An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

PART B

1. EIGHT IDEAS IN COMPUTER ARCHITECTURE

❖ **Explain the basic concepts of Eight ideas in computer architecture. Apr/May:2015)(8)**

a. DESIGN FOR MOORE'S LAW:

The one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resource double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity. Moore's Law made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Computer architects must anticipate this rapid change.

Icon used: "up and to the right" Moore's Law graph represents designing for rapid change.

b. USE ABSTRACTION TO SIMPLIFY DESIGN:

Both computer architects and programmers had to invent techniques to make themselves more productive. A major productivity technique for hardware and soft ware is to use abstractions to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels.

Icon used: abstract painting icon.

c. **MAKE THE COMMON CASE FAST:**

Making the common case fast will tend to enhance performance better than optimizing the rare case. The common case is often simpler than the rare case and it is often easier to enhance. Common case fast is only possible with careful experimentation and measurement.

Icon used: sports car the icon for making the common case fast (as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan.)

d. **PERFORMANCE VIA PARALLELISM:**

Computer architects have offered designs that get more performance by performing operations in parallel.

Icon Used: multiple jet engines of a plane are the icon for parallel performance.

e. PERFORMANCE VIA PIPELINING:

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Pipelining improves performance by increasing instruction

throughput. For example, before fire engines, a human chain can carry a water source to fire much more quickly than individuals with buckets running back and forth.

Icon Used: pipeline icon is used. It is a sequence of pipes, with each section representing one stage of the pipeline.

f. PERFORMANCE VIA PREDICTION:

In some cases it can be faster on average to guess and start working rather than wait until you know for sure. This mechanism to recover from a misprediction is not too expensive and the prediction is relatively accurate.

Icon Used: fortune-teller's crystal ball

g. HIERARCHY OF MEMORIES :

Programmers want memory to be fast, large, and cheap memory speed often shapes performance, capacity limits the size of problems that can be solved, the cost of memory today is often the majority of computer cost.

Architects have found that they can address these conflicting demands with a hierarchy of memories the fastest, smallest, and most expensive memory per bit is at the top of the hierarchy the slowest, largest, and cheapest per bit is at the bottom.

Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy.

Icon Used: a layered triangle icon represents the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

h. DEPENDABILITY VIA REDUNDANCY:

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, systems can made dependable by including redundant components. These components can take over when a failure occurs and to help detect failures.

Icon Used: the tractor-trailer, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails.

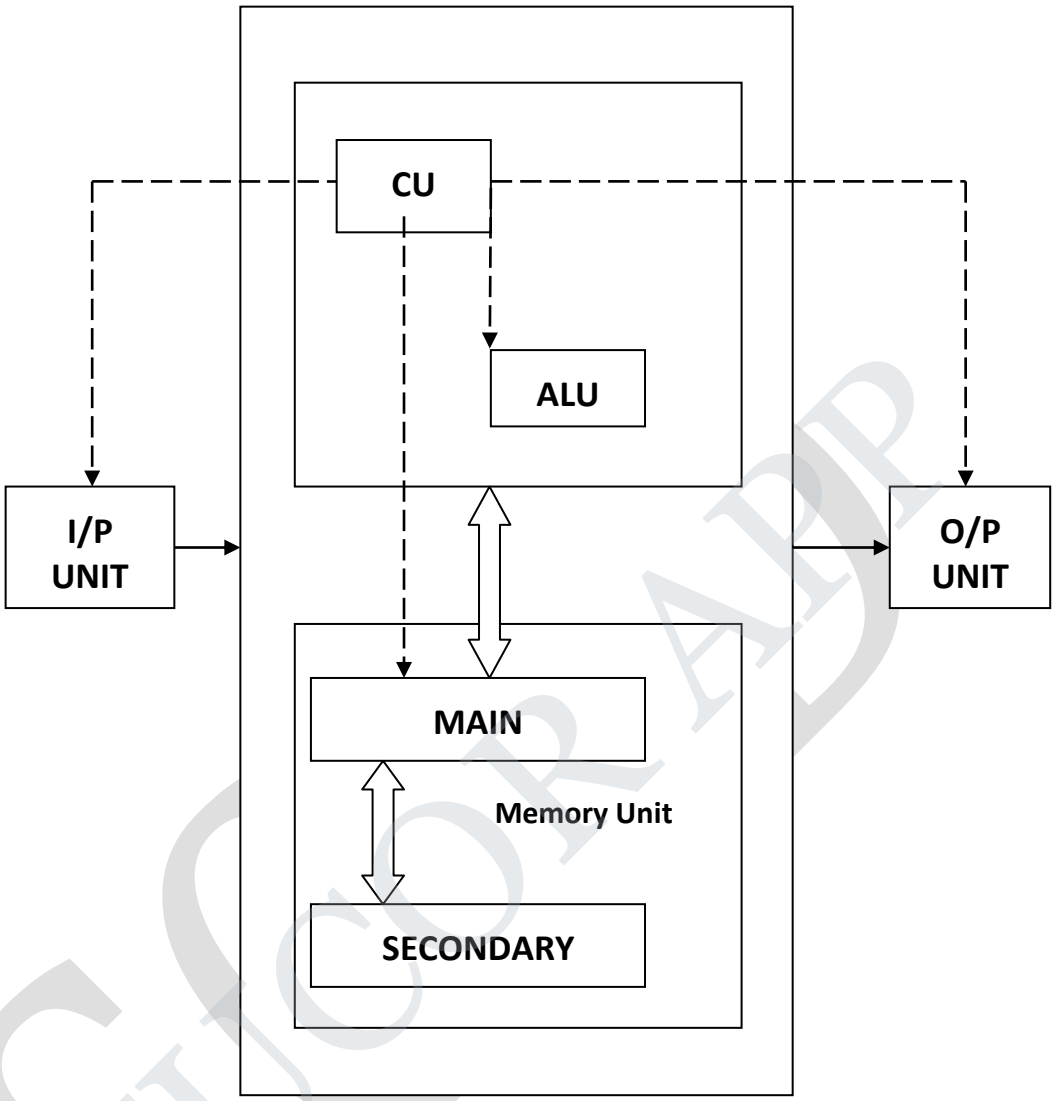
2. COMPONENTS OF A COMPUTER SYSTEM

- **Explain about von Neumann architecture (Nov/Dec 14)**
- **Explain in detail the various components of computer system with neat diagram(Nov/Dec 15, May/June 16)**

A computer consists of five functionally independent main parts. They are

1. Input
2. Memory
3. Arithmetic and logic
4. Output
5. Control unit

Basic Functional Units of a Computer



The computer accepts programs and the data through an input and stores them in the memory. The stored data are processed by the arithmetic and logic unit under program control. The processed data is delivered through the output unit. All above activities are directed by control unit.

a. Input Unit

The computer accepts coded information through input unit. The input can be from human operators, electromechanical devices such as keyboards or from other computer over communication lines.

Examples of input devices are Keyboard, joysticks; trackballs and mouse are used as graphic input devices in conjunction with display.

Keyboard

- It is a common input device.
- Whenever a key is pressed; the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over cable to the memory of the computer.

b. Memory Unit

Memory unit is used to store programs as well as data. Memory is classified into primary and secondary storage.

Primary Storage

It also called main memory. It operates at high speed and it is expensive. It is made up of large number of semiconductor storage cells, each capable of storing one bit of information. These cells are grouped together in a fixed size called word. This facilitates reading and writing the content of one word (n bits) in single basic operation instead of reading and writing one bit for each operation.

Secondary Storage

It is slow in speed. It is cheaper than primary memory. Its capacity is high. It is used to store information that is not accessed frequently. Various secondary devices are magnetic tapes and disks, optical disks (CD-ROMs), floppy etc.

c. Arithmetic and Logic Unit

Arithmetic and logic unit (ALU) and control unit together form a processor. Actual execution of most computer operations takes place in arithmetic and logic unit of the processor. Example: Suppose two numbers located in the memory are to be

added. They are brought into the processor, and the actual addition is carried out by the ALU.

Registers:

Registers are high speed storage elements available in the processor. Each register can store one word of data. When operands are brought into the processor for any operation, they are stored in the registers. Accessing data from register is faster than that of the memory.

d. Output unit

The function of output unit is to produce processed result to the outside world in human understandable form. Examples of output devices are Graphical display, Printers such as inkjet, laser, dot matrix and so on. The laser printer works faster.

e. Control unit

Control unit coordinates the operation of memory, arithmetic and logic unit, input unit, and output unit in some proper way. The **control unit** issues **control signals** that cause the CPU (and other components of the computer) to fetch the instruction to the IR (Instruction Register) and then execute the actions dictated by the machine language instruction that has been stored there. Control units are well defined, physically separate unit that interact with other parts of the machine. A set of control lines carries the signals used for timing and synchronization of events in all units Example: Data transfers between the processor and the memory are controlled by the control unit through timing signals**. Timing signals** are the signals that determine when a given action is to take place.

Basic Operational Concept

Top-Level View

PC the **program counter** contains the address of the assembly language instruction to be executed next. IR the **instruction register** contains the binary word

corresponding to the machine language version of the instruction currently being executed.

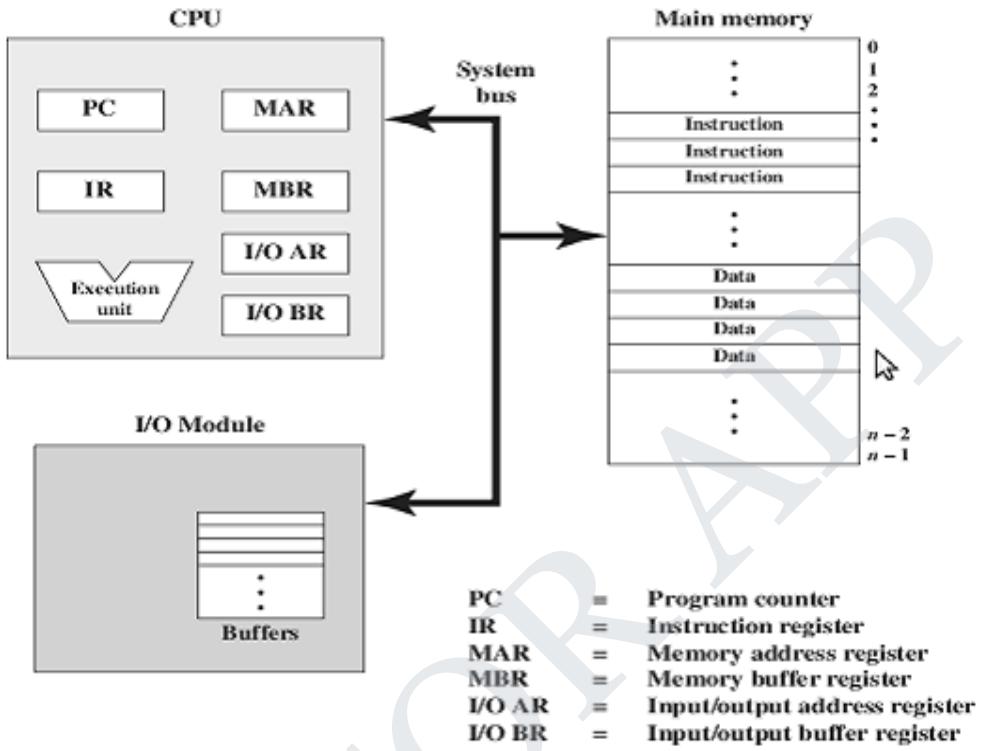


Figure 1.2 Computer Components

MAR the **memory address register** contains the address of the word in main memory that is being accessed. The word being addressed contains either data or a machine language instruction to be executed.

MBR the **memory buffer register** (also called MDR for memory data register) is the register used to communicate data to and from the memory.

The operation of a processor is characterized by a fetch-decode-execute cycle. In the first phase of the cycle, the processor fetches an instruction from memory. The address of the instruction to fetch is stored in an internal register named the program counter, or PC. As the processor is waiting for the memory to respond with the instruction, it increments the PC. This means the fetch phase of the next cycle will

fetch the instruction in the next sequential location in memory. In the decode phase the processor stores the information returned by the memory in another internal register, known as the instruction register, or IR. The IR now holds a single machine instruction, encoded as a binary number. The processor decodes the value in the IR in order to figure out which operations to perform in the next stage.

In the execution stage the processor actually carries out the instruction. This step often requires further memory operations; for example, the instruction may direct the processor to fetch two operands from memory, add them, and store the result in a third location (the addresses of the operands and the result are also encoded as part of the instruction). At the end of this phase the machine starts the cycle over again by entering the fetch phase for the next instruction. The CPU exchanges data with memory. For this purpose, it typically makes use of **two internal (to the CPU) register:**

- A memory address register (MAR), which specifies the address in memory for the next read or write, and
- A memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory.

An I/O addresses register (I/OAR) specifies a particular I/O device. An I/O buffer (I/OBR) register is used for the exchange of data between an I/O module and the CPU.A memory module consists of a set of locations, defined by sequentially numbered address. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa.

It contains internal buffers for temporarily holding these data until they can be sent on.

3a. TECHNOLOGY

❖ Technologies for Building Processors and Memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better compute.

The table shows the technology that has been used over time, with an estimate of the relative performance per unit cost for each technology. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

A **transistor** is simply an on/off switch controlled by electricity. The integrated circuit (IC) combined dozens to hundreds of transistors into a single chip. When Gordon Moore predicted the continuous doubling of resources, he was predicting the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective very large scale is added to the term, creating the abbreviation **VLSI**, for **very large-scale integrated circuit**. The rate of increasing integration has been remarkably stable.

Figure 1.3 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

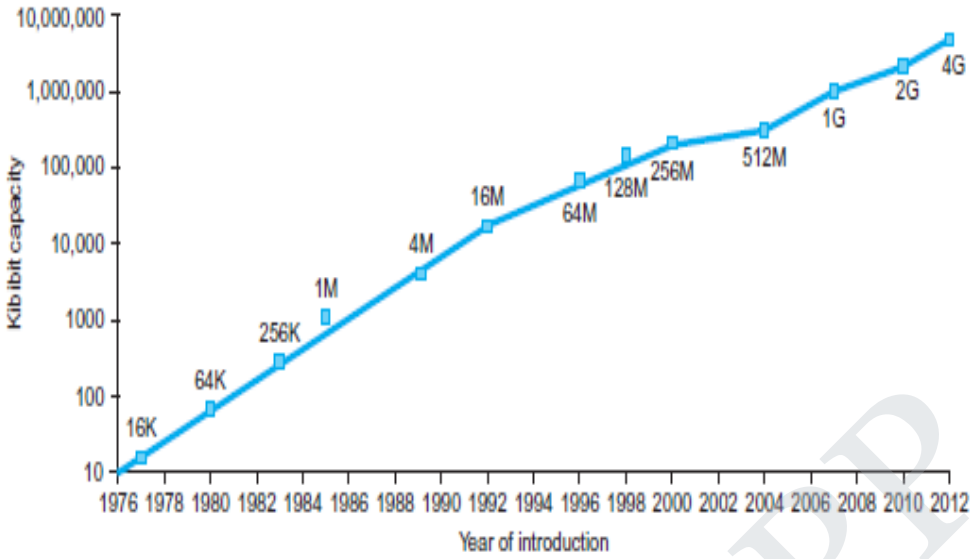


Figure 1.3 Technologies for Building Processors and Memory

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under special conditions (as a switch)

Transistors fall in the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package. The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers.

Figure 1.4 shows that process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 12–24 inches long. An ingot is finely sliced into **wafers** no more than 0.1 inches thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed on each wafer, creating transistors, conductors, and insulators.

Today's integrated circuits contain only one layer of transistors but may have from two to eight levels of metal conductor, separated by layers of insulators. A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can

result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or diced, into these components, called **dies** and more informally known as **chips**.

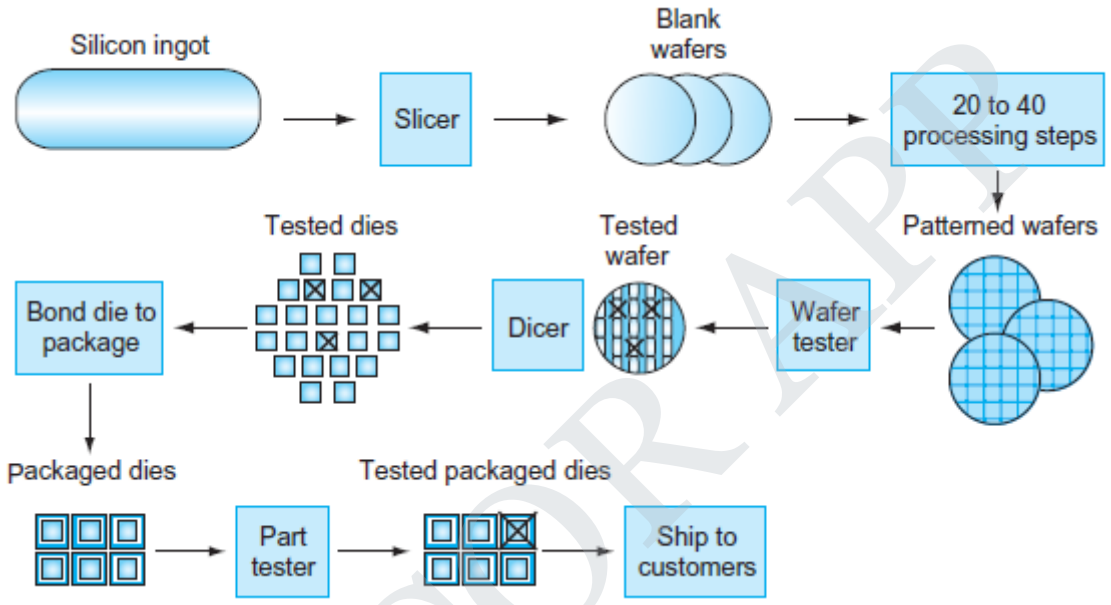


Figure 1.4 Chip Manufacturing Process.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of dies that fit on a wafer. To reduce the cost, using the next generation process shrinks a large die as it uses smaller sizes for both transistors and wires. This improves the yield and the die count per wafer. A 32-nanometer (nm) process was typical in 2012, which means essentially that the smallest

feature size on the die is 32 nm. Once we've found good dies, they are connected to the input/output pins of a package, using a process called bonding. These packaged parts are tested a final time,

3b. PERFORMANCE OF A COMPUTER SYSTEM

❖ **State the CPU performance equation and discuss the factors that affect the performance (Nov/Dec 14)**

Response time: The time between the start and the completion of an event also referred to as execution time.

Throughput: The total amount of work done in a given time. In comparing design alternatives, we often want to relate the performance of two different machines, say X and Y. The phrase X is faster than Y here it means that the response time or execution time is lower on X than on Y for the given task. In particular, X is *n* times faster than Y will mean

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Since execution time is the reciprocal of performance, the following relationship

$$n = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = \frac{\frac{1}{\text{Performance}_Y}}{\frac{1}{\text{Performance}_X}} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

The performance and execution time are reciprocals, increasing performance decreases execution time. Improve performance or Improve execution time when we mean increase performance and decrease execution time.

CPU Performance Equation:

All computers are constructed using a clock running at a constant rate. These discrete time events are called ticks, clock ticks, clock periods, clocks, cycles, or clock cycles. Computer designers refer to the time of a clock period by its duration (e.g., 1

ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

CPU time = CPU clock cycles for a program X Clock cycle time

CPU time = CPU clock cycle to exe a program / clock rate

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed, the instruction path length or instruction count (IC).

CPI is computed as:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction Count}}$$

By transposing instruction count in the above formula, clock cycles can be defined As IC * CPI. This allows us to use CPI in the execution time formula

$$\text{CPU time} = \text{Instruction Count} \times \text{Clock cycle time} \times \text{Cycles per Instruction}$$

$$\text{CPU time} = \frac{\text{Instruction Count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

CPU to calculate the number of total CPU clock cycles as,

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

Basic Performance Equation

Let T be the time required for the processor to execute a program in high level language. The compiler generates machine language object program corresponding to the source program. Assume that complete execution of the program requires the execution of N machine language instructions.

Consider that average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock

rate is R cycles per second, the program execution time is given by T = (N x S) / R this is often called Basic performance equation.

To achieve high performance, the performance parameter T should be reduced. T value can be reduced by reducing N and S, and increasing R.

- Value of N is reduced if the source program is compiled into fewer number of machine instructions.
- Value of S is reduced if instruction has a smaller no of basic steps to perform or if the execution of the instructions is overlapped.
- Value of R can be increased by using high frequency clock, ie. Time required to complete a basic execution step is reduced.
- N, S and R are dependent factors. Changing one may affect another.

Example :

Suppose the frequency of FP operations (other than FPSQR) = 25% Average CPI of FP operations = 4.0 Average CPI of other instructions = 1.33 Frequency of FPSQR = 2% CPI of FPSQR = 20 Assume that the two design alternatives are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the CPU performance equation.

$$\text{CPI}_{\text{original}} = \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}} \right)$$
$$= (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

$$\text{CPI}_{\text{with new FPSQR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}})$$
$$= 2.0 - 2\% \times (20 - 2) = 1.64$$

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

$$\text{Speedup}_{\text{new FP}} = \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}$$

$$= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23$$

3c. POWER WALL

❖ **Elaborate about power wall with neat sketch.**

The figure 1.5 shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 years. Both clock rate and power increased rapidly for decades, and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

Although power provides a limit to what we can cool, in the PostPC Era the really critical resource is energy. Battery life can trump performance in the personal mobile device, and the architects of warehouse scale computers try to reduce the costs of powering and cooling 100,000 servers as the costs are high at this scale. Just as measuring time in seconds is a safer measure of program performance than a rate like MIPS (see Section 1.10), the energy metric joules is a better measure than a power rate like watts, which is just joules/second.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is so-called dynamic energy— that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied.

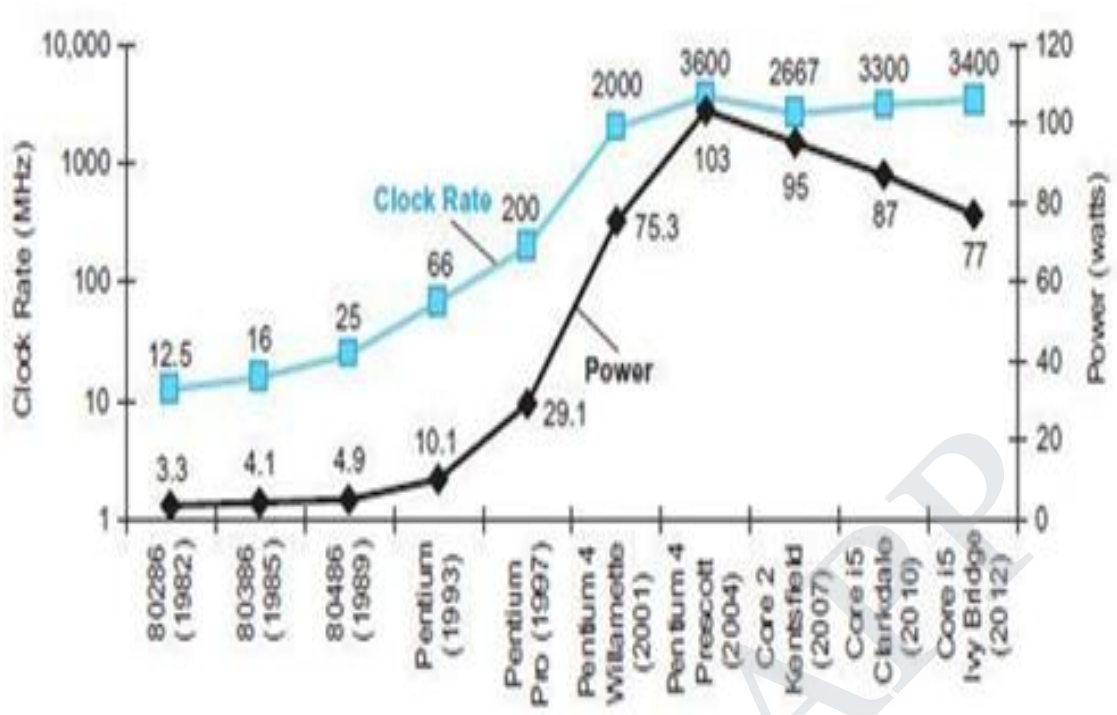


Figure 1.5 Clock rate and Power of eight generations

$$\text{Energy} \propto \text{Capacitive load X Voltage}^2$$

This equation is the energy of a pulse during the logic transition of 0 → 1 → 0 or 1 → 0 → 1. The energy of a single transition is then

$$\text{Energy} \propto \tfrac{1}{2} \text{ X Capacitive load X Voltage}^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions

$$\text{Power} \propto \tfrac{1}{2} \text{ X Capacitive load X Voltage}^2 \text{ X Frequency Switched}$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the fan-out) and the technology, which determines the capacitance of both wires and transistors.

From the Figure Energy and power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the

voltage squared. Typically, the voltage was reduced about 15% per generation. In 20 years, voltages have gone from 5 V to 1 V, which is why the increase in power is only 30 times.

3d. UNIPROCESSOR TO MULTIPROCESSOR

❖ **Explain the relationship between Uniprocessors to Multiprocessors. (8)**

The power limit has forced a dramatic change in the design of microprocessors.

Figure 1.6 shows the improvement in response time of programs for desktop Microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.

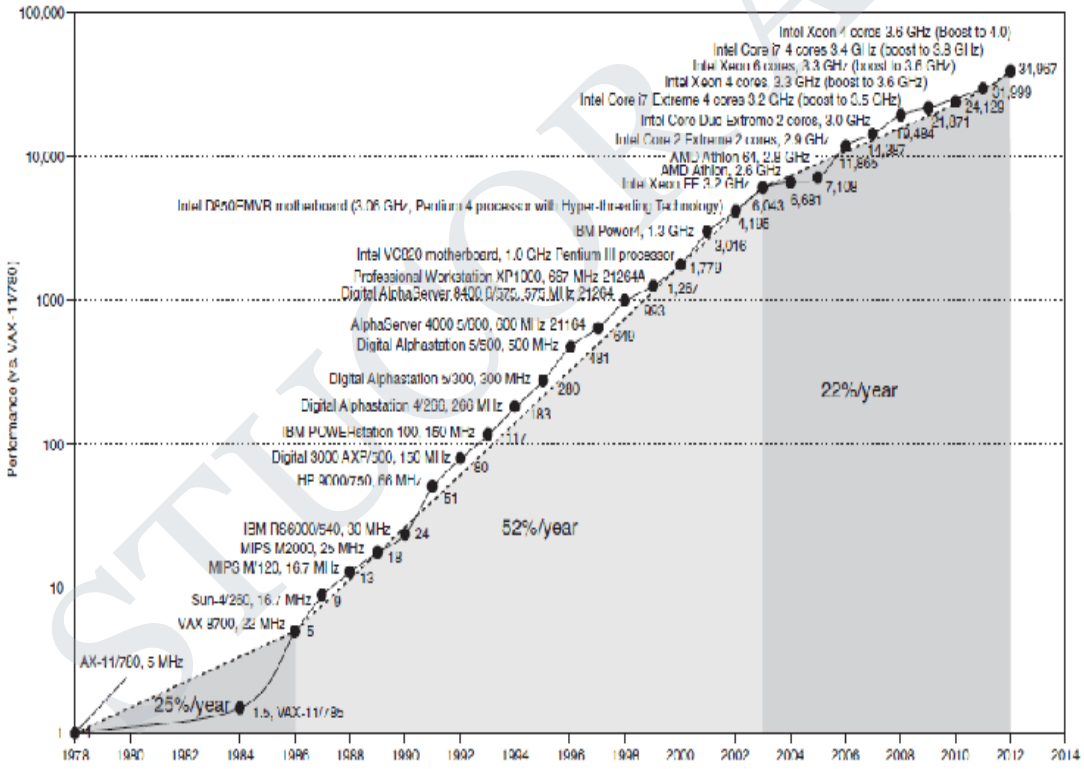


Figure 1.6 Growth in processor performance since the mid-1980s

Rather than continuing to decrease the response time of a single program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor

and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors. Hence, a "Quad core" microprocessor is a chip that contains four processors or four cores.

In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores increases.

Parallelism has always been critical to performance in computing, but it was often hidden. P**ipelining**, an elegant technique that runs programs faster by overlapping the execution of instructions. This is one example of instruction-level parallelism, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the "third rail" of computer architecture, for companies in the past that depended on such a change in behaviour failed .From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

Why has it been so hard for programmers to write explicitly parallel programs?

The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the Program need to be correct, solve an important problem, and provide a useful Interface to the people or other programs that invoke it, the program must also be fast. Otherwise, if we don't need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must schedule the sub-tasks.

If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must balance the load evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. Thus, care must be taken to reduce communication and synchronization overhead. For both this analogy and parallel programming, the

Challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. The challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

Multiprocessor:

A type of architecture that is based on multiple computing units. Some of the operations are done in parallel and the results are joined afterwards. There are many types of classifications for multiprocessor architectures, the most commonly known would be the Flynn Taxonomy. MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies.

To reduce confusion between the words processor and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors.

4. INSTRUCTIONS

- **How we can represent the instruction format in computer system (Apr/May 15)**
- **Discuss about the various techniques to represent instruction in a computer system.**

(April/May 2015). (16)

The words of a computer's language are called instructions, and its vocabulary is called an **instruction set**.

Category	Instructions	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands, data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands, data in registers
Data Transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Data from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant

Category	Instructions	Example	Meaning	Comments
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional Branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address

Figure 1.7 MIPS Assembly Language

J-Type

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the J-type, which consists of 6 bits for the operation field. Thus, **j 10000 # go to location 10000**

2	10000
6 bits	**26 bits**

Where the value of the jump opcode is 2 and the jump address is 10000

Techniques to Represent Instruction In A Computer System Instruction:

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. Each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the

instruction.

Since registers are referred to by almost all instructions, there must be a convention to map register names into numbers. In MIPS assembly language, registers $s0 to $s7 map onto registers 16 to 23, and registers $t0 to $t7 map onto registers 8 to 15. Hence, $s0 means register 16, $s1 means register 17, $s2 means register 18, . . . , $t0 means register 8, $t1 means register 9, and so on.

Translating a MIPS Assembly Instruction into a Machine Instruction

The real MIPS language version of the instruction represented symbolically as first as a combination of decimal numbers and then of binary numbers.

The decimal representation of the instruction **add $t0, $s1, $s2** is

0	17	18	8	0	32

Each of these segments of an instruction is called a field. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = $s1), and the third field gives the other source operand for the addition (18 = $s2). The fourth field contains the number of the register that is to receive the sum (8 = $t0). The fifth field is unused in this instruction, so it is set to 0.

Thus, this instruction adds the register $s1 to register $s2 and places the sum in register $t0. This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 Bits	5 Bits	5 Bits	5 Bits	5 Bits	6 Bits

This layout of the instruction is called the instruction format. All MIPS instructions are 32 bits long. We avoid that tedium by using a higher base than binary

that converts easily into binary. Since almost all computer data sizes are multiples of 4, hexadecimal (base 16) numbers are popular.

Hexa decimal	Binary	Hexa decimal	Binary	Hexa decimal	Binary	Hexa decimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c hex	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d hex	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a hex	1010_{two}	e hex	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b hex	1011_{two}	f hex	1111_{two}

Figure 1.8 The Hexadecimal to binary Conversion Table

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with ten, binary numbers with two, and hexadecimal numbers with hex. (If there is no subscript, the default is base 10.)

Binary to Hexadecimal and Back:

Convert the following hexadecimal and binary numbers into the other base:

- eca8 6420hex
- 0001 0011 0101 0111 1001 1011 1101 1111 two
-

MIPS Fields:

MIPS fields are given names to make them easier to discuss.

The meaning of each name of the fields in MIPS instructions:

op : Basic operation of the instruction, traditionally called the opcode.

rs : The first register source operand.

rt : The second register source operand.

rd : The register destination operand. It gets the result of the operation.

shamt : Shift amount.

funct : Function. This field selects the specific variant of the operation in the op field and is sometimes called the function code.

Today's computers are built on two key principles:

1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like numbers.

These principles lead to the **stored-program concept**; its invention let the computing genie out of its bottle. Specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called R-type (for register) or

R-format. A second type of instruction format is called I-type (for immediate) or I-format and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	Constant or Address
6 Bits	5 Bits	5 Bits	16 Bits

The 16-bit address means a load word instruction can load any word within a region of ±215 or 32,768 bytes (±213 or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than ±215. That more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	Constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Fig 1.9 MIPS Instruction Encoding.

TRANSLATING MIPS ASSEMBLY LANGUAGE INTO MACHINE LANGUAGE

If $t1 has the base of the array A and $s2 corresponds to h, the assignment statement

A[300] = h + A[300];

is compiled into

```
lw $t0,1200($t1)     # Temporary reg $t0 gets A[300]
add $t0,$s2,$t0      # Temporary reg $t0 gets h + A[300]
```

```
sw $t0,1200($t1)     # Stores h + A[300] back into A[300]
```

op	rs	rt	rd	Address/ funct	shamt
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

The lw instruction is identified by 35 in the first field (op). The base register 9 ($t1) is specified in the second field (rs), and the destination Register 8 ($t0) is specified in the third field (rt). The offset to select A[300] (1200 = 300 × 4) is found in the final field (address). The add instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to $s2, $t0, and $t0.

The sw instruction is identified with 43 in the first field. The rest of this final instruction is identical to the lw instruction. Since 1200ten = 0000 0100 1011 0000two, the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32

Name	Format	Example						Comments
								bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Figure 1.10 MIPS machine language

❖ **Consider a computer with three instruction classes and CPI measurements are given below and instruction counts for each instruction class for the same program from two different compilers are given. (Nov/Dec 2014). (6)**

(i) Which code sequence executes the more number of instructions?

(ii) Which code sequence will be faster?

(iii) What is the CPI for each sequence?

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

ode sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

SOLUTION:

(i) Code sequence 1:

Executes 2+1+2= 5 instructions

Code sequence 2:

Executes 4+1+1= 6 instructions

That is code sequence 2 executes more number of instructions than code sequence 1.

(i) The total number of clock cycles for each sequence can be found using the following equation.

CPU clock cycles = $\sum_{i=1}^{n}$ (CPI_i X C_i)

CPU clock cycles $_1$ = (2x1)+(1x2)+(2x3)=2+2+6=10 cycles.

CPU clock cycles $_2$ = (4x1)+(1x2)+(1x3)=4+2+3=9 cycles.

It is clear that code sequence 2 is faster than code sequence 1 even though it executes one extra instruction.

(ii) The CPI values can be computed by,

CPI = CPU clock cycles/Instruction count

CPI $_1$ = 10/5= 2

CPI_2 = 9/6 =1.5

5. LOGICAL OPERATIONS AND CONTROL OPERATIONS

❖ **Explain about the concepts of Logical operations and control operations.**

Logical Operations:

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation. It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and

unpacking of bits into words. These instructions are called logical operations.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or,ori
Bit-by-bit NOT	~	~	nor

The first class of such operations is called shifts. They move all the bits in a word to the left or right, filling the emptied bits with 0s.

For example, if register $s0 contains

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

and the instruction to shift left by 4 was executed, the new value would be:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called shift left logical (sll) and shift right logical (srl).

The following instruction performs the operation above, assuming that the original value was in register $s0 and the result should go in register $t2

```
sll $t2,$s0,4      # reg $t2 = reg $s0 << 4 bits shamt field in the R-format
                     used  in shift instructions, it stands for shift amount.
```

AND Operation

Another useful operation that isolates fields is **AND**. AND is a bit by bit operation that leaves a 1 in the result only if both bits of the operands are 1.

For example,

register $t2 contains 0000 0000 0000 0000 0000 1101 1100 0000_{two}

register $t1 contains 0000 0000 0000 0000 0011 1100 0000 0000_{two}

then, after executing the MIPS instruction

and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2

the value of register $t0 would be 0000 0000 0000 0000 0000 1100 0000 0000_{two}

AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a mask, since the mask "conceals" some bits.

OR Operation

It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1. To elaborate, if the registers $t1 and $t2 are unchanged from the preceding example, the result of the MIPS instruction

or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2

$t0 : 0000 0000 0000 0000 0011 1101 1100 0000_{two}

NOT Operation

NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa.

With the three-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT (A OR 0) = NOT (A). If the register $t1 is

unchanged from the preceding example and register $t3 has the value 0, the result of the MIPS instruction

nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)

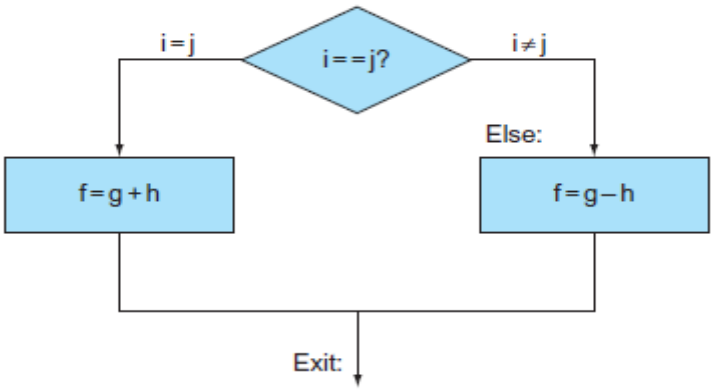
the value in register $t0 : 1111 1111 1111 1111 1100 0011 1111 1111_{two}

CONTROL OPERATIONS:

Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the if statement, sometimes combined with go to statements and labels. MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to.

Decision making is commonly represented in programming languages using the if statement, sometimes combined with go to statements and labels. MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to. The first instruction is beq register1, register2, L1 This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal.

The second instruction is bne register1, register2, L1 It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. The mnemonic bne stands for branch if not equal. These two instructions are traditionally called **conditional branches**.



Loops

Decisions are important both for choosing between two alternatives—found in if statements—and for iterating a computation—found in loops.

Compiling a while Loop in C

```
while (save[i] == k)
i += 1;
```

Assume that i and k correspond to registers $s3 and $s5 and the base of the array is saved in $s6.

The first step is to load save[i] into a temporary register. Before we can load save[i] into a temporary register, we need to have its address. Before we can add i to the base of array save to form the address, we must multiply the index i by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4. We need to add the label Loop to it so that we can branch back to that instruction at the end of the loop:

Loop: sll $t1,$s3,2 # Temp reg $t1 = i * 4

To get the address of save[i], we need to add $t1 and the base of save in $s6:

add $t1,$t1,$s6 # $t1 = address of save[i]

Now we can use that address to load save[i] into a temporary register:

lw $t0,0($t1) # Temp reg $t0 = save[i] .

The next instruction performs the loop test, exiting if save[i] ≠ k:

bne $t0,$s5, Exit # go to Exit if save[i] ≠ k

The next instruction adds 1 to i:

addi $s3,$s3,1 # i = i + 1

The end of the loop branches back to the while test at the top of the loop. We just add the Exit label after it, and we're done:

```
j Loop                    # go to Loop
Exit:
```

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is larger than any that begins with a 0.

MIPS offers two versions of the set on less than comparison to handle these alternatives. Set on less than (slt) and set on less than immediate (slti) work with signed integers. Unsigned integers are compared using set on less than unsigned (sltu) and set on less than immediate unsigned (sltiu).

Case/Switch Statement

Most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements. Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence.

The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a jump register instruction (jr), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction.

PART C

1.What is an addressing mode? Explain the various addressing modes with suitable examples. (April/May 2015) (Nov/Dec 2015)(May/June 16). (16)

- **What is the need for addressing in a computer system? Explain the different addressing modes with suitable examples. (April/May 2015). (16)**
- **Assume two address format specified as source and destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction. (Nov/Dec 2014). (16)**

Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. The various addressing modes that are defined in a given instruction set architecture define how machine language instructions in that architecture identify the operand (or operands) of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere. The CPU can access data in various ways. The data could be in a register, or in memory, or be provided as an immediate value. These various ways of accessing data are called addressing modes.

Perform any operation; the corresponding instruction is to be given to the microprocessor. In each instruction, programmer has to specify 3 things:

- Operation to be performed.
- Address of source of data.

- Address of destination of result.

The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes. The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

- To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
- To reduce the number of bits in the addressing field of the instruction.

The MIPS addressing modes are the following:

1. Immediate addressing, where the operand is a constant within the instruction itself
2. Register addressing, where the operand is a register
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction
5. Pseudo direct addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

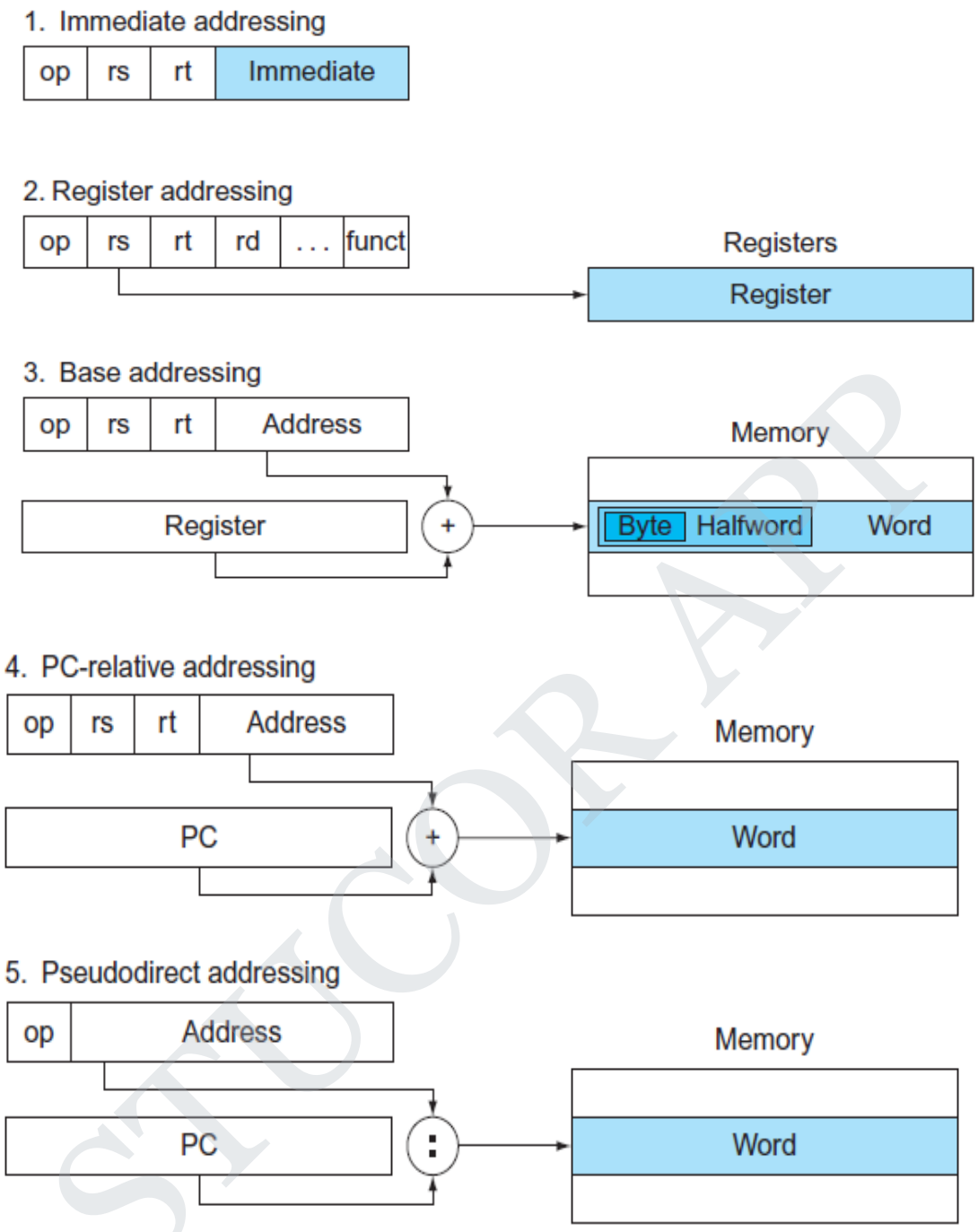


Fig: 11 Various Addressing Modes

Types of Addressing Modes

- Implied addressing mode
- Immediate addressing mode
- Direct addressing mode
- Indirect addressing mode

- Register addressing mode
- Register Indirect addressing mode
- Auto increment or Auto decrement addressing mode
- Relative addressing mode
- Indexed addressing mode
- Base register addressing mode

a. Implied Addressing Mode:

The operands are specified implicitly in the definition of the instruction. For example the 'complement accumulator' instruction is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction itself. All register reference instructions that use an accumulator are implied mode instructions. Zero address instructions in a stack organized computer are implied mode instructions since the operands are implied to be on the top of the stack.

b. Immediate Addressing Mode:

The operand is specified in the instruction itself. In other words, an immediate mode instruction has a operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate mode instructions are useful for initializing registers to a constant value.

Example: ADD 5 # Add 5 to contents accumulator

c. Direct Addressing Mode:

The effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of instruction. In a branch type instruction the address field specifies the actual branch

address.

E.g. LDA A # Look in memory at address A for operand. Load contents of A to accumulator

d. Indirect Addressing Mode:

The address field of the instruction gives the address where the effective address is stored in memory/register. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

EA = address contained in register/memory location

Example **Add (M)**

- Look in M, find address contained in M and look there for operand
- Add contents of memory location pointed to by contents of M to accumulator

e. Register Addressing Mode:

The instruction specifies a register in the CPU whose contents give the effective address of the operand in the memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Therefore EA = the address stored in the register R

- Operand is in memory cell pointed to by contents of register R
- Example **Add (R2), R0**

f. Register Indirect Addressing Mode:

The instruction specifies a register in the CPU whose contents give the effective address of the operand in the memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Therefore EA = the address stored in the register R

- Operand is in memory cell pointed to by contents of register R

g. Auto Increment or Auto Decrement Addressing Mode:

Auto Increment Mode:

The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

The Auto increment mode is denoted by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Auto increment mode is written as **(Ri) +**

Auto Decrement Mode:

The contents of a register specified in the instruction are first automatically

decremented and is then used as the effective address of the operand. We denote the Auto decrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write **- (Ri)**

- These two modes are useful when we want to access a table of data.

ADD (R1)+	will increment the register R1.
LDA -(R1)	will decrement the register R1.

h. Relative Addressing Mode:

The content of the program counter is added to the address part of the instruction in order to obtain the effective address. Effective address is defined as the memory address obtained from the computation dictated by the given addressing mode. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

Relative addressing is often used with branch type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the bits required to designate the entire memory address.

EA = A + contents of PC

Example: PC contains 825 and address pa rt of instruction contains 24.

After the instruction is read from location 825, the PC is incremented to 826. So EA=826+24=850. The operand will be found at location 850 i.e. 24 memory locations forward from the address of the next instruction.

i. Indexed Addressing Mode:

The content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index type instruction does not include an address field in its format, then the instruction converts to the register indirect mode of operation.

Therefore EA = A + IR Example MOV AL , DS: disp [SI]

j. Base register Addressing Mode:

The content of base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed.

An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of the programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change.

2.Assume a two address format specified as source, destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction. (April/May 2014) (10)

(1) Move (R_5)+, R_0

(2) Add (R_5) +, R_0

(3) Move R_0 , (R_5)

(4) Move 16(R_5), R_3

(5) Add #40,R_5.

SOLUTION

Assume the register R_5

1. Move (R_5) + R_0

The auto increment addressing mode is used.

regs[R_0] ⟵ Mem(Regs[R_5])

regs[R_5] ⟵ Regs[R_5]+d

R_5 points to the start of the array, each reference increments R_5 by size of an element d. The instruction copies the contents of register R_0 into the memory location whose address is specified by the contents of register R_2. After the contents of register R_2 are automatically incremented by size of an element d.

2) Add (R_5) + ,R_0 – Autoincrement mode

The above instruction adds the contents of register R_0 with the memory location specified by the contents of register R_2 , after addition, the contents of register R_5 are automatically incremented by 1.

Add(R_5) + , R_0 ⇨ Regs[R_0] ⟵ Regs[R_0]+

Mem(Regs[R_5]) Regs[R_5] ⟵ Regs[R_5] + d

3) Move R_0 , (R_5) – Register Indirect Mode

Move R_0 , (R_5) ⇨ Mem(Regs(R_5)) ⟵ Regs[R_0]

The instruction copies the contents of register R_0 into the memory location whose address is stored at the register R_5.

4) Move 16(R_5), R_3 – Displacement mode.

Move 16(R_5), R_3 ⇨ Regs[R_3] ⟵ Regs[R_3] + Mem[16 + (Regs[R_5]]

The above instruction loads the contents of memory location whose address is calculated by addition of the contents of register R_5 and constant 16 into the register R_3.

5) Add #40,R_5 – Immediate mode

The above instruction copies operand 40 into the register R_5

Add #40,R_5 ⇨ Regs[R_5] ⟵ Regs[R_5] +40

UNIT II

UNIT II ARITHMETIC OPERATIONS 7

ALU - Addition and subtraction – Multiplication – Division – Floating Point operations – Subword parallelism.

PART A

1. How overflow occur in subtraction. (April/May 2015)

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. It means a borrow occurred from the sign bit.

2. What is meant by Little endian and Big endian? (Nov/Dec 14)

Big endian systems in which the most significant byte of the word is stored in the smallest address given and the least significant byte is stored in largest.

Little endian systems are those in which the least significant byte is stored in the smallest address.

3. Define – Guard and Round. (May/June 2014)

Guard is the first of two extra bits kept on the right during intermediate calculations of floating point numbers. It is used to improve rounding accuracy.

Round is a method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format. IEEE 754, therefore, always keeps two extra bits on the right during intermediate additions, called guard and round, respectively.

4. Let X=1010100 & Y=1000011 perform X-Y Using 2's Complement, Y-X

(1) 2's comp of Y is 0111101

X-Y = 10010001

(2) 2's comp of X is 0101100

Y-X = 1101111

5. What is carry look ahead adder?

A carry-look ahead adder (CLA) or fast adder is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.

6. What are the techniques to speed up the multiplication operation?

There are two techniques for speeding up the multiplication operation. The First technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is n/2 for n-bit operands. The second technique reduces the time needed to add the summands (carry-save addition of summands method).

7. List out the rules for add/sub of floating point number?

- Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
- Set the exponent of the result equal to the larger exponent.
- Perform addition /subtraction on the mantissa and determine the sign of the result.
- Normalize the resulting value, if necessary.

8. What do you meant by sub word parallelism? (April/May 2015)

By partitioning the carry chains within a 128 bit adder processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8 bit operands, eight 16 bit operands, four 32 bit operands, or two 64 bit operands. The cost of such partitioned adders was small. This concept is called as subword parallelism.

9. State the representation of double precision floating point number. [Nov/Dec 2015]

The double precision floating point representation is

$$(-1)^1 \text{ X } (1 + \text{Fraction}) \text{ X } 2^{(\text{Exponent} - 1023)}$$

Where Fraction is a 52 bit.

10. What are the functions of ALU?(May /June 2016)

An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

11. **When can overflow occur in addition? (Nov/Dec 2015)**

When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10 + 4 = - 6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

PART B

1. ALU

❖ **Explain the various steps in designing a 32 bit ALU.**

An arithmetic logic unit (ALU) represents the fundamental building block of the central processing unit of a computer. An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs

contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

The basic building block of all arithmetic & logic operation (ALU) is a parallel adder. The basic functions such as AND, OR, NOT & X-OR. Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data and the ALU stores the result in an output register.

The control unit moves the data between these registers, the ALU, and memory. All information in a computer is stored and manipulated in the form of binary numbers, i.e. 0 and 1.Transistor switches are used to manipulate binary numbers since there are only two possible states of a switch: open or closed. An open transistor, through which there is no current, represents a 0.

A closed transistor, through which there is a current, represents a 1.Operations can be accomplished by connecting multiple transistors. One transistor can be used to control a second one in effect, turning the transistor switch on or off depending on the state of the second transistor. This is referred to as a gate because the arrangement can be used to allow or stop a current.

The simplest type of operation is a NOT gate. This uses only a single transistor. It uses a single input and produces a single output, which is always the opposite of the input. This figure shows the logic of the NOT gate

Design of Arithmetic Unit

The basic component of the arithmetic unit is parallel adder. A parallel adder can be constructed using number of full-adder circuits connected in cascade.

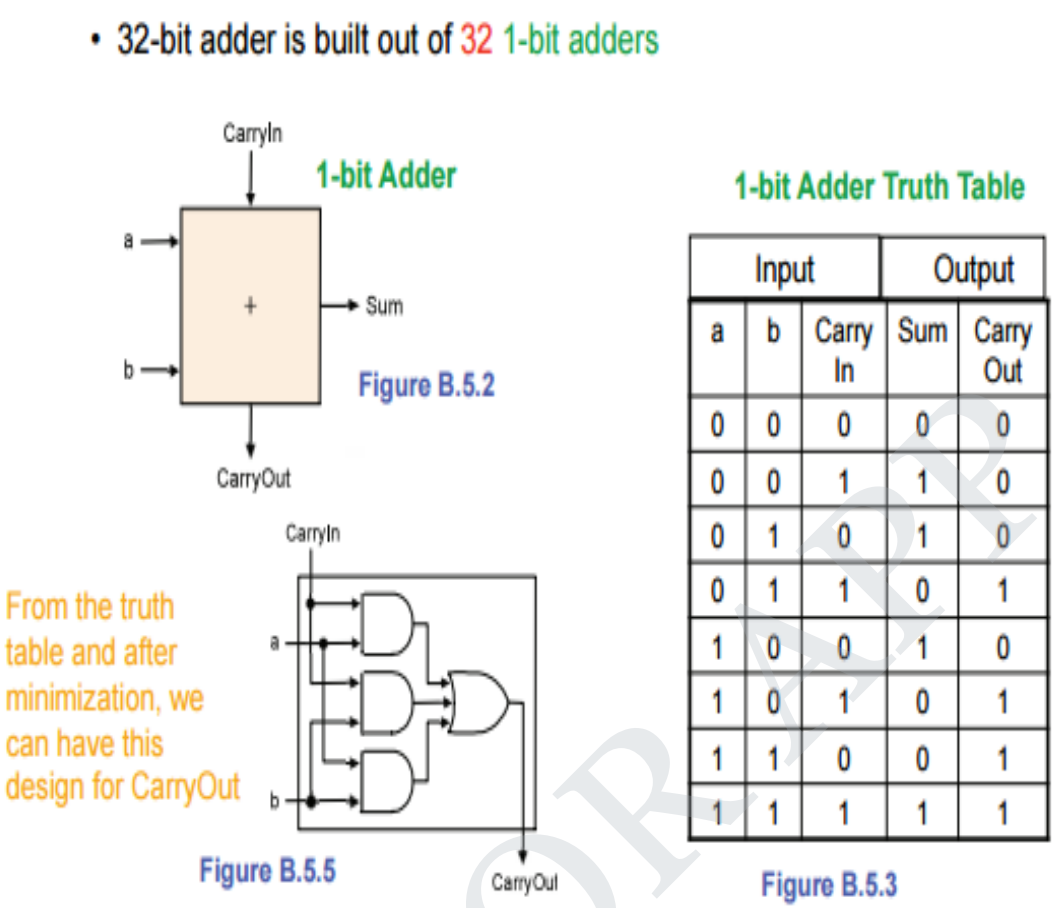


Figure 2.1 32 bit Adder

Design of Logic Unit It is simpler than arithmetic circuit design.

S1	S2	Output	Function
0	0	Y = A OR B	OR
0	1	Y = A XOR B	XOR
1	0	Y = A AND B	AND
1	1	Y = NOT A	NOT

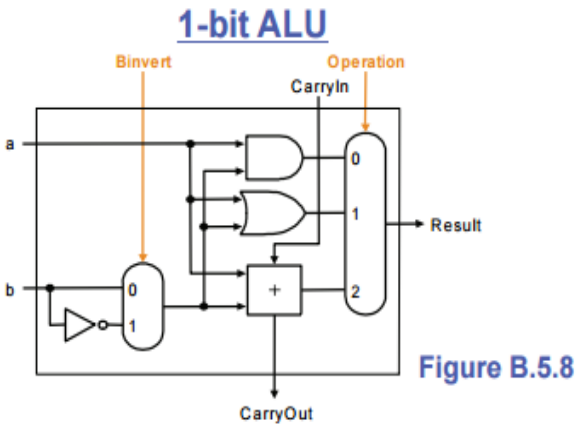


Figure B.5.8

Function	Control lines: Binvert (1 line)	Control lines: Operation (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10

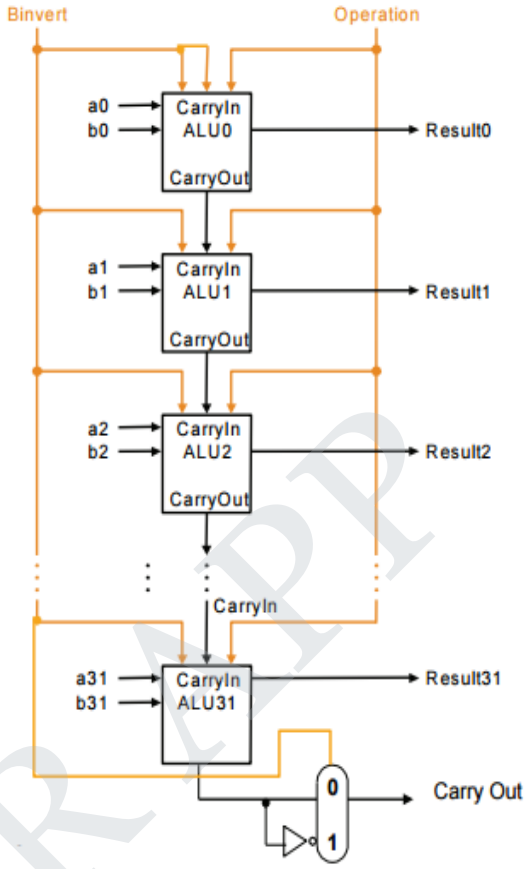
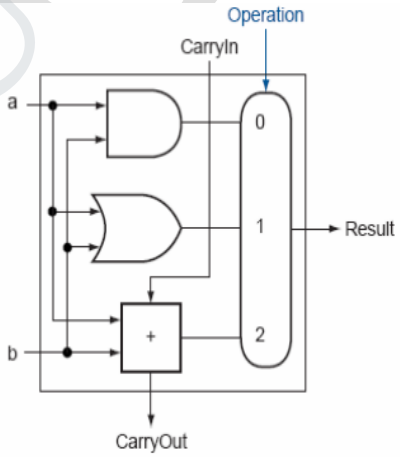


Figure 2.2 1 Bit ALU

Design of a full adder



$$\text{Sum}=(a\cdot\bar{b}\cdot\overline{\text{CarryIn}})+(\bar{a}\cdot b\cdot\overline{\text{CarryIn}})+(\bar{a}\cdot\bar{b}\cdot\text{CarryIn})+(a\cdot b\cdot\text{CarryIn})$$

Figyre 2.3 Design of a Full adder

2. ADDITION AND SUBTRACTION

- **Explain in detail about Addition and Subtraction operations.**
- **Briefly Explain carry look ahead adder.(Nov/Dec 2014).**

Digits are added bit by bit from right to left with carries passed to the next digit to the left , just as we would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6ten from 7ten in binary.

$$\begin{aligned} &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ +\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ =\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten} \end{aligned}$$

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{aligned} &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ -\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ =\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{aligned}$$

or via addition using the two's complement representation of - 6:

$$\begin{aligned} &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ +\ &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ =\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{aligned}$$

Overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands.

For example, -10 + 4 = -6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the same, overflow cannot occur.

To see this, remember that c – a = c + (-a) because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by adding operands of different signs.

Adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs, the sign bit is set with the value of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a borrow occurred from the sign bit.

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

The computer designer must therefore provide a way to ignore overflow in

some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
- Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do *not* cause exceptions on overflow.

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

MIPS include a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception. The instruction move from system control (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

CARRY-LOOK AHEAD ADDER

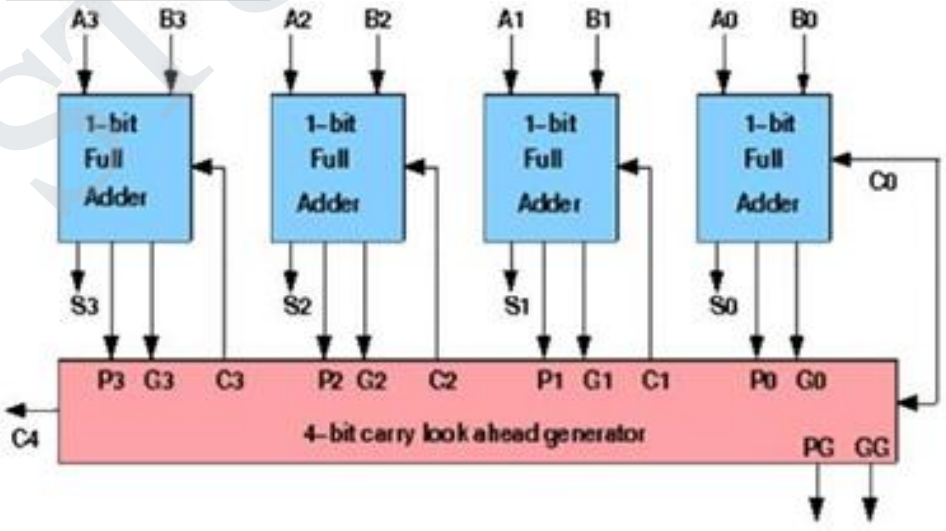


Figure 2.3 Carry-look ahead adder

- Binary addition would seem to be dramatically slower for large registers consider 0111 + 0011, carries propagate left-to-right. So 64-bit addition would be 8 times slower than 8- bit addition
- It is possible to build a circuit called a "carry look-ahead adder" that speeds up addition by eliminating the need to "ripple" carries through the word.
- Carry look-ahead is expensive
- If n is the number of bits in a ripple adder, the circuit complexity (number of gates) is O(n)
- For full carry look-ahead, the complexity is O(n3)
- Complexity can be reduced by rippling smaller look-aheads: e.g., each 16 bit group is handled by four 4-bit adders and the 16-bit adders are rippled into a 64-bit adder.

The advantage of the CLA scheme used in this circuit is its simplicity, because each CLA block calculates the generate and propagate signals for two bits only. This is much easier to understand than the more complex variants presented in other textbooks, where combinatorial logic is used to calculate the G and P signals of four or more bits, and the resulting adder structure is slightly faster but also less regular.

3a.MULTIPLICATION

- **Explain in detail about the multiplication algorithm with suitable example and Diagram(Nov/Dec 2015)(May/June 2016) 16 marks**
- **Explain the sequential version of multiplication algorithm and its hardware. (April/May 2015)**

The first operand is called the **multiplicand** and the second the **multiplier**. The final result is called the product. Take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier and shifting the intermediate product one digit to the left of the earlier intermediate products. The first observation is that the number of digits in the product is

considerably larger than the number in either the multiplicand or the multiplier. If we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long.

Multiplying 1000_{ten} by 1001_{ten}:

Multiplicand	1000
Multiplier X	1001
	———
	1000
	0000
	0000
	1000
Product	1001000

That is, n + m bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In the above example we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

- Just place a copy of the multiplicand in the proper place if the multiplier digit is 1
- Place 0 (0 X multiplicand) in the proper place if the digit is 0.

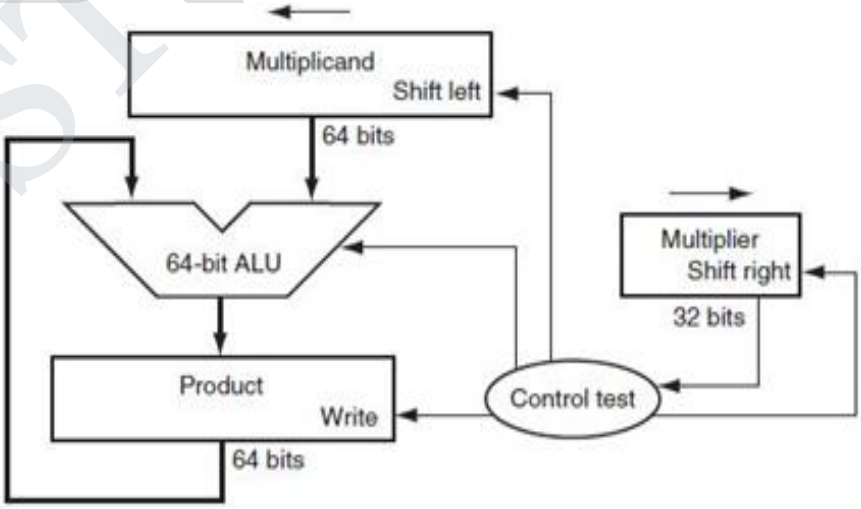


Figure 2.4 First version of the multiplication hardware

Multiplicand, ALU and product register are all 64 bit and multiplier is 32 bit.

Multiplicand starts in the right half of register and it is shifted left 1 bit on each step.

Multiplier is shifted in opposite direction at each step.

Algorithm starts with product equal to 0. Control decides when to shift the multiplicand and multiplier register and when to write new values into the product register.

MULTIPLICATION ALGORITHM

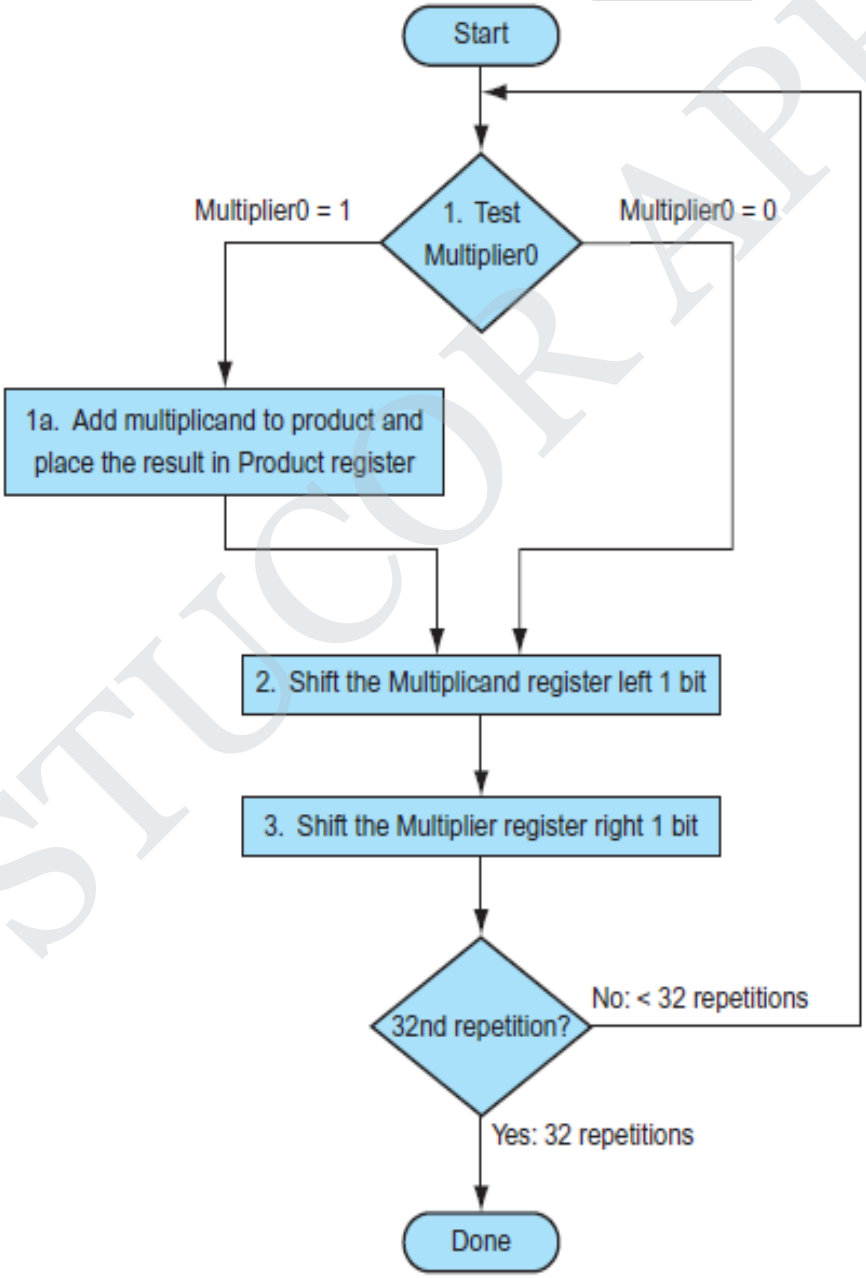


Figure 2.5 The First Multiplication Algorithm

Step 1: Test the LSB of the multiplier.

If LSB = 1 add multiplicand to the product register

Step 2 : Shift the Multiplicand register left 1 bit

Step 3: Shift the Multiplier register 1 bit.

Step 4: Repeat 32 times to obtain the product.

This algorithm requires at most 100 clock cycles to multiply two 32 bit numbers. The speed of the process is increased by performing the operation in parallel. That means the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1.

REVISED MULTIPLICATION HARDWARE

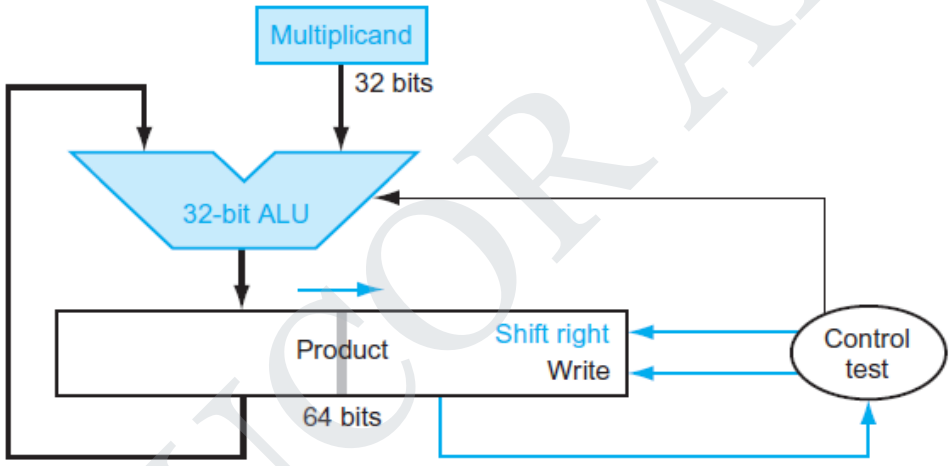


Figure 2.6 Revised Multiplication Hardware

The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register.

MULTIPLICATION ALGORITHM

Using 4-bit numbers to save space, multiply 2_{ten} X 3_{ten}, or 0010_{two} X 0011_{two}. with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001**1**	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	**0000 0010**
	2: Shift left Multiplicand	0011	**0000 0100**	0000 0010
	3: Shift right Multiplier	**0001**	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	**0000 0110**
	2: Shift left Multiplicand	0001	**0000 1000**	0000 0110
	3: Shift right Multiplier	**0000**	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	**0001 0000**	0000 0110
	3: Shift right Multiplier	**0000**	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	**0010 0000**	0000 0110
	3: Shift right Multiplier	**0000**	0010 0000	0000 0110

SIGNED MULTIPLICATION:

In the signed multiplication, convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. The shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

FASTER MULTIPLICATION

Moore's Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits.

Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand AND ed with a multiplier bit, and the other is the output of a prior adder. Connect the outputs of adders on the right

to the inputs of adders on the left, making a stack of adders 32 high.

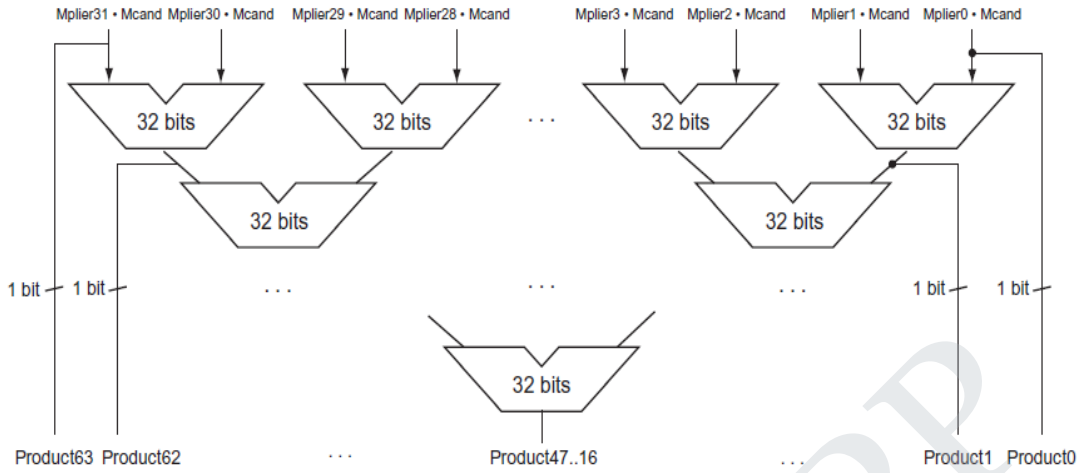


Figure 2.7 Faster multiplier

The figure 2.7 shows an alternative way to organize 32 additions in a parallel tree. Instead of waiting for 32 add times, we wait just the $\log_2$ (32) or five 32-bit add times. Multiply can go even faster than five add times because of the use of carry save adders. It is easy to pipeline such a design to be able to support many multiplies simultaneously

Multiply in MIPS:

MIPS provide a separate pair of 32-bit registers to contain the 64-bit product, called Hi and Lo. To produce a properly signed or unsigned product, MIPS have two instructions: multiply (mult) and multiply unsigned (multu). To fetch the integer 32-bit product, the programmer uses move from lo (mflo). The MIPS assembler generates a pseudo instruction for multiply that specifies three general purpose registers, generating mflo and mfhi instructions to place the product into registers.

3b. BOOTH ALGORITHM (Nov/Dec 2014).

• Three n bit registers, one 1 bit register logically to the right of Q (denoted as Q_{-1})

• Register set up

- Q register = multiplier

- $Q_{-1} = 0$
- M register = multiplicand
- A register = 0
- Count = n

• Product will be 2n bits in AQ registers

• Bits of the multiplier are scanned one at a time (the current bit Q_0)

• As bit is examined the bit to the right is considered also (the previous bit Q_{-1})

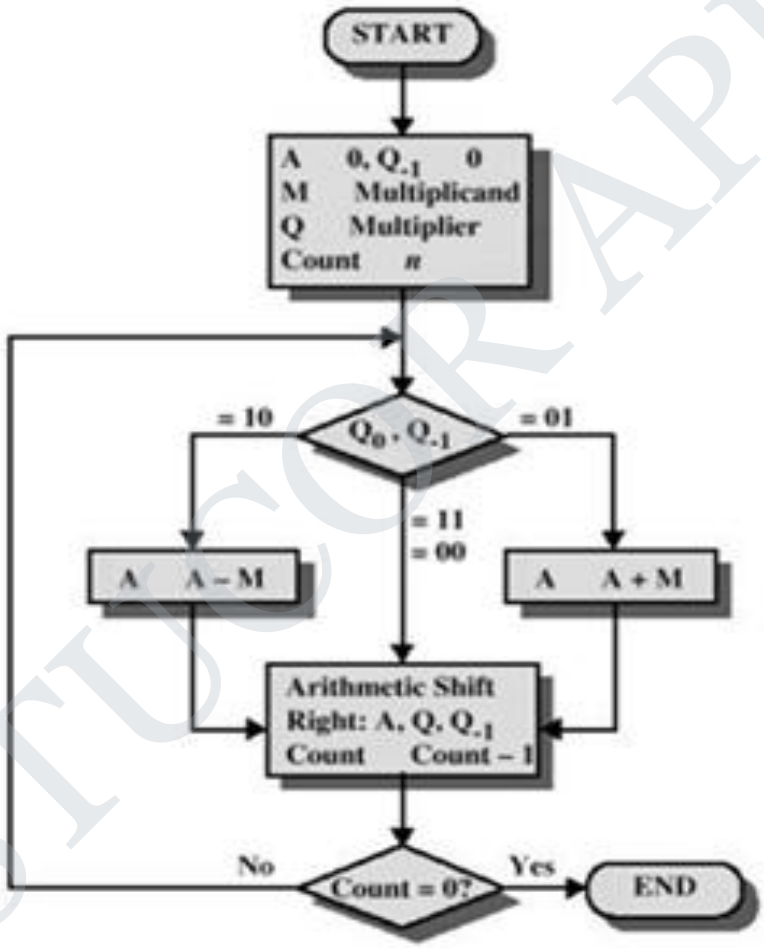


Figure 2.8 Booth algorithm

Then if Q_0 Q_{-1}

00: No arithmetic operation.

01: Add the multiplicand to the left half of the product (A).

10: Subtract the multiplicand from the left half of the product (A).

11: No arithmetic operation.

- Then shift A, Q, bit Q_{-1} right one bit using an arithmetic shift

In an arithmetic shift, the MSB remains unchanged.

Example of Booth's Algorithm (7*3=21)

A	Q	Q-1	M			
0000	0011	0	0111	Initial Values		
1001	0011	0	0111	A	A-M	First Cycle
1100	1001	1	0111	Shift		First Cycle
1110	0100	1	0111	Shift		Second Cycle
0101	0100	1	0111	A	A-M	Third Cycle
0010	1010	0	0111	Shift		Third Cycle
0001	0101	0	0111	Shift		Fourth Cycle

4a.DIVISION

- **Discuss in detail about division algorithm in detail with diagram and examples. (Nov/Dec 2015). (16)**
- **Explain the concepts of Division Algorithm and its hardware./Divide $(12)_{10}$ by $(3)_{10}$ using Restoring and Non Restoring division algorithm with step by step intermediate results and explain. (Nov/Dec 2014). (16)**

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0. The example is dividing 1,001,010 by 1000. The two operands (dividend and divisor) and the result (quotient) of divide are accompanied by a second result called the remainder. Here is another way to

express the relationship between the components:

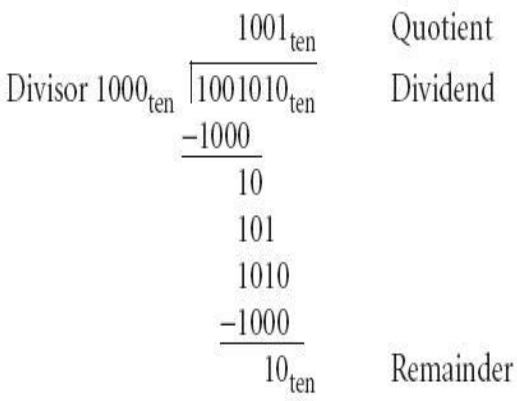


Figure 2.9 Basic Method

Dividend = Quotient * Divisor + Remainder where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient. The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division. If both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values.

DIVISION HARDWARE

The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

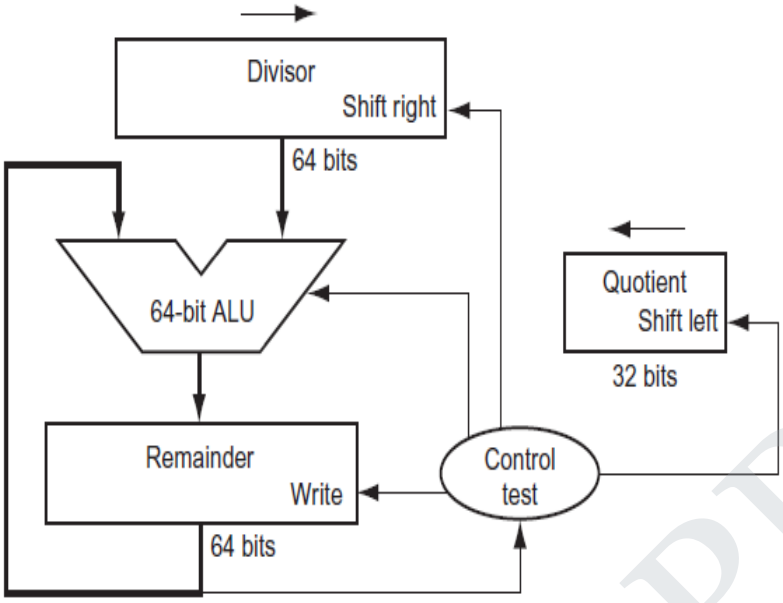


Figure 2.10 First version of the division hardware

DIVISION ALGORITHM

Initially, the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

The figure 2.11 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; If the result is positive, the divisor was smaller or equal to the dividend, so generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

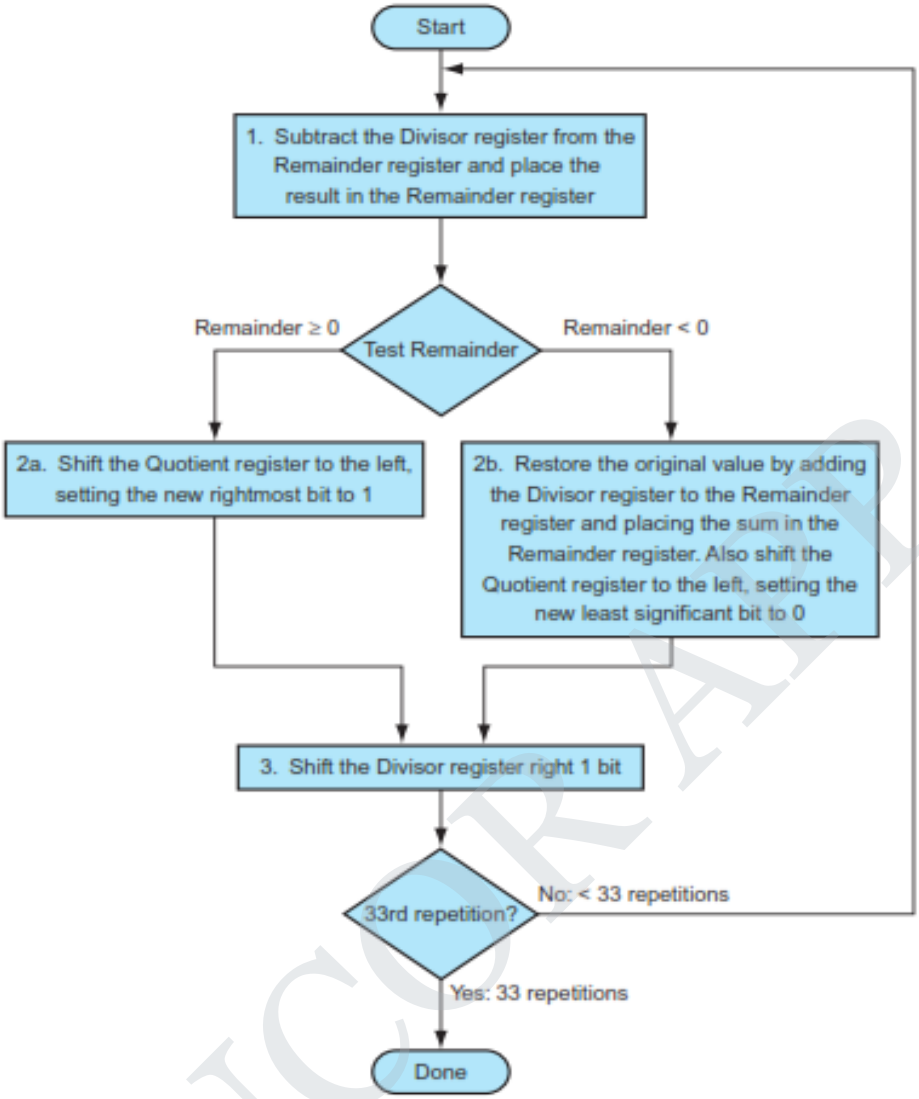


Figure 2.11 The first division algorithm

For Example

Using a 4-bit version of the algorithm to save pages, let's try dividing 7ten by 2ten, or 0000 0111two by 0010two.Figure shows the value of each register for each of the steps, with the quotient being 3ten and the remainder 1ten. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	**1110 0111**
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0010 0000	**0000 0111**
	3: Shift Div right	0000	**0001 0000**	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	**1111 0111**
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0001 0000	**0000 0111**
	3: Shift Div right	0000	**0000 1000**	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	**1111 1111**
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0000 1000	**0000 0111**
	3: Shift Div right	0000	**0000 0100**	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	**0000 0011**
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	**0001**	0000 0100	0000 0011
	3: Shift Div right	0001	**0000 0010**	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	**0000 0001**
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	**0011**	0000 0010	0000 0001
	3: Shift Div right	0011	**0000 0001**	0000 0001

Figure 2.12 Values of register in division algorithm

An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. The ALU and Divisor registers are halved and the remainder is shifted left . This version also combines the Quotient register with the right half of the Remainder register.

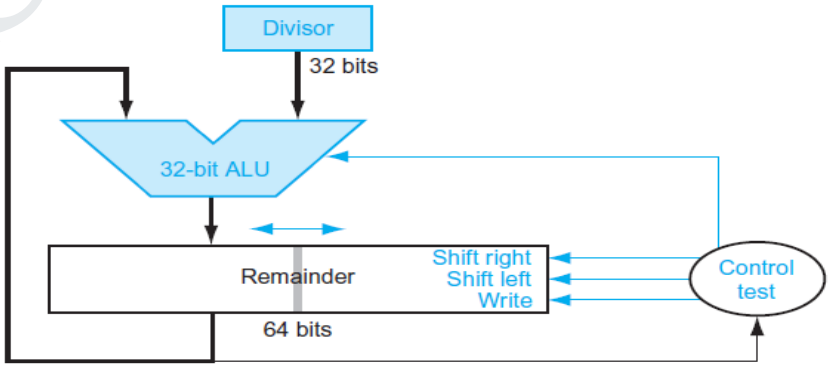


Figure 2.13 Revised Division Hardware

SIGNED DIVISION:

The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

Dividend = Quotient × Divisor + Remainder

To understand how to set the sign of the remainder,
For example dividing all the combinations of $\pm 7_{10}$ by $\pm 2_{10}$.

Case:

+7 ÷ +2 : Quotient = +3, Remainder = +1

Checking the results:7 = 3 × 2 + (+1) = 6 + 1

If we change the sign of the dividend, the quotient must change as well:
–7 ÷ +2: Quotient = –3

Rewriting the formula to calculate the remainder:

Remainder = (Dividend – Quotient × Divisor)

= –7 – (–3 × +2)
= –7–(–6)
= –1

So, –7 ÷ +2 : Quotient = –3, Remainder = –1

Checking the results again: –7 = –3 × 2 + (–1) = – 6 – 1

The reason the answer is not a quotient of –4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor. Clearly, if programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient. The other combinations by following the same rule is calculated as follows

$-(x \div y) \neq (-x) \div y$

$+7 \div -2$: Quotient = –3, Remainder = +1

$-7 \div -2$: Quotient = +3, Remainder = –1

Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division:

Many adders can be used to speed up multiply, cannot be used to do the same trick for divide. The reason is that it is needed to know the sign of the difference before performing the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per step. The SRT division technique tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong guesses.

4b. RESTORING DIVISION ALGORITHM

Division of 4-bit number by 7-bit dividend:

- Load 0 into accumulator k-bit A and dividend X is loaded into the k-bit quotient register Q.
- Shift A and Q left by one bit
- Subtract the divisor from A and place the answer back in A (A=A-B)
- Check MSB of A. If 1 : Set Q_0 =0 and restore A, Otherwise Set Q_0 to 1
- Repeat till the total number of cyclic operations = k. At the end, A has the remainder and MQ has the quotient.

	A Register					Q Register			
Initially	0	0	0	0	0	1	0	1	0
Shift Left	0	0	0	0	1	0	1	0	
A= A-B	1	1	1	0	1				

Check MSB of A. If 1 : Set Q_0 =0 and restore A, Otherwise Set Q_0 to 1

Set Q_0	1	1	1	1	0				
Restore	0	0	0	1	1				
	0	0	0	0	1	0	1	0	0
Shift	0	0	0	1	0	1	0	0	
Subtract B	1	1	1	0	1				
Set Q_0	1	1	1	1	1				
Restore	0	0	0	1	1				
	0	0	0	1	0	1	0	0	0
Shift	0	0	1	0	1	0	0	0	
Subtract B	1	1	1	0	1				
Set Q_0	0	0	0	1	0				
						0	0	0	1
Shift	0	0	1	0	0	0	0	1	
Subtract B	1	1	1	0	1				
Set Q_0	0	0	0	0	1				
	Remainder					0	0	1	1
						Quotient			

PART C

. 1.. DIVISION USING NON-RESTORING ALGORITHM

- If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise shift A and Q left and add divisor to A
- Repeat Steps 1 to n times.
- If the sign of A is 1 add divisor to A
- For Example : 10/3

	A Register					Q Register			
Initially	0	0	0	0	0	1	0	1	0
Shift Left	0	0	0	0	1	0	1	0	
A= A-B	1	1	1	0	1				
Set Q_0	1	1	1	1	0	0	1	0	0
Shift	1	1	1	0	0	1	0	0	
Add	0	0	0	1	1				
Set Q_0	1	1	1	1	1	1	0	0	0
Shift	1	1	1	1	1	0	0	0	
Add	0	0	0	1	1				
Set Q_0	0	0	0	1	0	0	0	0	1
Shift	0	0	1	0	0	0	0	1	
Subtract	1	1	1	0	1				
Set Q_0	0	0	0	0	1				
	Remainder					0	0	1	1
						Quotient			

Figure 2.14 Division using Non-restoring Algorithm

2. Explain how floating point addition is carried out in a computer system. Give an example for a binary floating addition. /Explain in detail about Floating Point Operations. (April/May 2015)(May/June 16). (16)

The scientific notation has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a normalized number, which is the usual way to write it. Floating point - Computer arithmetic that represents numbers in which the binary point is not fixed. Floating-point numbers are usually a multiple of the size of a word.

The representation of a MIPS floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number. This representation is called sign and magnitude, since the sign has a separate bit from the rest of the number.

A standard scientific notation for reals in normalized form offers three advantages.

- It simplifies exchange of data that includes floating-point numbers;
- It simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form;
- It increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

31	30 to 23	22 to 0
s	exponent	fraction
1 bit	8 bits	23 bits

Figure2.15 Scientific Notation

5a.FLOATING POINT ADDITION

Example: Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$. Assume that we can store only four decimal digits of the significant and two decimal digits of the exponent.

Step 1.

To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus, the first step shifts the significant of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really $0.016_{ten} \times 10^{1}$

Step 2. Next comes the addition of the significant:

$$9.999_{ten} + 0.016_{ten} = 10.015_{ten} \qquad \text{so the sum is } 10.015_{ten} \times 10^{1}.$$

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significant can be only four digits long (excluding

the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number $1.0015_{ten} \times 10^2$ is rounded to four digits in the significant to $1.002_{ten} \times 10^2$ since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

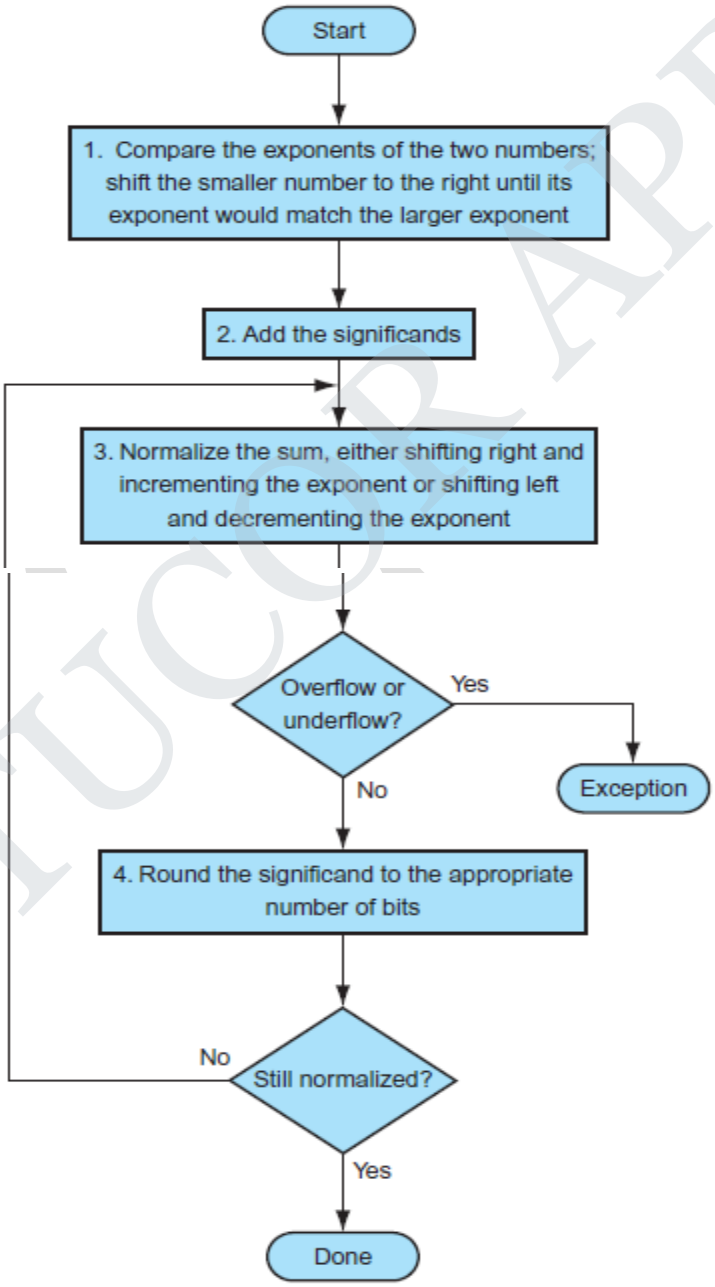


Figure 2.16 Floating Point Addition

The algorithm for binary floating-point addition that follows this decimal

example. Adjust the significant of the number with the smaller exponent and then add the two significant. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers. Thus, for single precision, the maximum exponent is 127, and the minimum exponent is -126. The limits for double precision are 1023 and -1022.

2..FLOATING-POINT MULTIPLICATION

First start by multiplying decimal numbers in scientific notation by hand:

$$1.110_{ten} \text{ X } 10^{10} \text{ X } 9.200_{ten} \text{ X } 10^{-5}.$$

Assume that we can store only four digits of the significand and two digits of the exponent.

Step1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: 10 + 127 = 137, and -5 + 127 = 122, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 = +2 \text{ X } 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

New exponent = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127) and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$
\begin{array}{l}
1.110_{ten} \\
\times\ 9.200_{ten} \\
\hline
0000 \\
\ \ 0000 \\
2220 \\
9990 \\
\hline
10212000_{ten}
\end{array}
$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{ten}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is $10.212 \text{ X } 10^5$.

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \text{ X } 10^5 = 1.0212_{ten} \text{ X } 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number $1.0212_{ten} \text{ X } 10^6$ is rounded to four digits in the significand to $1.021ten \text{ X } 10^6$

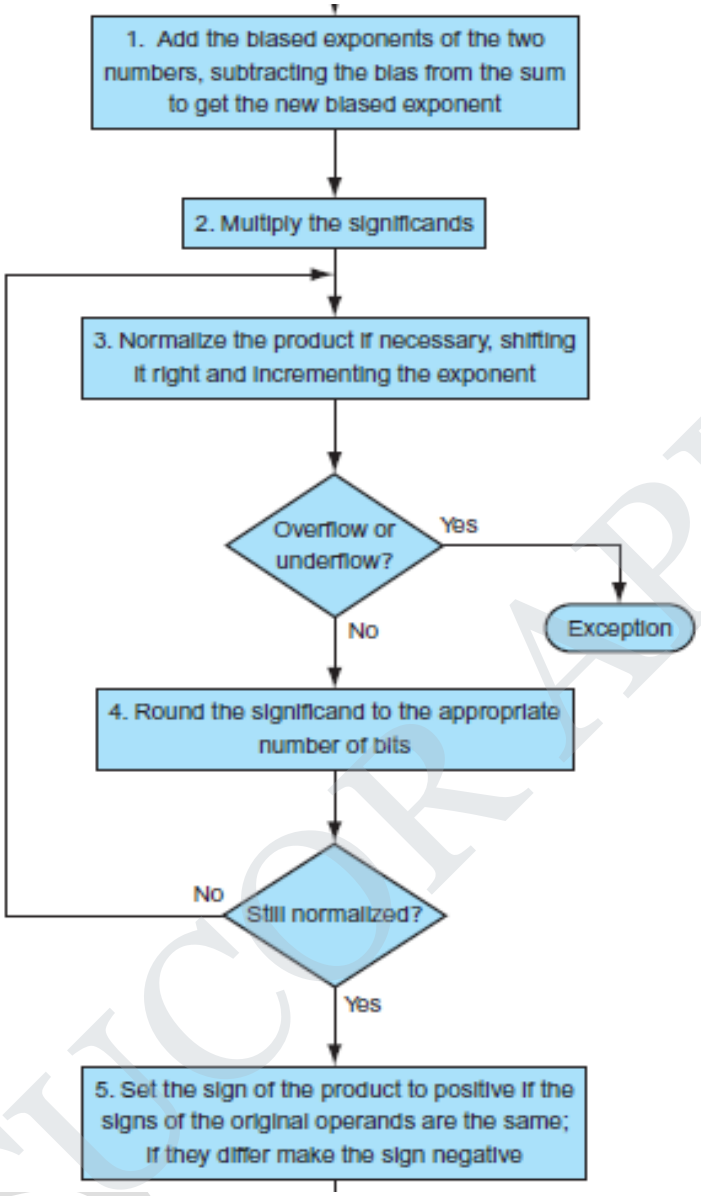


Figure 2.17 Floating Point Multiplication

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is $+1.021_{ten}$ X 10^6 The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication, the sign of the product is determined by the signs of the operands

Multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using

In binary, the task is multiplying 1.000_{two} X 2^{-1} by -1.110_{two} X 2^{-2}

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times\ 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is 1.110000_{two} X 2^{-3}, we need to keep it to 4 bits, so it is 1.110_{two} X 2^{-3}.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow.

Step 4. Rounding the product makes no change:

$$1.110_{two} \text{ X } 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{two} \text{ X } 2^{-3}$$

Converting to decimal to check our results:

$$-1.110_{two} \text{ X } 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^{5}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

SUB WORD PARALLELISM

3.Write short notes on SubWord Parallelism.

A subword is a lower precision unit of data contained within a word. In subword parallelism, we pack multiple subwords into a word and then process whole words. With the appropriate subword boundaries, this technique results in parallel processing of subwords. Since the same instruction applies to all the subwords within the word, this is a form of small-scale SIMD (single-instruction. multiple-data) processing. However, implementations are much simpler if we allow only a few subword sizes and if a single instruction operates on contiguous subwords that arc all the same size.

Data-parallel programs that benefit from subword parallelism tend to process data that are of the same size. For example, if the word size is 64 bits, some useful subword sizes are 8, 16, and 32 bits. Hence, an instruction operates on eight g-bit subwords, four l & bit subwords, two 32-bit subwords, or one 64-bit subword (a word) in parallel. The degree of SIX/ID parallelism within an instruction, then, depends upon the size of the subwords. Data parallelism refers to an algorithm‘s execution of the same program module on different sets of dacd.

Subword parallelism is an efficient and flexible solution for media processing, because the algorithms exhibit a great deal of data parallelism on lower precision data. The basic components of multimedia objects are usually simple integers with 8, 12, or 16 bits of precision. Subword parallelism is also useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data. One key advantage of subword parallelism is that it allows general-purpose processors to exploit wider word sizes even when not processing high-precision data. The processor can achieve more subword parallelism on lower precision data rather than wasting much of the word-oriented data paths and registers. Media processors specially designed for media rather than general-purpose processing and DSPs allow more flexibility in data path widths. Even for these processors, however. organizing the data paths for words that

support subword parallelism provides low overhead parallelism with less duplication of control.

To exploit data parallelism, we need subword parallel compute primitives, which perform the same operation simultaneously on subwords packed into a word. These may include basic arithmetic operations like add, subtract, and multiply, as well as some form of divide, logical, and other compute operations.

Data-parallel computations also need 0 data alignment before or after certain operations for subwords representing fixed-point numbers or fractions; 0 subword rearrangement within a register so that algorithms can continue parallel processing at full clip; 0 a way to expand data into larger containers for more precision in intermediate computations. It adds two source operands then performs a divide by two. This is a combined operation instruction, involving an add and a right shift of one bit. In the process, the instruction shifts in the carry-out bit as the most significant bit of the result, so the instruction has the added advantage that no overflow can occur. In addition, it rounds the least significant bit to conserve precision in cascaded average operations. Figure.2.18 Permute allows rearrangement and repetition (a) and reversal (b) of subwords.

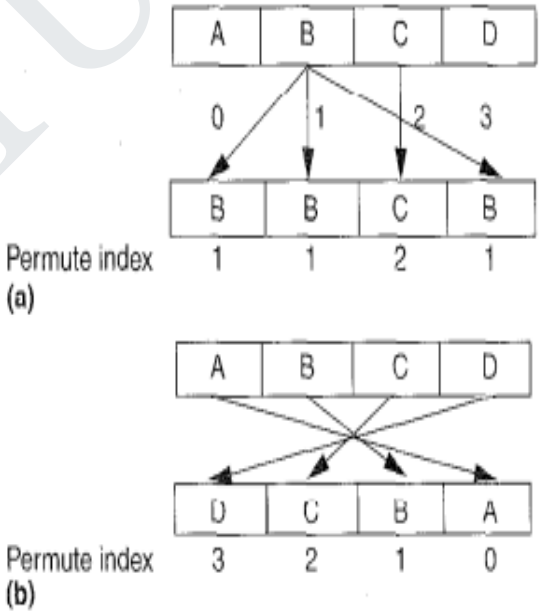


Figure 2.18 Rearrangement and repetition (a) and reversal (b) of subwords.

UNIT III

UNIT III PROCESSOR AND CONTROL UNIT 11

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

PART A

1. What are R type instructions? (April/May 2015)

For R-type instructions, an additional 6 bits are used (B5-0) called the function. Thus, the 6 bits of the opcode and the 6 bits of the function specify the kind of instruction for R-type instructions. This is the destination register.

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

op : Basic operation of the instruction called opcode

rs- The first register source operand

rt- The second register source operand

rd- The register destination operand .It gets the result of the operation.

shamt – shift amount

funct- Function. This field called function code, selects the specific variant of the operation in the op field.

2. What is branch prediction buffer and branch prediction? (April/May, Nov/Dec 2015)

A **branch prediction buffer** or branch history table is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

Branch prediction is a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting

to ascertain the actual outcome.

3. What is meant by speculation? (Nov/Dec 2014)

One of the most important methods for finding and exploiting more ILP is speculation. It is an approach whereby the compiler or processor guesses the outcome of an instruction to remove it as dependence in executing other instructions.
For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier.

4.What are exceptions and interrupts? (Nov/Dec 2014) (May/June 2016)

Exception, also called interrupt, is an unscheduled event that disrupts program execution used to detect overflow. Eg. Arithmetic overflow, using an undefined instruction. Interrupt is an exception that comes from outside of the processor. Eg. I/O device request.

It is also used to detect an overflow condition. Events other than branches or jumps that change the normal flow of instruction execution come under exception.

An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor.

5.Write control signals for storing a word in memory. (May/June 2014)

- R1out
- MARin R2out
- MDRin
- write MDRout E
- WMFC

6.What is meant by data hazards and control hazards in pipelining? (May/June 2012), (Nov/Dec 2013, Nov/Dec 2015)

DATA HAZARDS:

This is when reads and writes of data occur in a different order in the pipeline

than in the program code. There are three different types of data hazard named according to the order of operations that must be maintained

- **RAW:** A Read after Write hazard
- **WAR:** A Write After Read hazard is the reverse of a RAW
- **WAW:**A Write After Write hazard

CONTROL HAZARDS:

This is when a decision needs to be made, but the information needed to make the decision is not available yet. A Control Hazard is actually the same thing as a RAW data hazard (see above), but is considered separately because different techniques can be employed to resolve it - in effect, we'll make it less important by trying to make good guesses as to what the decision is going to be.

7.What is meant by speculative execution? (May/June 2012)

Speculative execution is an optimization technique where a computer system performs some task that may not be actually needed. The main idea is to do work before it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work after it is known whether it is needed. If it turns out the work was not needed after all, any changes made by the work are reverted and the results are ignored.

The objective is to provide more concurrency if extra resources are available. This approach is employed in a variety of areas, including branch prediction in pipelined processors, prefetching memory and files, and optimistic concurrency control in database systems.

8. What is meant by pipelining and pipeline stall? (April/May 2010)

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Pipelining improves performance by increasing instruction

throughput, as opposed to decreasing the execution time of an individual instruction.

Pipeline stall, also called bubble, is a stall initiated in order to resolve a hazard. They can be seen elsewhere in the pipeline.

9. What are the advantages of pipelining? (April/May 2010)(May/June 2016)

- The cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases.
- Some combinational circuits such as adders or multipliers can be made faster by adding more circuitry. If pipelining is used instead, it can save circuitry vs. a more complex combinational circuit

10.Define pipeline speedup. (Nov/Dec 2013)

The ideal speedup from a pipeline is equal to the number of stages in the pipeline.

Speedup = Time per instruction on unpipelined machine / Number of pipe stages.

However, this only happens if the pipeline stages are all of equal length. Splitting a 40 ns operation into 5 stages, each 8 ns long, will result in a 5x speedup. Splitting the same operation into 5 stages, 4 of which are 7.5 ns long and one of which is 10 ns long will result in only a 4x speedup.

- If your starting point is a multiple clock cycle per instruction machine then pipelining decreases CPI.
- If your starting point is a single clock cycle per instruction machine then pipelining decreases cycle time.

11. What is meant by data path element? (April/May 2014)

A data path element is a unit used to operate on or hold data within a processor. In the MIPS implementation, the data path elements include the instruction and data memories, the register file, the ALU, and adders.

PART B

1. BUILDING DATA PATH

❖ **Explain Data path and its control in detail .(Nov/Dec 2014). (16)**

The major components needed for datapath design are called as **datapath elements.**

Data Path Element

It is a unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements are the instruction and data memories, Program Counter, Adder, the register file, ALU, and Sign Extension Unit.

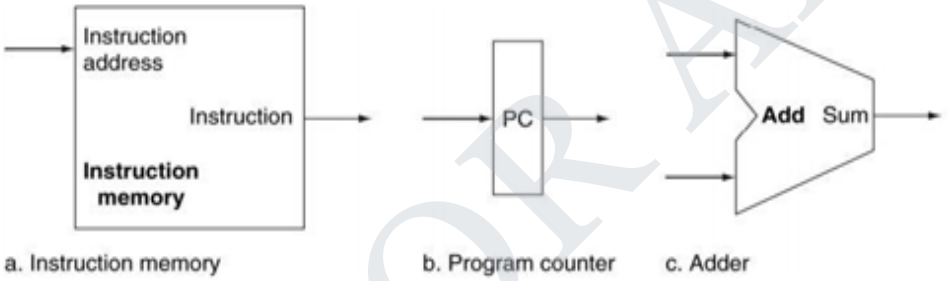


Figure 3.1 Components

Instruction Memory Unit is used to store the instructions of a program and supply instructions given an address.

Program Counter (PC), is a 32 bit register that holds the address of the current instruction.

Adder is a combinational circuit element used to add two inputs and place the sum on its output.

Register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file.

For each data word to be **read** from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers.

To **write** a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

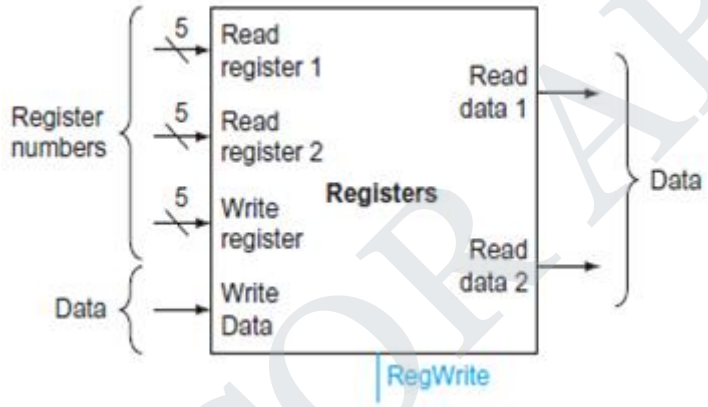


Figure 3.2 Register File

In Figure 3.2 we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers (32 = 25), whereas the data input and two data output buses are each 32 bits wide.

Arithmetic Logic Unit(ALU)

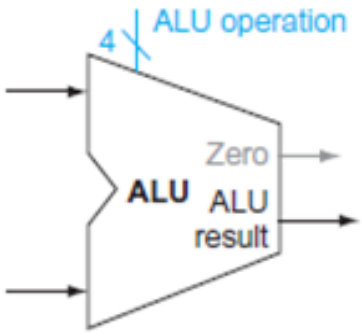


Figure 3.3 ALU

ALU which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0.

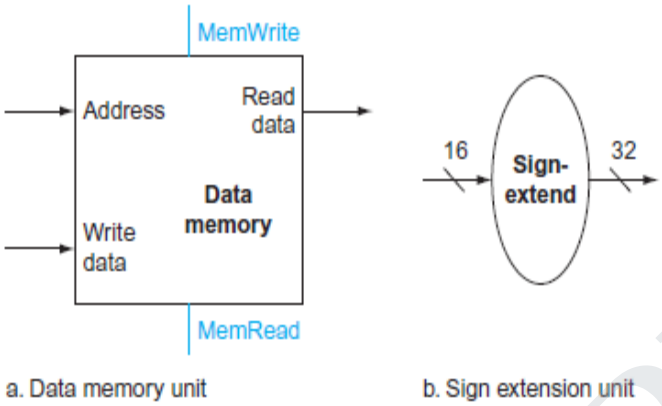


Figure 3.4 Components

Data Memory Unit

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems,

Sign Extension Unit

The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output. It will increase the size of a data item by replicating the high-order sign bit of the original data item in the high order bits of the larger, destination data item.

Datapath for fetching and incrementing the program counter

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points to the next instruction, 4 bytes later.

Figure 3.5 shows how to combine the three elements to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

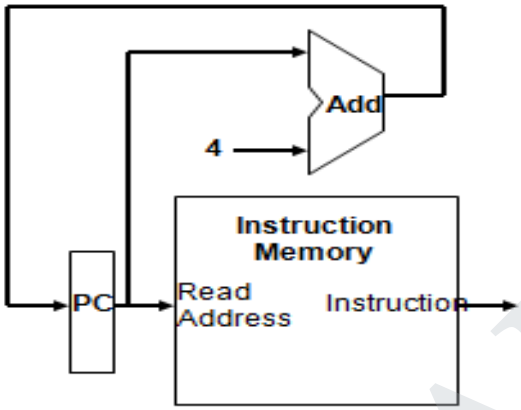


Figure 3.5 Datapath used for fetching and incrementing the program counter

Data Path for Arithmetic Logical Instructions

Consider the **R-format instructions**. To perform ALU operation these instructions read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. This instruction class includes add, sub, AND, OR, and slt,

For example add $t1,$t2,$t3,

which reads $t2 and $t3, performs addition and writes into $t1.

The data elements needed for arithmetic Logical Instructions are

- Register File
- ALU
- Sign Extension Unit

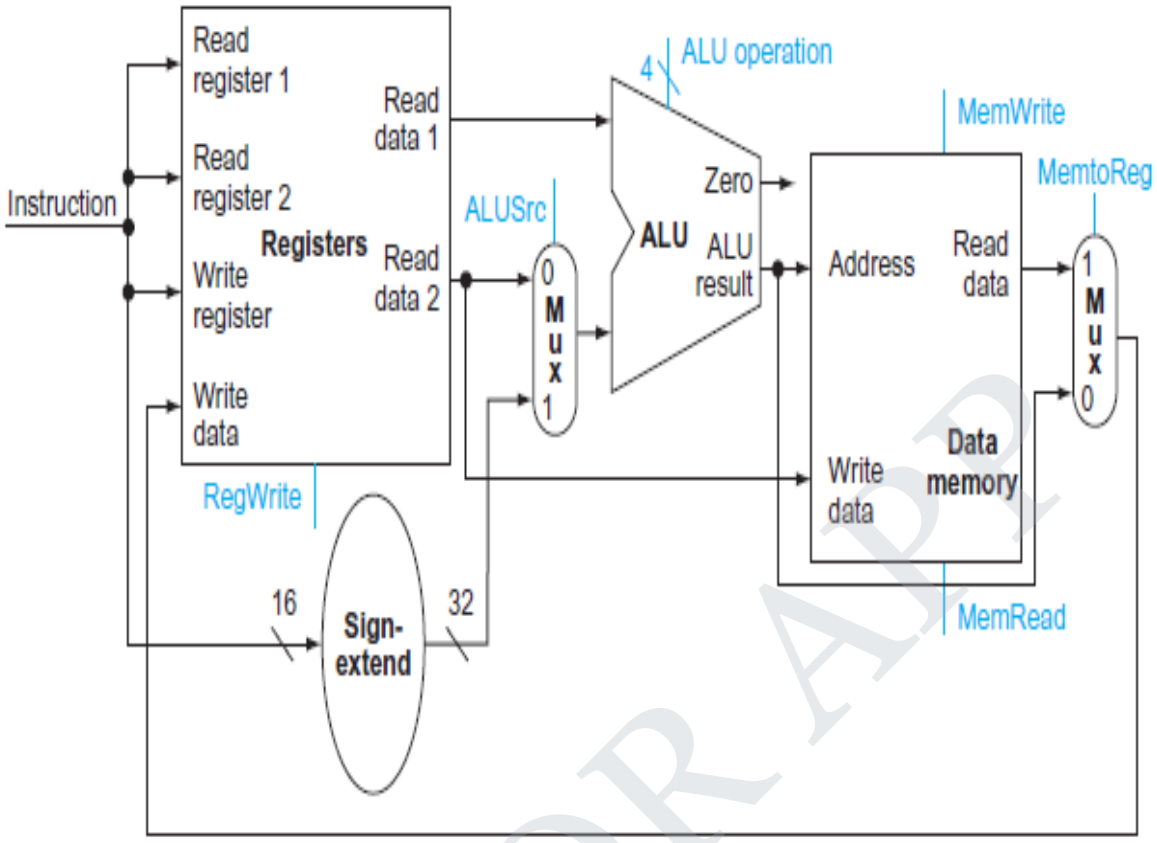


Figure 3.6 The datapath for R-type instructions.

Data Path for Load Word and Store Word Instructions

lw $t1,offset_value($t2)
sw $t1,offset_value($t2).

These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed off set field contained in the instruction.

If the instruction is a **store**, the value to be stored must also be read from the register file where it resides in $t1.

If the instruction is a **load**, the value read from memory must be written into the register file in the specified register, which is $t1. Thus, we will need both the register file and the ALU

Units needed to implement Loads and Stores

- Register File
- ALU
- Data Memory Unit
- Sign Extension Unit

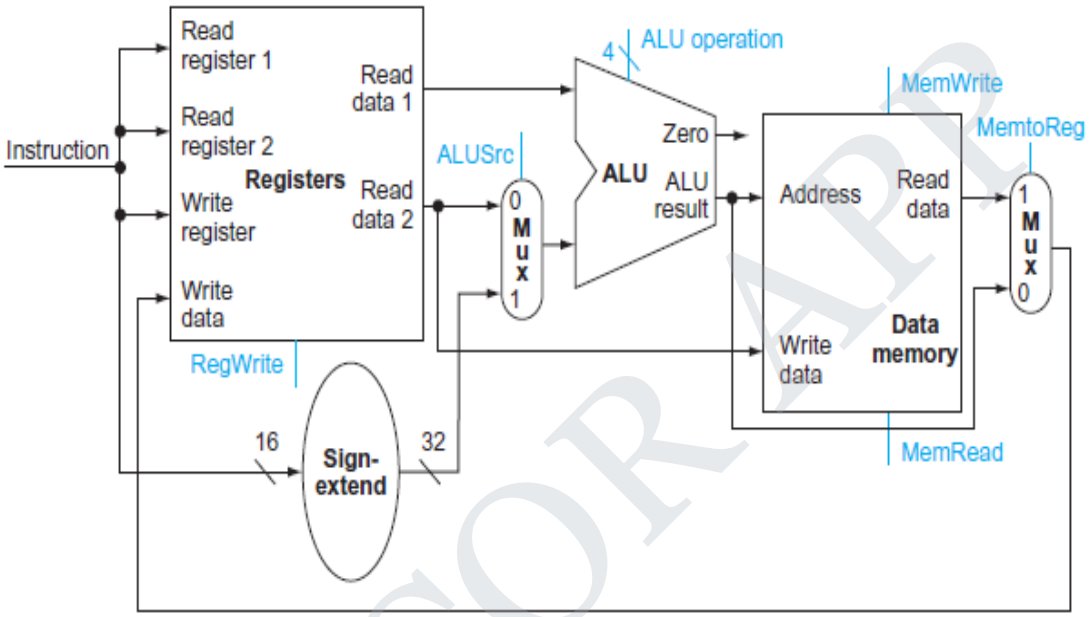


Figure 3.7 The datapath for the memory instructions

Data Path for Branch Instructions

There are two kinds of branch instructions

Beq – branch equal

Bnq- branch unequal

The beq instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the **branch target address** relative to the branch instruction address.

Example beq $t1,$t2,offset.

To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC.

Branch Target Address Calculation

The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4.

For computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is **true** (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operands are **not equal**, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch** is **not taken**.

Thus, the branch datapath must do two operations:

- Compute the branch target address
- Compare the register contents.

Figure 3.8 shows the structure of the datapath segment that handles branches. The unit labeled ***Shift left 2*** is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended off set field; no actual shift hardware is needed, since the amount of the "shift " is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Datapath for the Core MIPS Architecture

Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch, the datapath from R-type and memory instructions and the datapath for branches

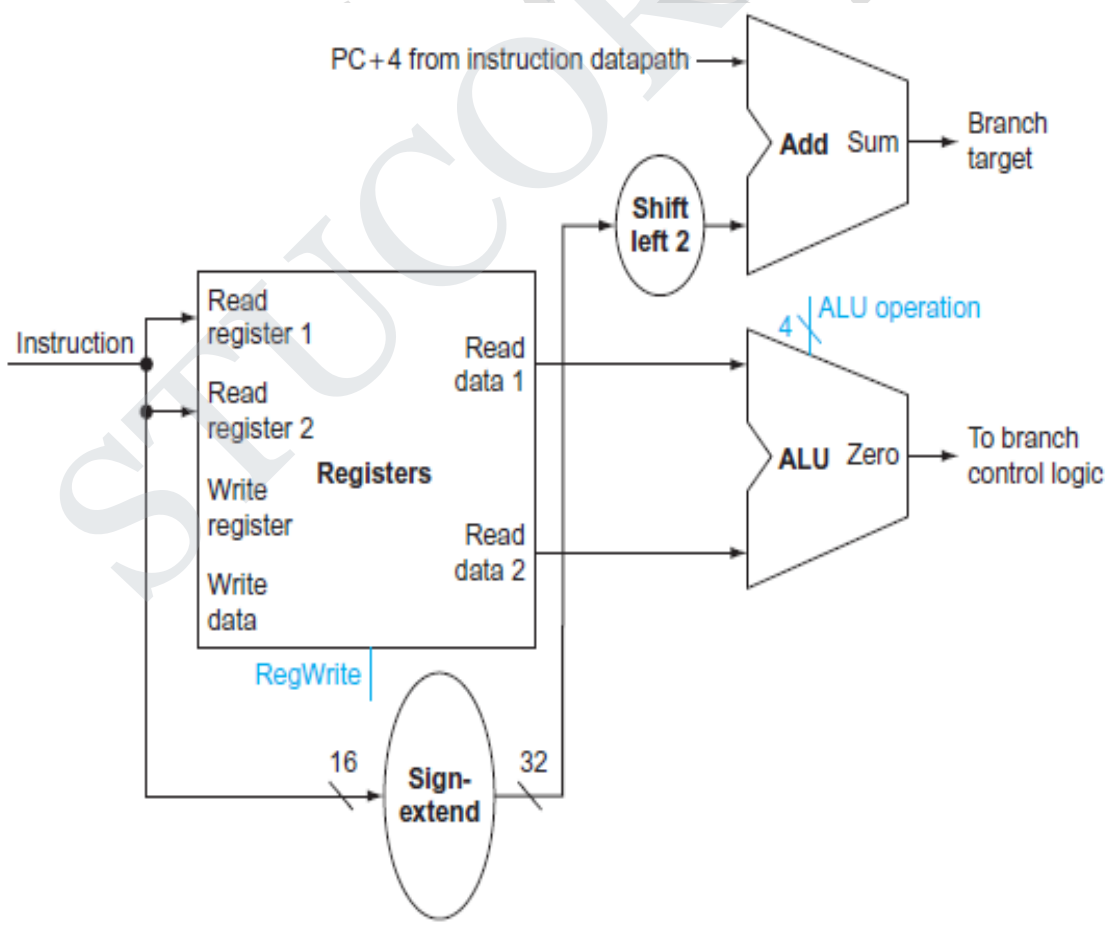


Figure 3.8 Datapath segment that handles branches.

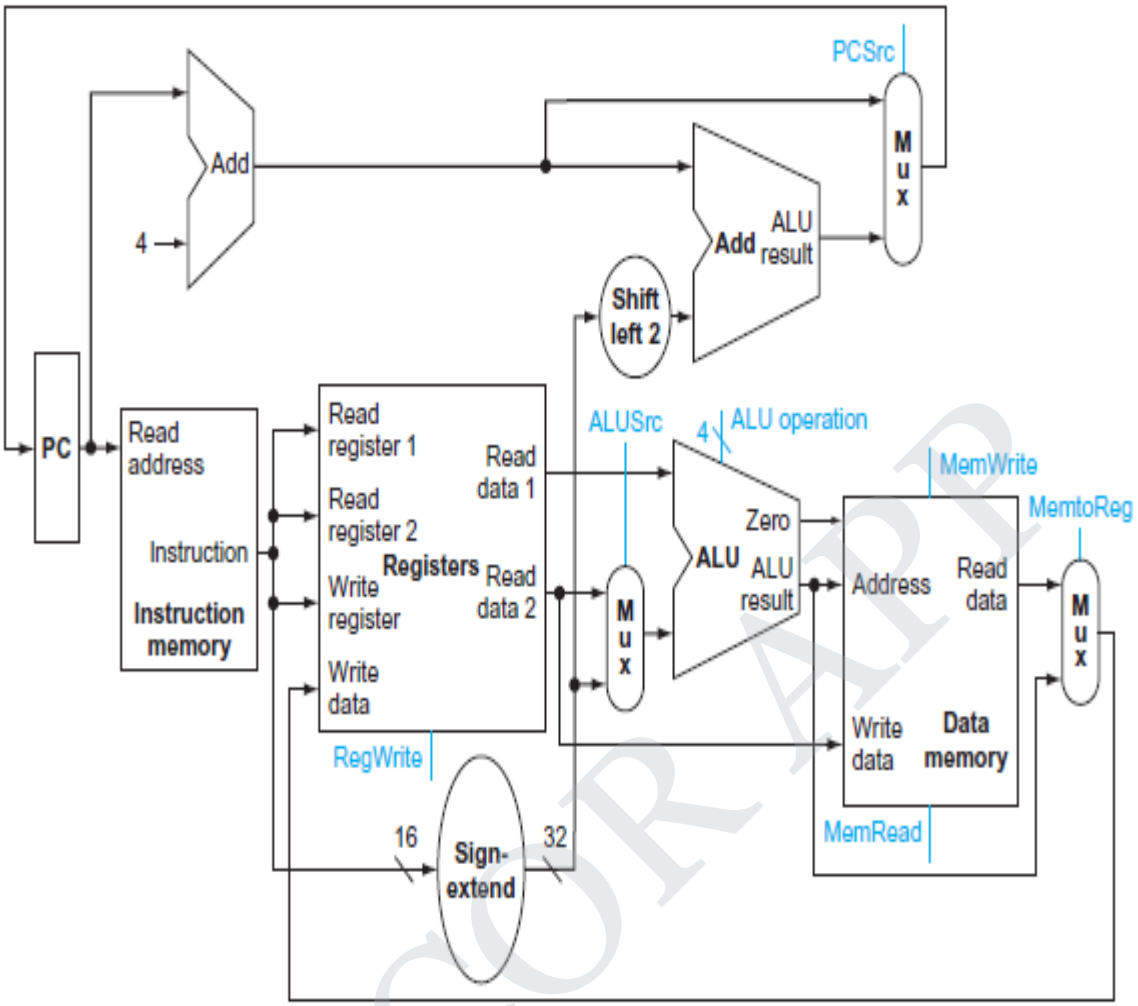


Figure 3.9 Datapath for the Core MIPS Architecture

The datapath is obtained by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder from Figure 4.9 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

2. CONTROL IMPLEMENTATION SCHEME

❖ **Explain the basic MIPS implementation with necessary multiplexers and Control Lines (Nov/Dec 2015) (16)**

Control implementation scheme can be build using datapath and some simple

implementation method. It covers load word, store word, branch equal and arithmetic logic instructions- add, sub, AND, OR and set on less than.

The ALU Control:

The MIPS ALU defines the 6 following combinations of four control inputs:

ALU Control Lines	Functions
0000	AND
0001	OR
0010	Add
0111	Subtract
1100	Set on less than
1100	NOR

Depending on the instruction class, the ALU needs to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set not found in the subset we are implementing.) For load word and store word instructions, we use the ALU to compute the memory address by addition.

For **R type instructions** the ALU has to perform one of the five operations (AND,OR, add, Subtract or set less than). For branch equal the ALU has to perform subtraction.

The 4 bit ALU control input is generated using a small control unit that has input function field of the instruction and a 2 bit control field it is called ALUop. ALUop indicates three kinds of operations

00 - Add for loads and stores operations

01 - Subtract for branch equal

10 – Determined by the operation encoded in the function field.

The output of the ALU control unit is a 4 bit signal and it directly controls the ALU by generating one of the 4 bit combinations.

The table shows how to set ALU control inputs based on the 2 bit ALUop and 6 bit function code

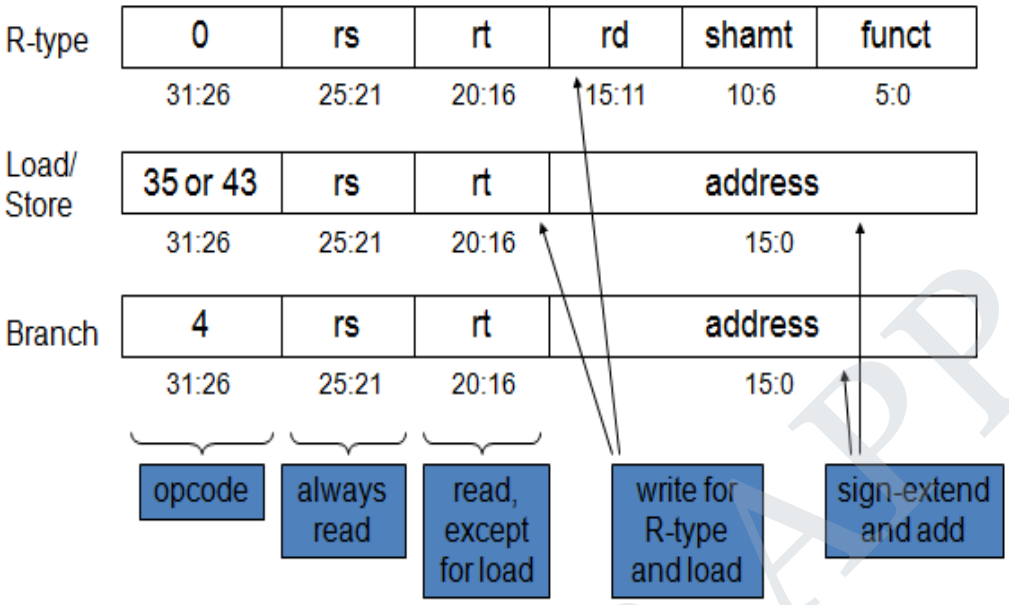
Instruction Opcode	ALUop	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	Load word	xxxxxx	add	0010
SW	00	Store sord	xxxxxx	add	0010
Branch Equal	01	Branch Equal	xxxxxx	Subtract	0110
R-Type	10	Add	100000	add	0010
		Subtract	100010	Subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		Set on less than	101010	Set on less than	0111

.
When ALUOP = 00 or 01 , the action depends on the control input
When ALUOP = 10 , the action depends on the control input and function field.

Designing the Main Control Unit:

To design the main control unit, the fields of an instruction and the control lines must be identified. The fields are used to form the connection in a single data path and Control Lines are needed for data path connection. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions.

The three instruction classes (R-type, load and store, and branch) use two different instruction formats:



The major observations about this instruction format are

- The op field, also called the opcode, is always contained in bits 31:26. It refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).
- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Using this information we can add the instruction labels and extra multiplexer to the simple datapath.

The figure 3.10 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

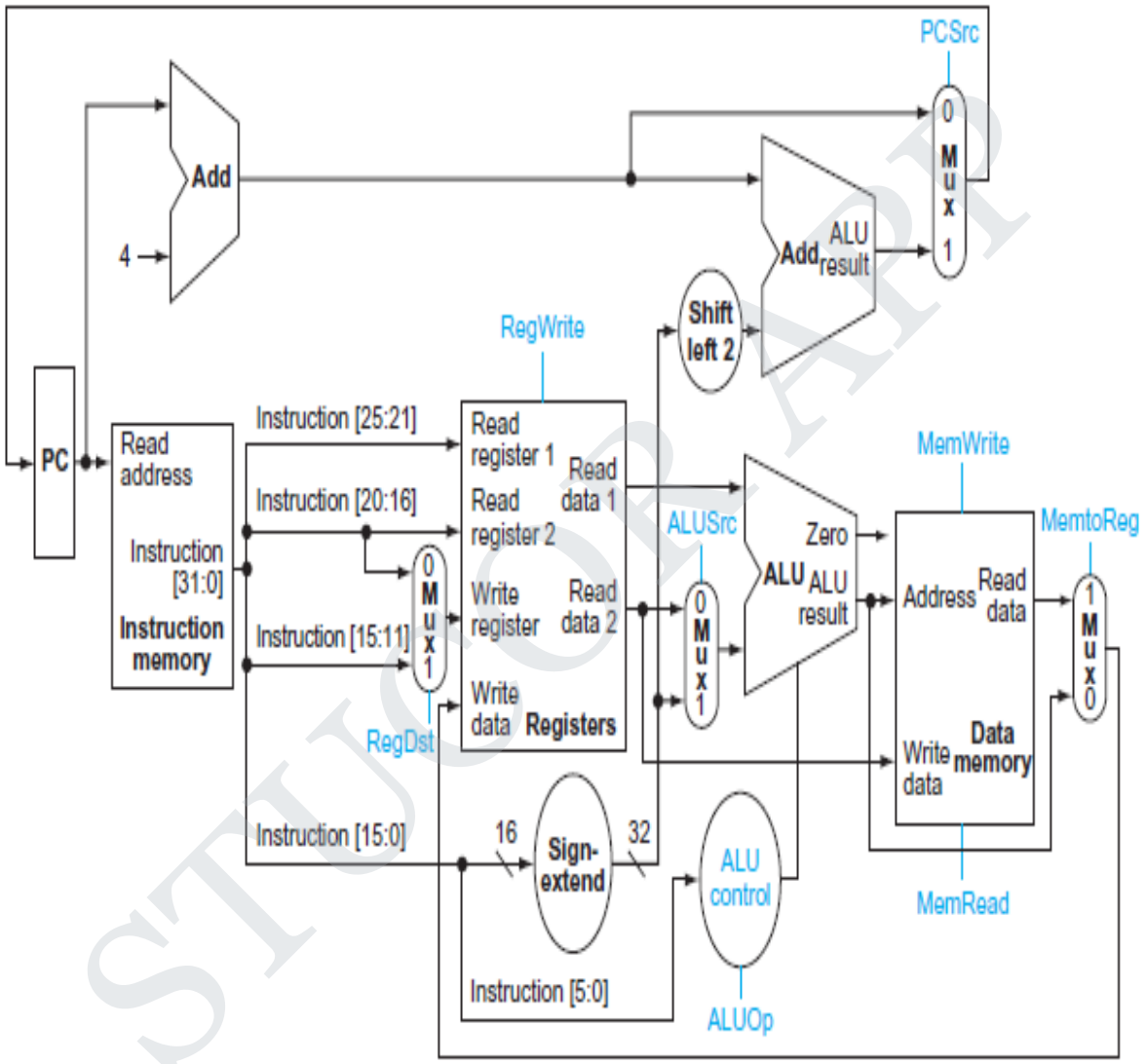


Figure 3.10 Datapath with multiplexors and all control lines

Signal Name	Effect When Deasserted	Effect When Asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).

Signal Name	Effect When Deasserted	Effect When Asserted
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 3.11 Effect of the function of these seven control lines.

Figure 3.11 describes the function of the seven control lines. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception.

That control line should be asserted if the instruction is branch on equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for equality comparison, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These nine control signals (seven from Figure 3.11 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26.

Figure 3.12 shows the datapath with the control unit and the control signals. The input of the control unit consists of three 1 bit signals that are used to control multiplexers and three signals for controlling reads and writes in the register file and data memory.

1 bit signal in determining whether to possibly branch and 2 bit control signal used for the ALUOP. AND gate is used to combine the branch control signal and the zero output from the ALU. AND gate output controls the selection of next PC.

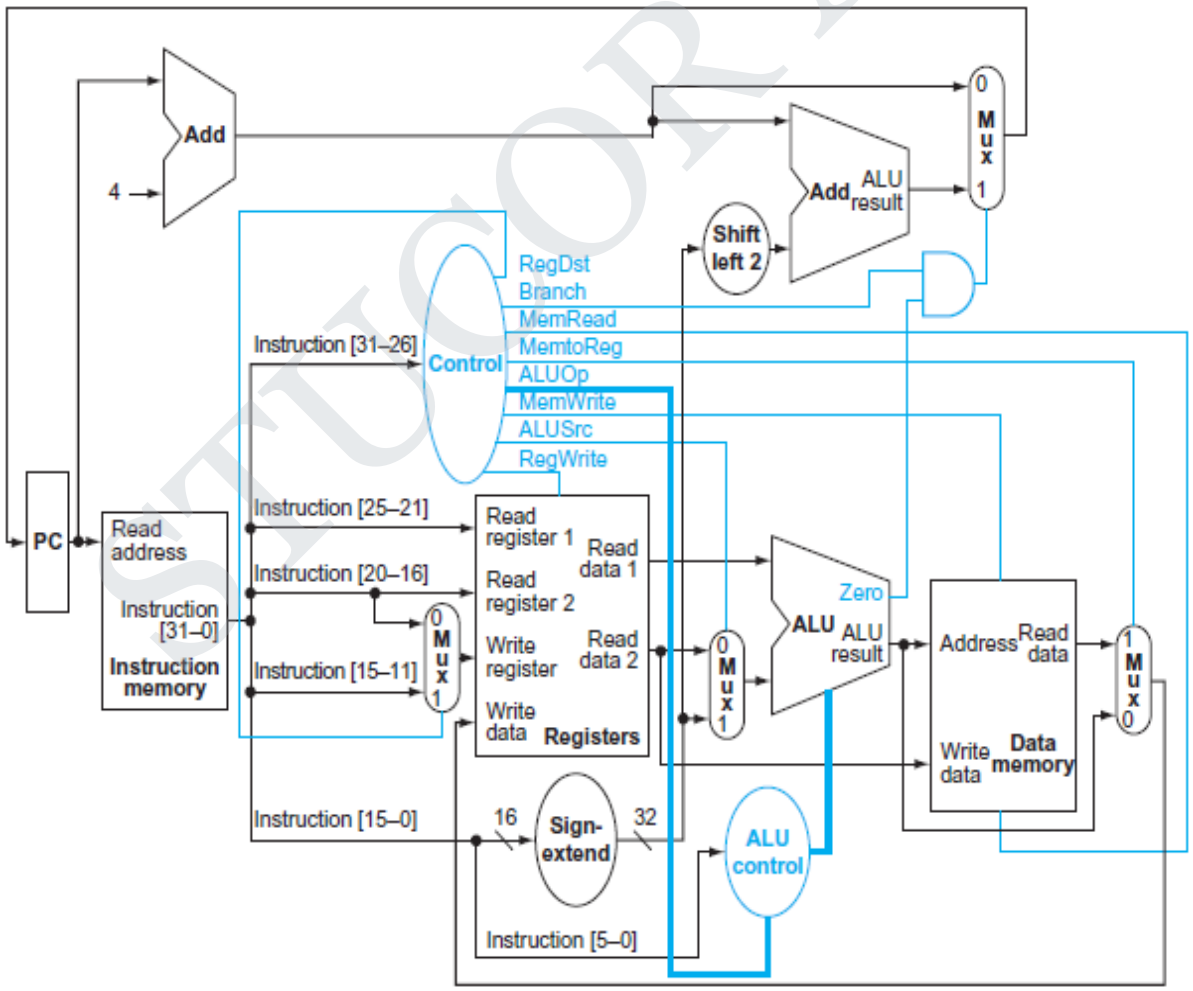


Figure 3.12 shows the datapath with the control unit and the control signals.

Operation of the Datapath

Operation of the Datapath for an R-Type Instruction

add $t1,$t2,$t3

Four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.
2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

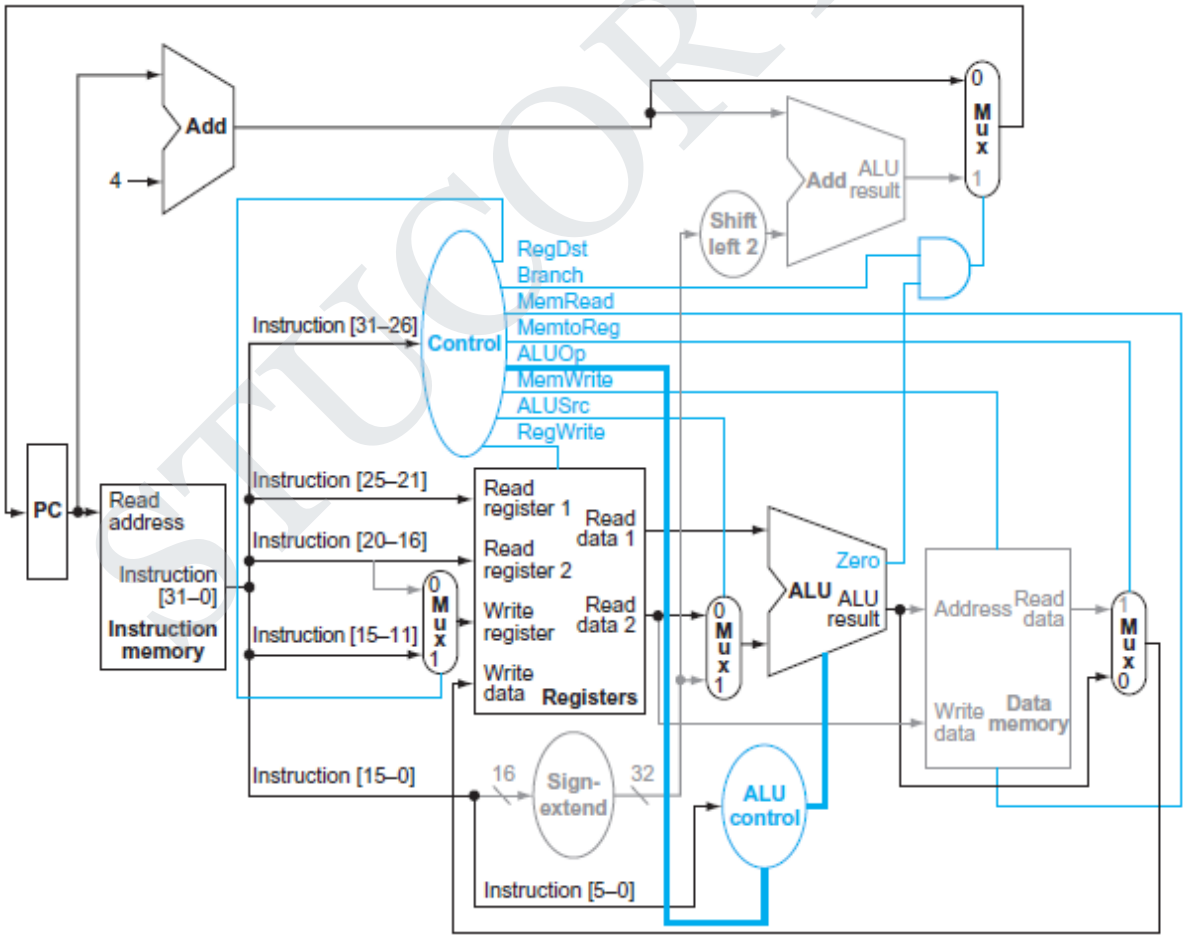


Figure 3.13 Datapath for an R-Type Instruction.

Operation of the Datapath for an load word Instruction

lw $t1, offset($t2)

Figure 3.14 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

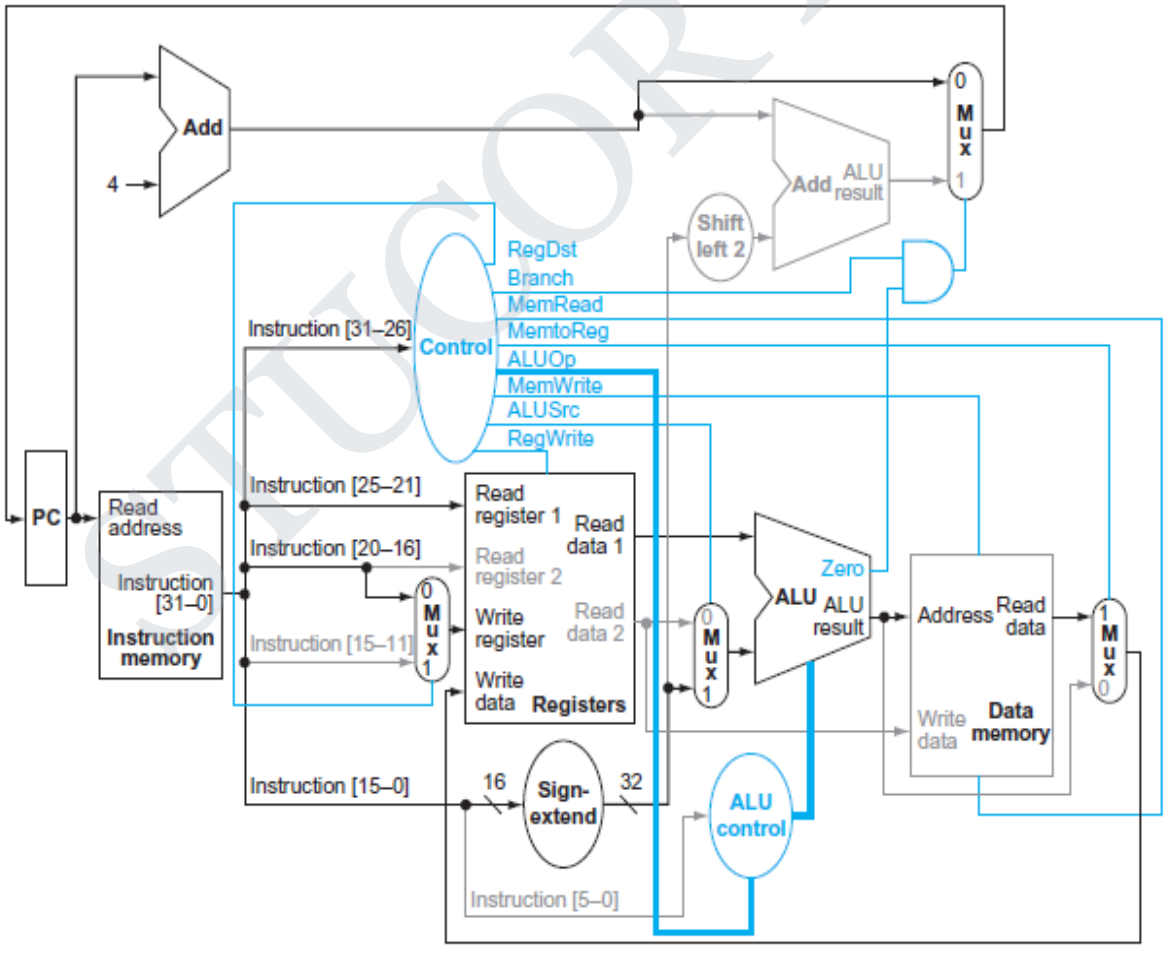


Figure 3.14 Datapath for an load word Instruction.

Operation of the Datapath for an the branch-on-equal instruction.

beq $t1, $t2, offset,

It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

Figure 3.15 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $t1 and $t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

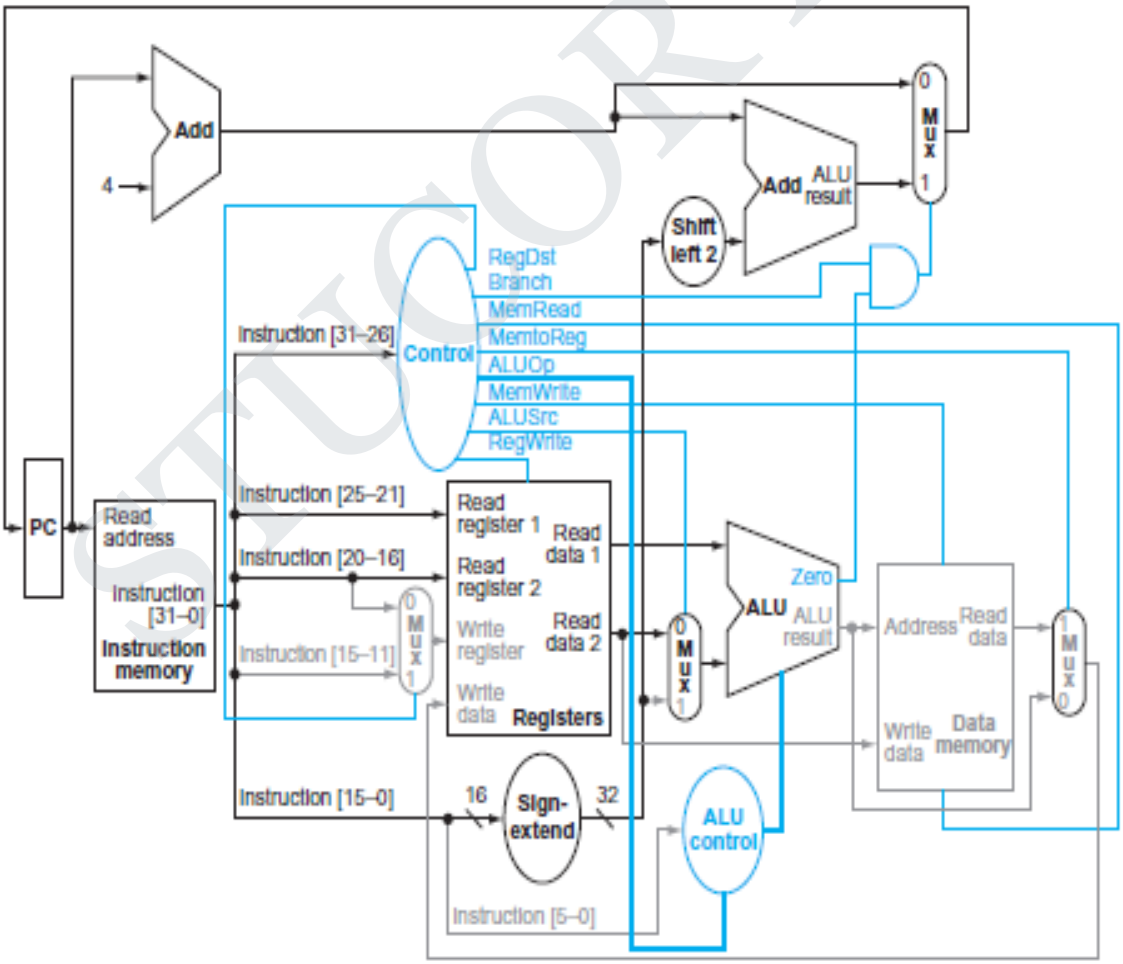


Figure 3.15 Datapath for branch word Instruction.

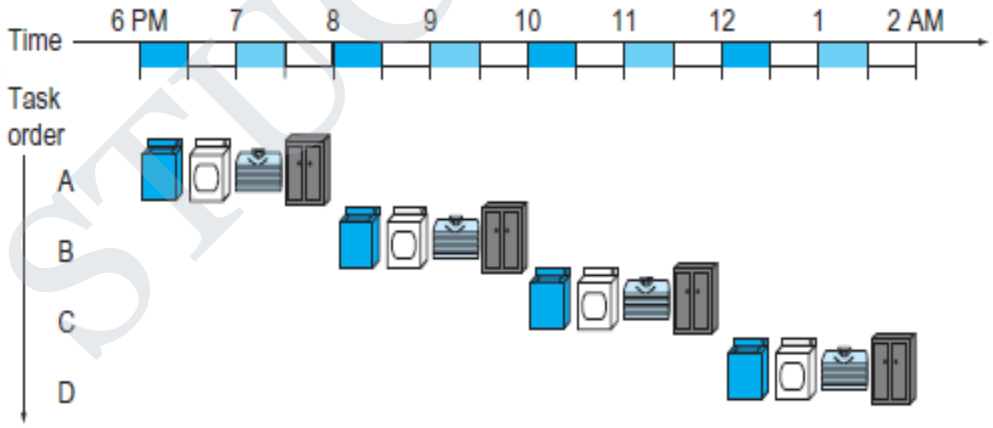
3. PIPELINING

❖ **What is pipelining? Discuss about pipelined data path control.(May/June 2016)**

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Used to increase the speed and performance of the processor.

Without a pipeline, a computer processor gets the first instruction from memory, performs the operation it calls for, and then goes to get the next instruction from memory, and so forth. While fetching (getting) the instruction, the arithmetic part of the processor is idle. It must wait until it gets the next instruction.

With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can be performed. The staging of instruction fetching is continuous. The result is an increase in the number of instructions that can be performed during a given time period.



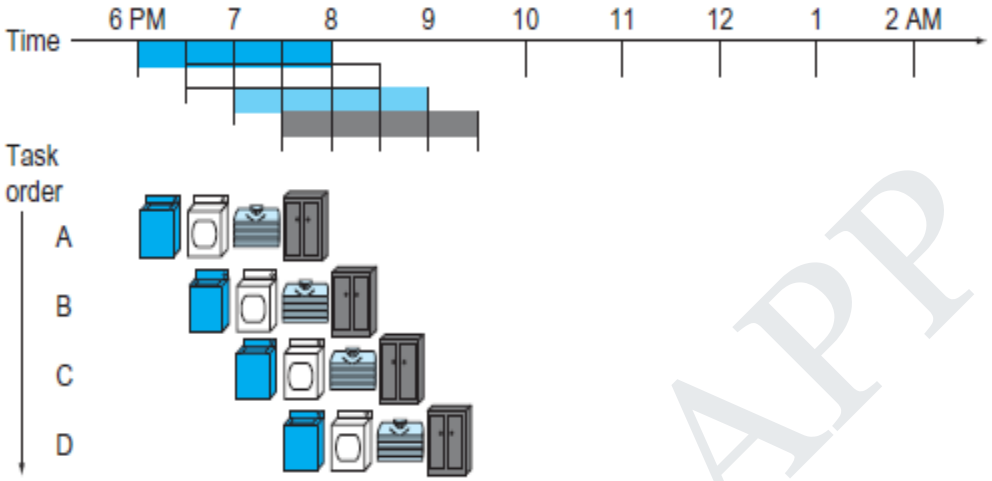
The *nonpipelined* approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.

4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, start over with the next dirty load.

The laundry analogy for pipelining.



The *pipelined* approach takes much less time. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour. Pipelining improves throughput of our laundry system.

Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work. If all the stages take about the same

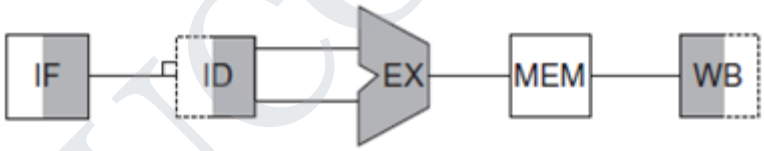
amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline, in this case four: washing, drying, folding, and putting away.

Pipelining in Processor

The same principles apply to processors where we pipeline instruction-execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory. **(IF)**
2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously. **(ID)**
3. Execute the operation or calculate an address. **(EX)**
4. Access an operand in data memory. **(MEM)**
5. Write the result into a register. **(WB)**

Hence, the MIPS pipeline has five stages. The five stages are



- **Instruction fetch cycle (IF)**
- **Instruction decode/register fetch cycle (ID)**
- **Execution/effective address cycle(EX)**
- **Memory access(MEM)**
- **Write-back cycle (WB)**

1. Instruction fetch cycle (IF):

Send the program counter (PC) to memory and fetch the current instruction from memory. PC=PC+4.

2. Instruction decode/register fetch cycle (ID):

Decode the instruction and read the registers. Do the equality test on the registers as they are read, for a possible branch. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture, known as fixed-field decoding.

3. Execution/effective address cycle(EX):

The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

- **Memory reference:** The ALU adds the base register and the offset to form the effective address.
- **Register-Register ALU instruction:** The ALU performs the operation specified by the ALU opcode on the values read from the register file.
- **Register-Immediate ALU instruction:** The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

4. Memory access(MEM):

If the instruction is a **load**, memory does a read using the effective address computed in the previous cycle. If it is a **store**, then the memory writes the data from the second register read from the register file using the effective address.

5. Write-back cycle (WB):

Write the result into the register or memory.

Designing Instruction Sets for Pipelining

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.

Fourth, operands must be aligned in memory. The requested data can be transferred between processor and memory in a single pipeline stage.

4. PIPELINED DATA PATH AND CONTROL

❖ **Explain about Pipelined Data Path and Control.**

The division of an instruction into five stages means a five stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

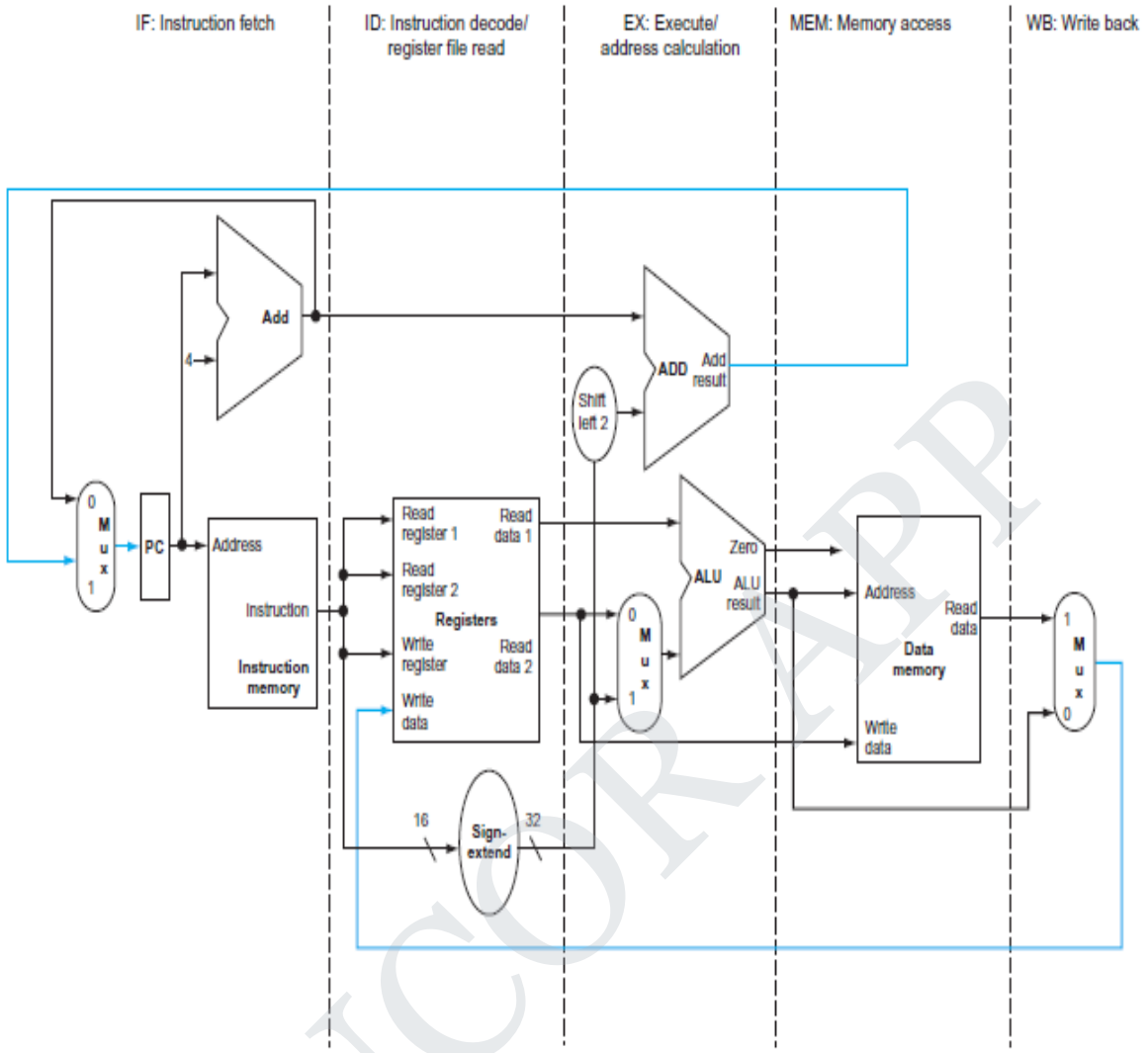


Figure 3.16 The single-cycle datapath

In figure 3.16 the five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.

There are, however, **two exceptions** to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

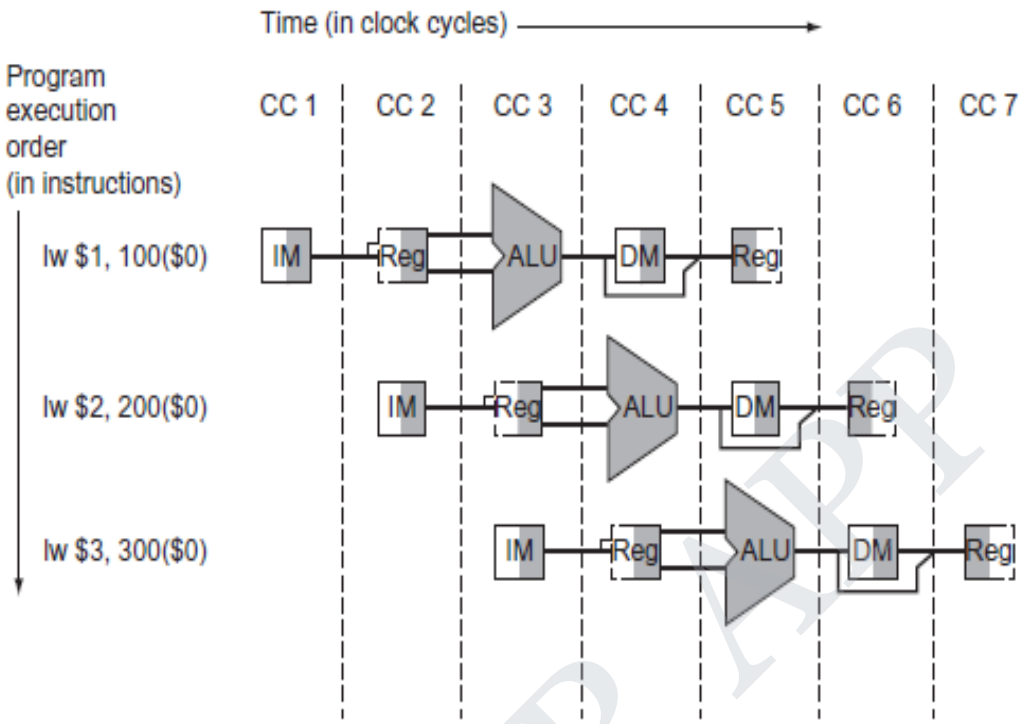


Figure 3.17 Instructions being executed using the single-cycle datapath

For example, figure 3.17 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages. To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages. ie. in our laundry analogy, having a basket between each pair of stages to hold the clothes for the next step.

.

4a. PIPELINED DATAPATH

Figure 3.18 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

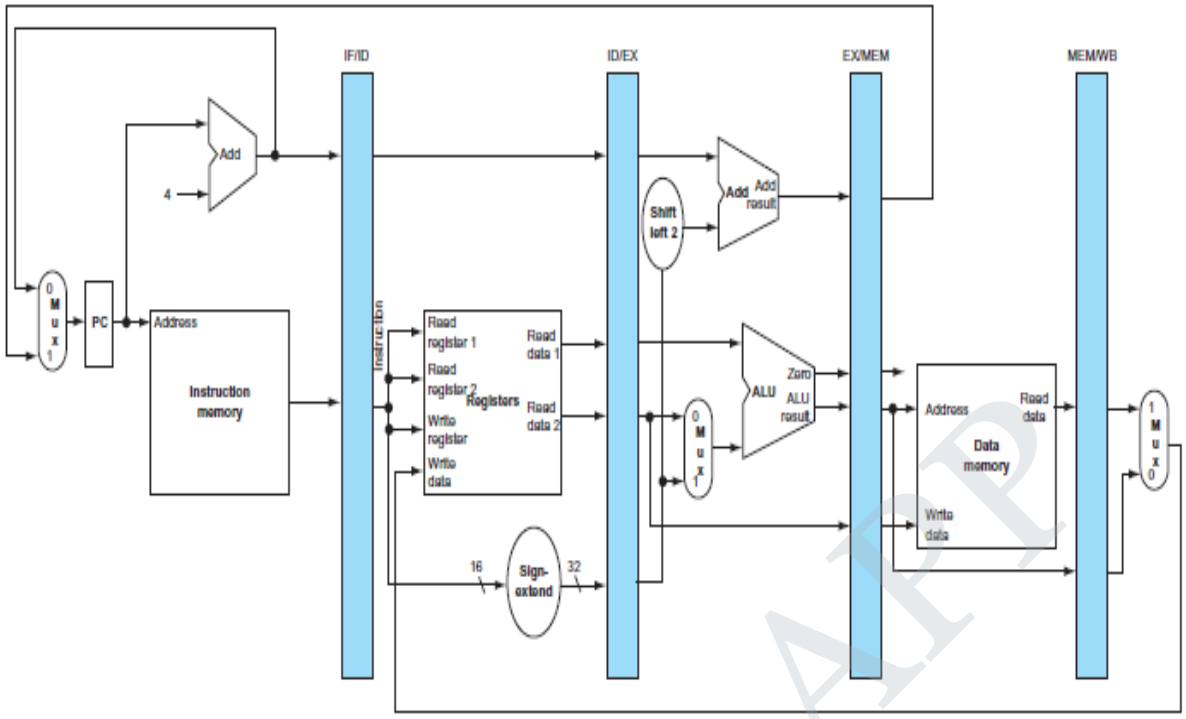


Figure 3.18 The pipelined version of the datapath

The pipeline registers separate each pipeline stage. The registers must be wide enough to store all the data corresponding to the lines that go through them.

FIVE STAGES OF LOAD INSTRUCTION:

Instruction Fetch

The figure 3.19 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

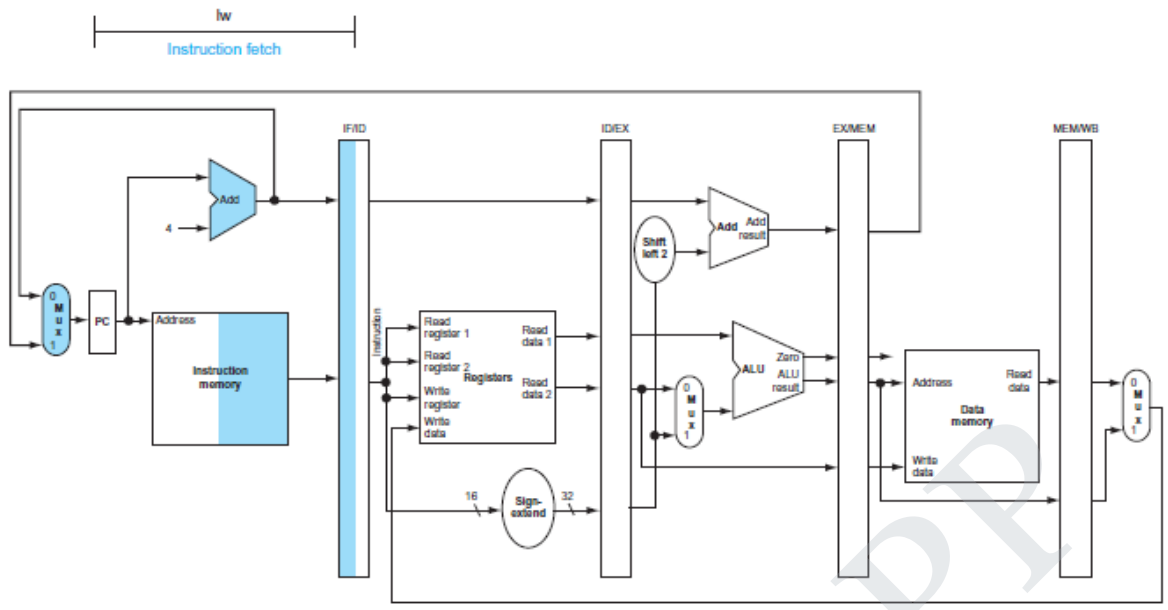


Figure 3.19 Instruction Fetch Pipeline

1. Instruction decode and register file read:

The figure 3.20 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

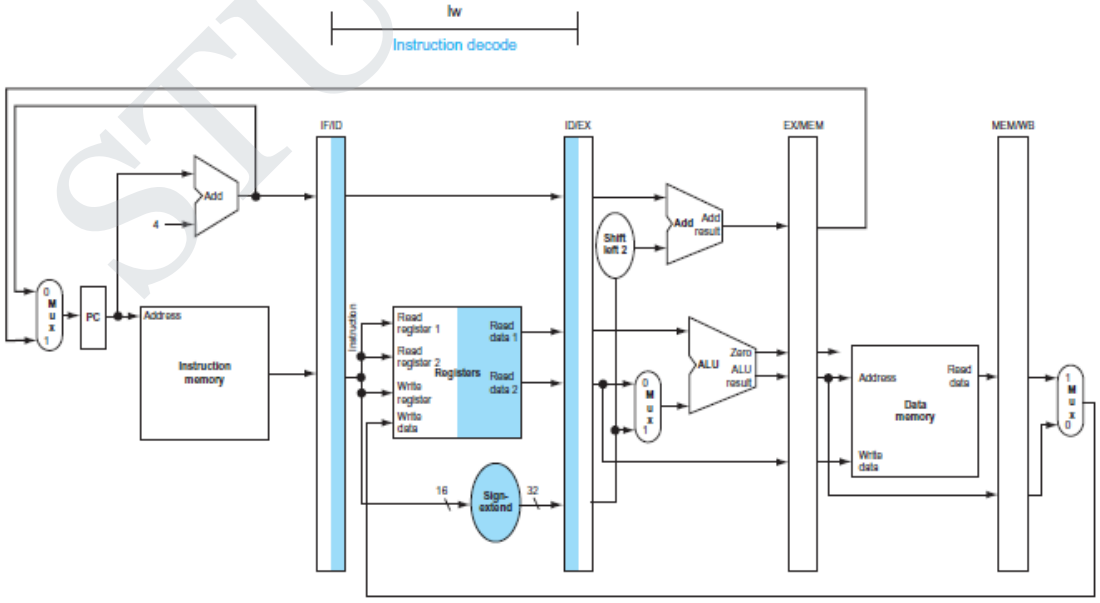


Figure 3.20 The second pipe stage of a load instruction

2. Execute or address calculation:

Figure 3.21 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

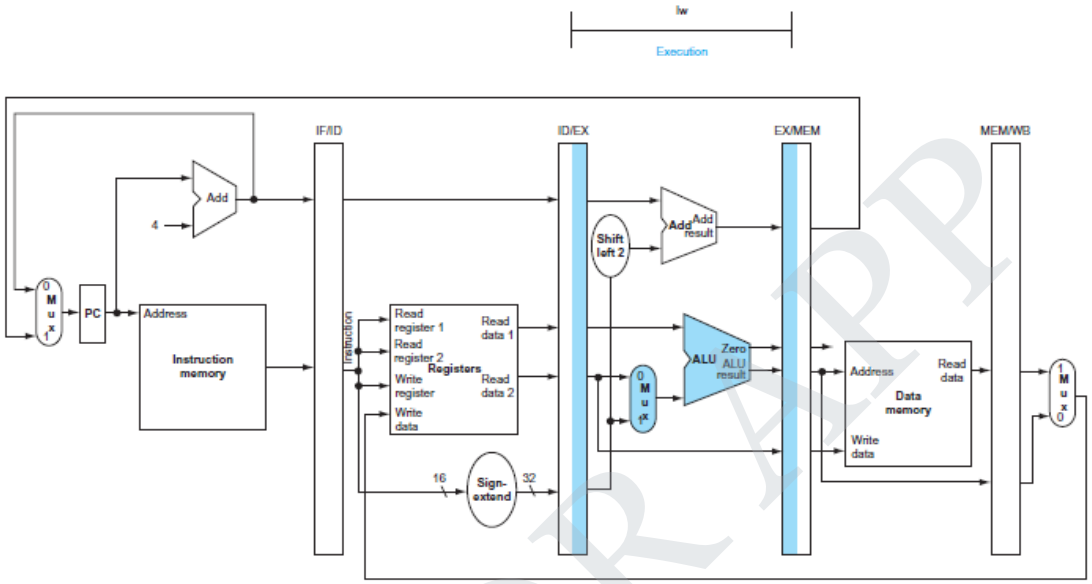


Figure 3.21 The third pipe stage of a load instruction

3. Memory access:

The figure 3.22 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

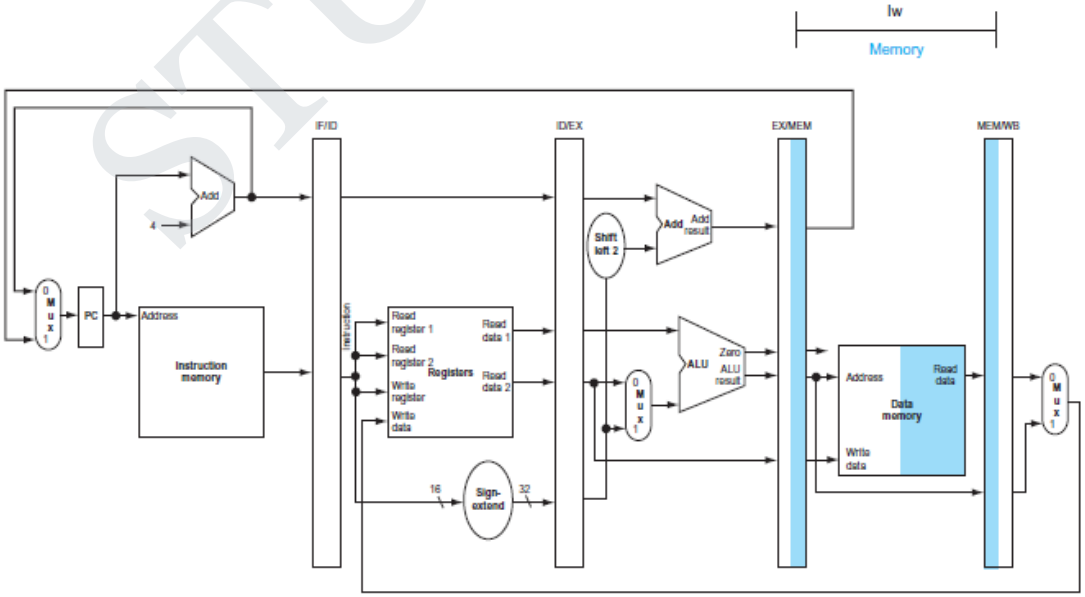


Figure 3.22 The fourth pipe stage of a load instruction

4. Write-back:

The figure 3.23 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

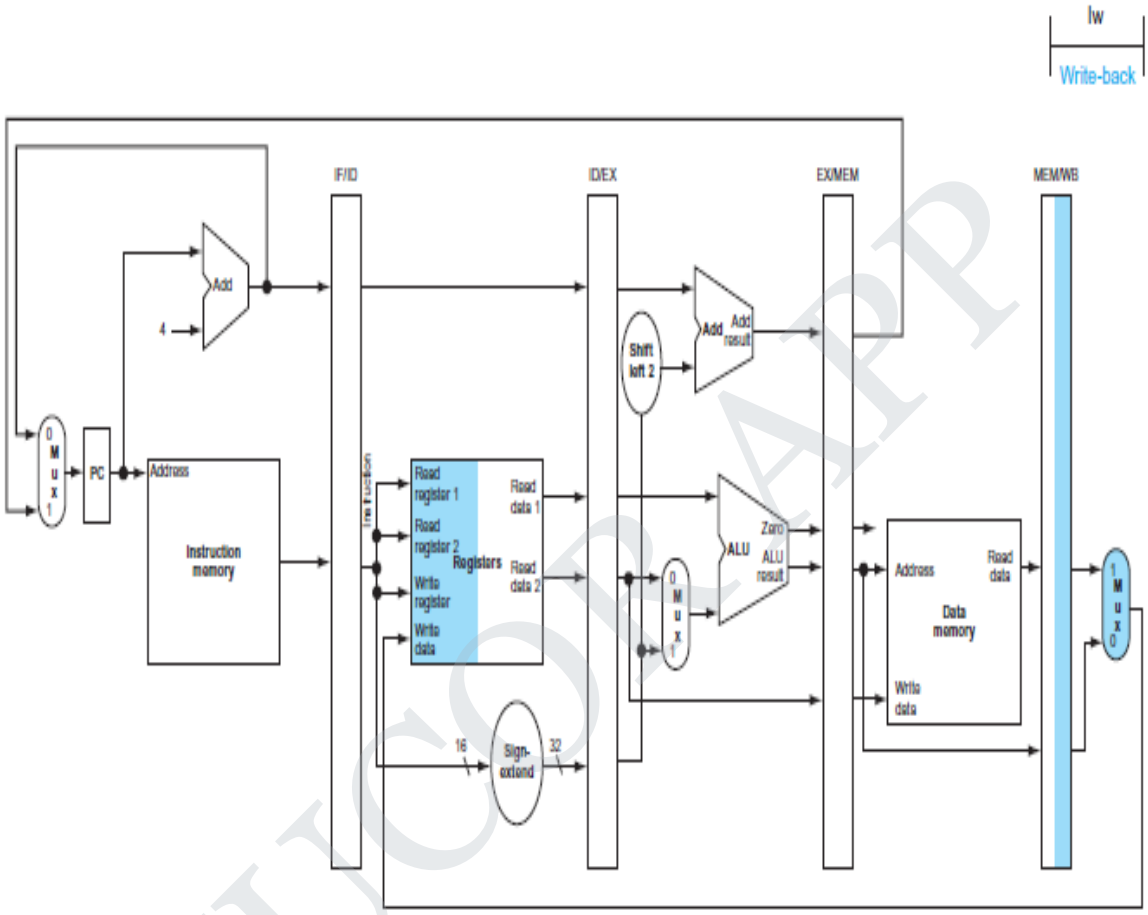


Figure 3.23 The fifth pipe stage of a load instruction

FIVE STAGES OF STORE INSTRUCTION:

1. **Instruction fetch**: Same as load instruction
2. **Instruction decode and register file read:** Same as load instruction
3. **Execute and address calculation:**

Figure 4.24 shows the third step; the effective address is placed in the EX/MEM pipeline register.

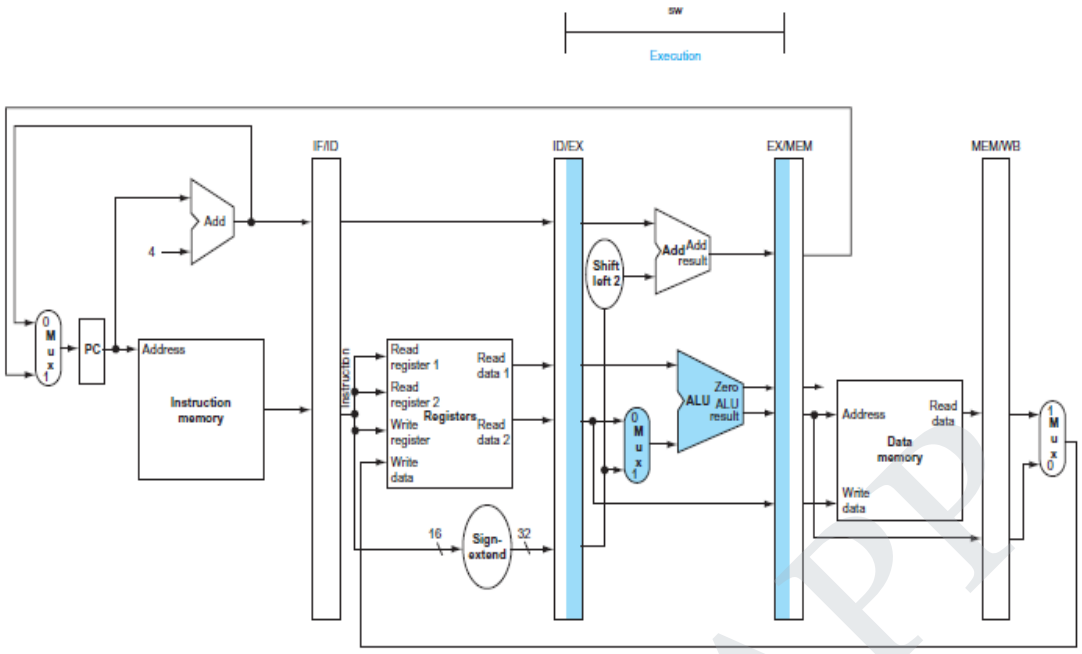


Figure 3.24 The third pipe stage of a store instruction

4. Memory access:

The figure 3.25 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

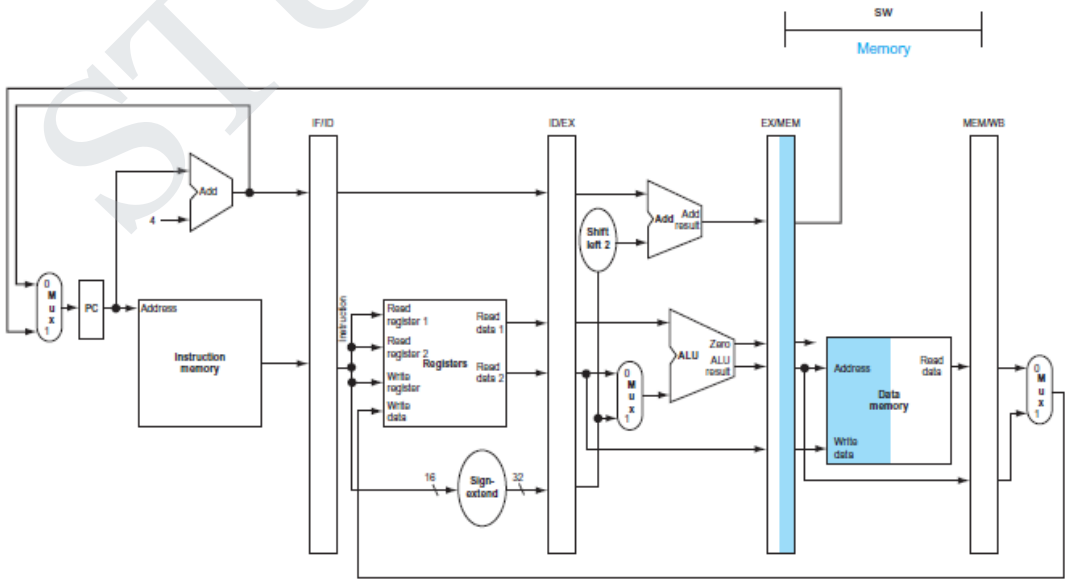


Figure 3.25 The fourth pipe stage of a store instruction

5. Write-back:

The figure 3.26 shows the final step of the store. For this instruction, **nothing happens in the write-back stage**. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

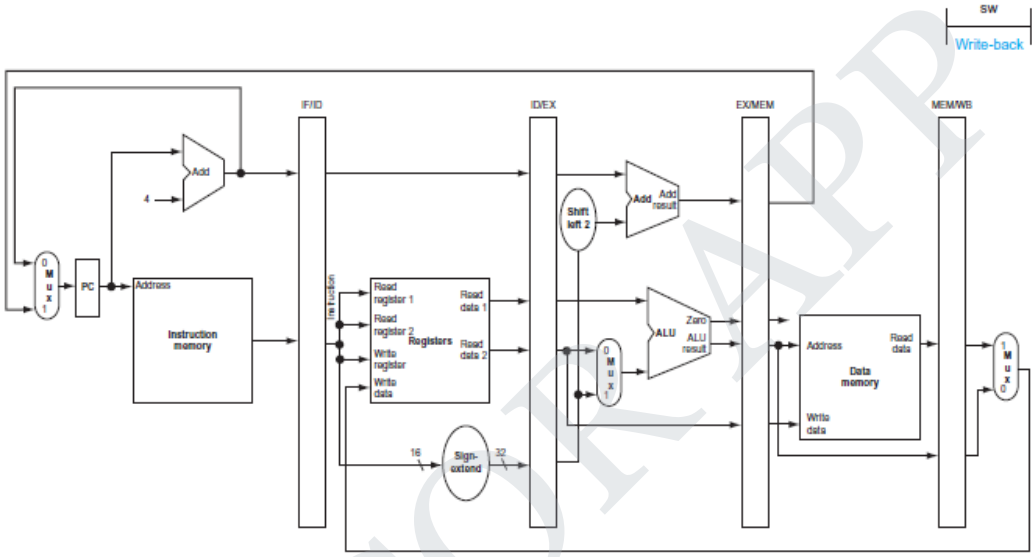


Figure 3.26 The fifth pipe stage of a store instruction

GRAPHICALLY REPRESENTING PIPELINES

There are two basic styles of pipeline figures:

- ***Multiple-clock-cycle pipeline diagrams***
- ***Single-clock-cycle pipeline diagrams***

Multiple-clock-cycle pipeline diagram of five instructions:

For example, consider the following five-instruction sequence:

lw $10, 20($1)

sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

add $14, $5, $6

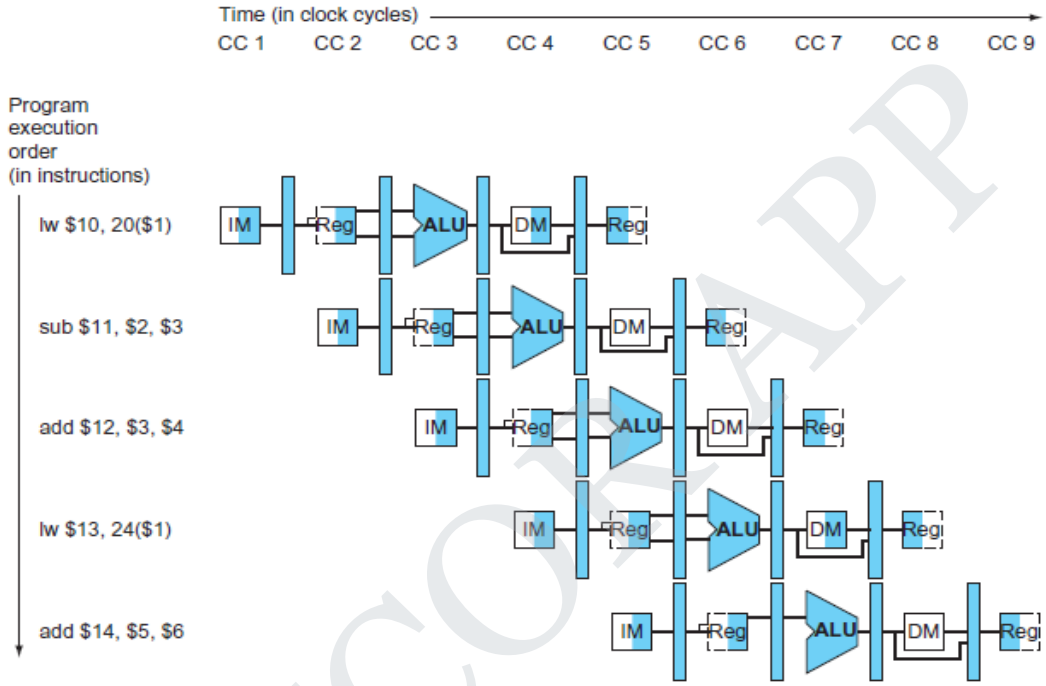


Figure 3.27 Multiple-clock-cycle pipeline diagram of five instructions.

Figure 3.27 shows the multiple clock cycle. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page, similar to the laundry pipeline. A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline graphically, but a rectangle naming each pipe stage works just as well. The physical resources used at each stage are shown.

Figure 3.28 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Uses the *name* of each stage.

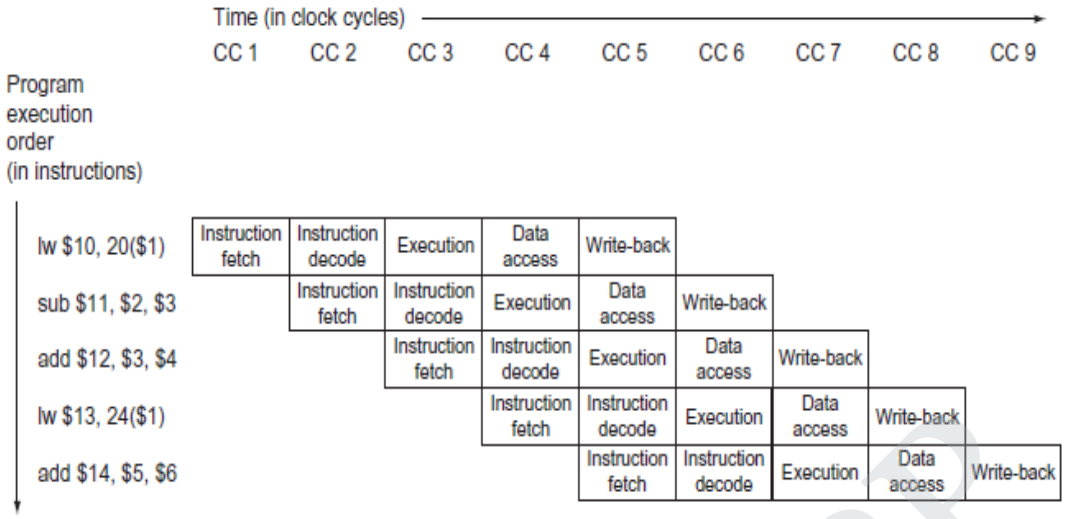


Figure 3.28 Traditional multiple-clock-cycle pipeline diagram of five instructions

Single-clock-cycle pipeline diagrams show the state of the entire datapath during a single clock cycle, and usually all five instructions in the pipeline are identified by labels above their respective pipeline stages. We use this type of figure 3.29 to show the details of what is happening within the pipeline during each clock cycle; typically the drawings appear in groups to show pipeline operation over a sequence of clock cycles. A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagrams, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle.

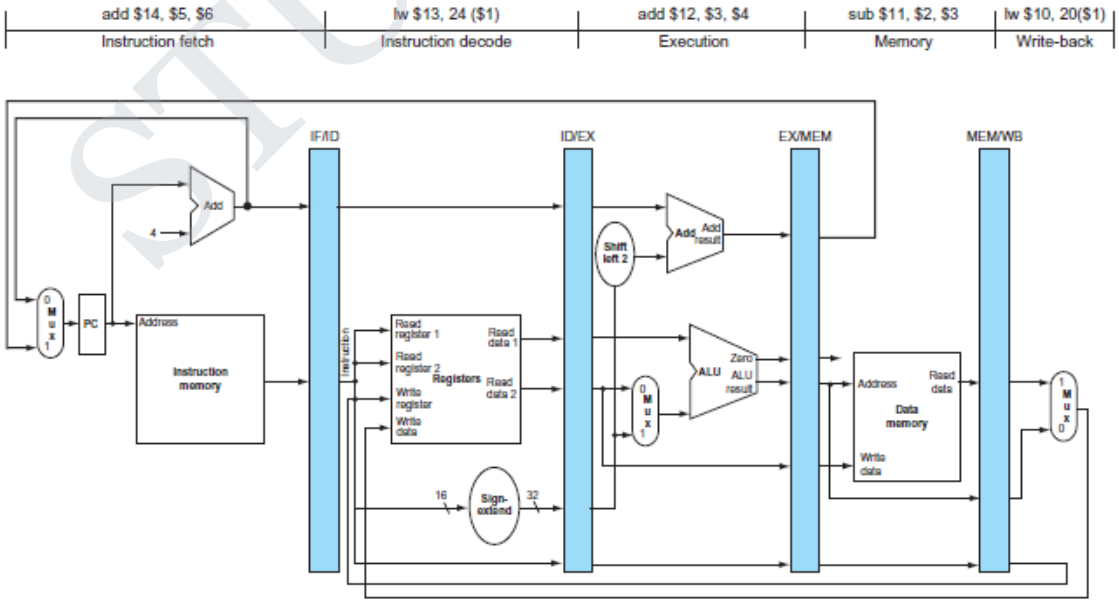


Figure 3.29 The single clock cycle diagram

The single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles.

4b.PIPELINED CONTROL

The first step is to label the control lines on the existing datapath. **(Refer the figure 3.11 for control signals)**

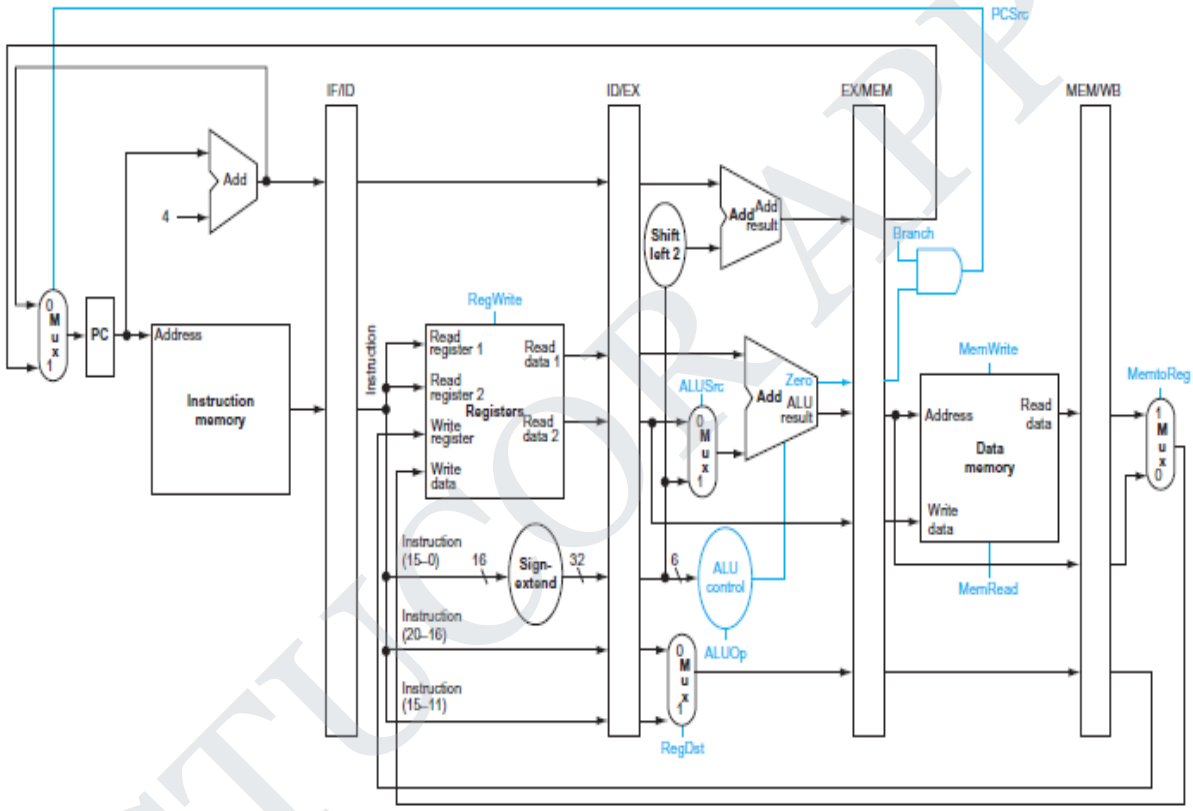


Figure 3.30 The pipelined datapath of with the control signals identified.

Figure 3.30 shows those lines. We borrow as much as we can from the control for the simple datapath in Figure. In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. **Instruction fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

2. **Instruction decode/register file read**: As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.

3. **Execution/address calculation:** The signals to be set are RegDst, ALUOp, and ALUSrc The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
4. **Memory access:** The control lines set in this stage are Branch, MemRead, and MemWrite.These signals are set by the branch equal, load, and store instructions, respectively. Recall that PCSrc in Figure selects the next sequential address unless control asserts Branch and the ALU result was 0.

5. **Write-back:** The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

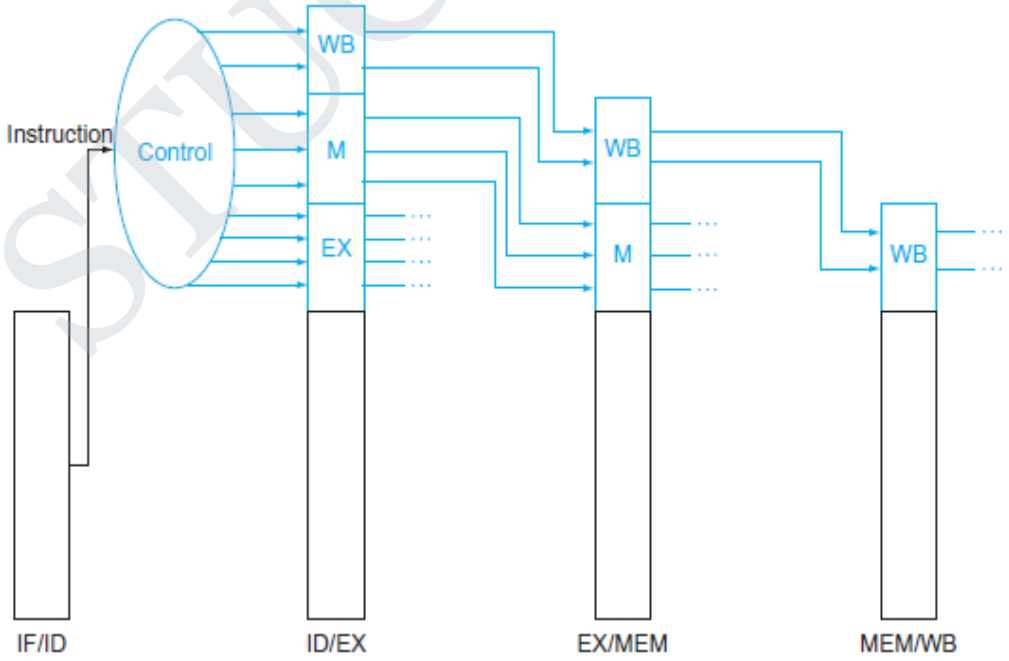


Figure 3.31 The control lines for the final three stages.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information. Since the control lines start with the EX stage, we can create the control information during instruction decode.

Figure 3.31 shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline

Figure 3.32 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage.

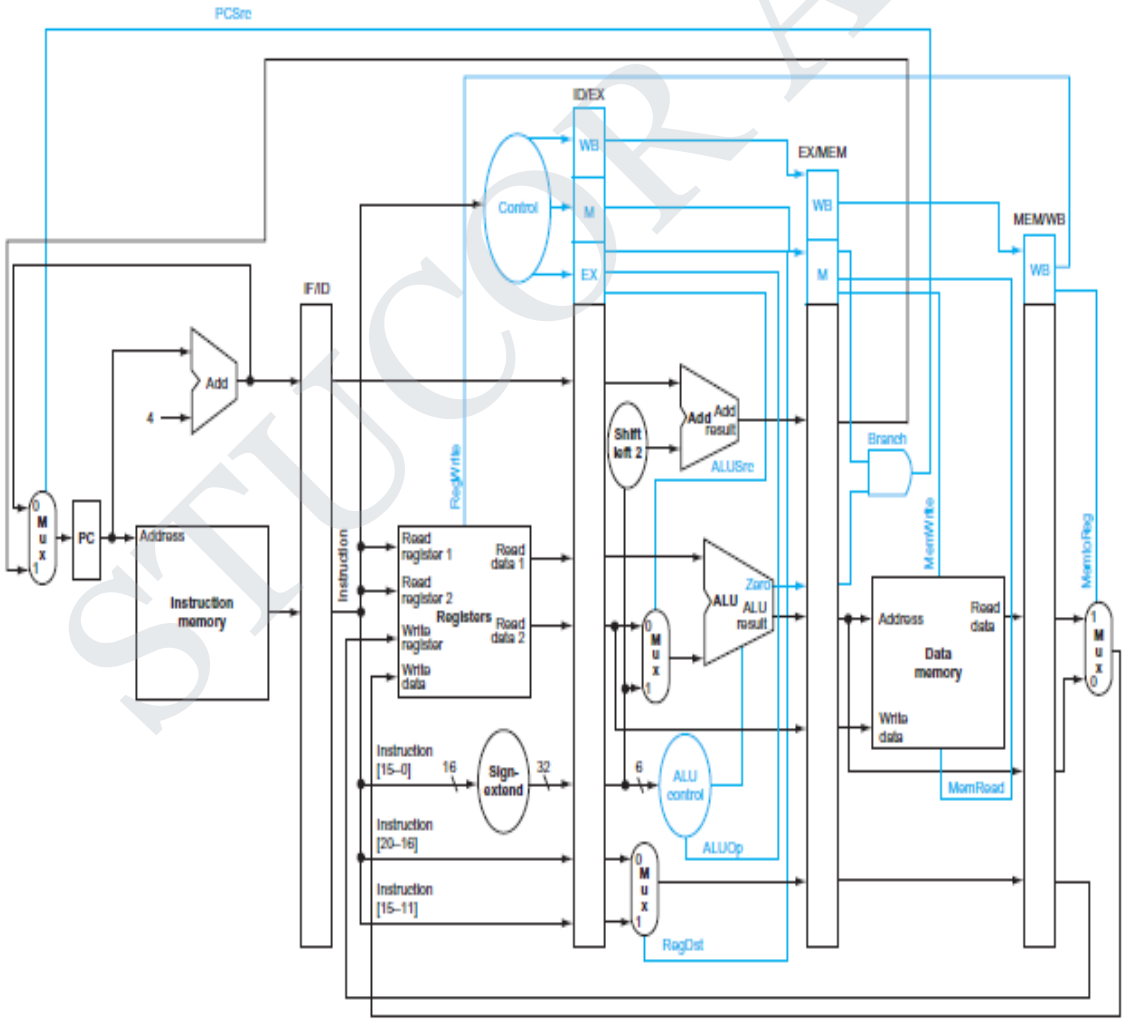


Figure 3.32 The pipelined datapath of with the control signals

5. HAZARDS

- **Explain the different types of pipeline hazards with suitable examples. (April/May 2015). (16)**
- **Explain how the instruction pipeline works? What are the various situations where an instruction pipeline can stall? Illustrate with an example. (Nov/Dec 2015). (16)**
- **Explain data hazard in detail (April/May 2014).(16)**
- **Discuss the methods to reduce hazards due ti conditional branches (April/May 2015). (16)**
- **What is Hazard? Explain its types with suitable examples. (Nov/Dec 2014)**
- **Briefly explain about various categories of hazards with examples.(May/June 2016)**

Situations in pipelining in which the next instruction cannot execute in the following clock cycle is called hazards, and there are three different types.

- **Structural Hazards**
- **Data Hazards**
- **Control Hazards**

STRUCTURAL HAZARDS

Structural hazard means when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

In MIPS instruction structural hazard will occur when the first instruction is accessing memory and the forth instruction is also accessing the same memory.

DATA HAZARDS:

Data hazards occur when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

In pipeline process if one step must wait for another to complete means it cause data hazards. Data Hazards mainly occur if one instruction depends on another instruction to complete their task. Also called a **pipeline data hazard**.

For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

```
add $s0,  $t0,  $t1
sub  $t2,  $s0,  $t3
```

The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.sub instruction wait for the of s0.
The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory.

Example for Forwarding with Two Instructions

```
add  $s0, $t0, $t1
sub   $t2, $s0, $t3
```

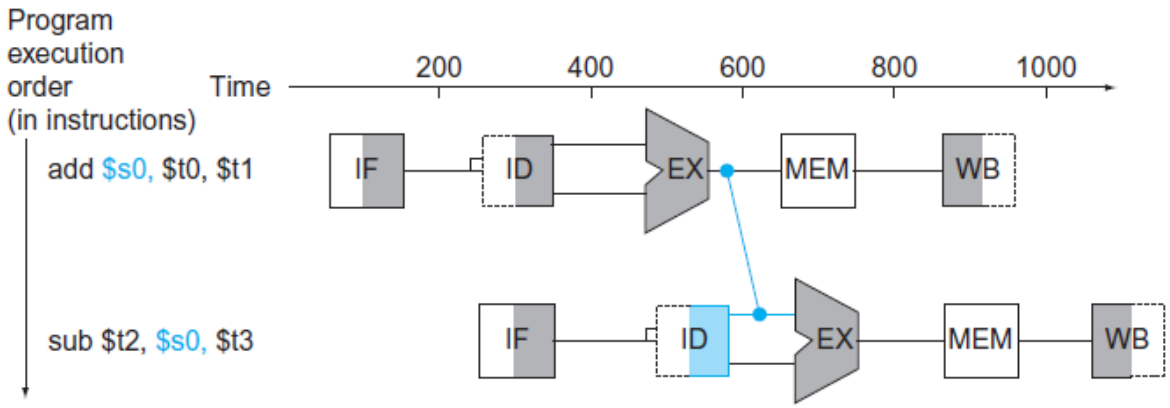


Figure 3.33 The Graphical Representation of Forwarding

The figure 3.33 shows the graphical representation for forwarding path from the output of EX stage of add to the input of EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Forwarding paths are valid only if the destination stage is later in time than the source stage.

Forwarding cannot prevent all pipeline stalls, so we need to focus new data hazard called **load-use data hazard** .The desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as

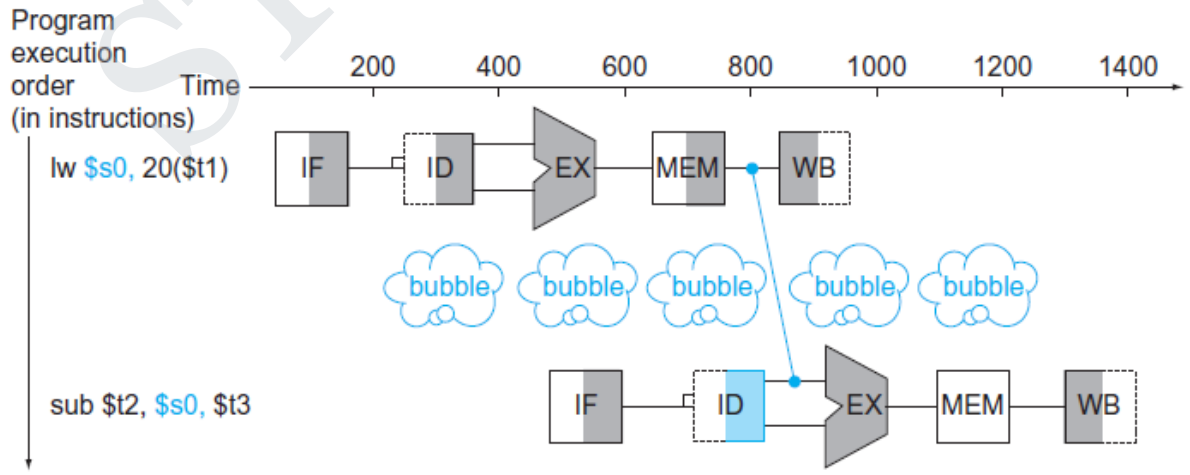


Figure 3.34 A Stall Even with Forwarding

The figure 3.34 shows an important pipeline concept, officially called a **pipeline stall**, but often given the nickname **bubble**. When reads and writes of data occur in a different order in the pipeline than in the program code.

There are three different types of data hazard (named according to the order of operations that must be maintained):

RAW:

A Read After Write hazard occurs when, in the code as written, one instruction reads a location after an earlier instruction writes new data to it, but in the pipeline the write occurs after the read (so the instruction doing the read gets stale data).

WAR :

A Write After Read hazard is the reverse of a RAW: in the code a write occurs after a read, but the pipeline causes write to happen first.

WAW:

A Write After Write hazard is a situation in which two writes occur out of order. For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Reordering Code to Avoid Pipeline Stalls:

To avoid pipeline stalls , the coding can be reordered.

For example consider the following code segment in C

```
a = b + e;
c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add $t3,   $t1,  $t2
sw   $t3, 12($t0)
lw    $t4, 8($t0)
add  $t5,  $t1,  $t4
sw   $t5, 16($t0)
```

Both **add** instructions have a hazard because of their respective dependence on the immediately preceding **lw** instruction. Notice that bypassing eliminates several other potential hazards, including the dependence of the first **add** on the first **lw** and any hazards for store instructions. Moving up the third **lw** instruction to become the third instruction eliminates both hazards:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

CONTROL HAZARDS

Control hazard, arising from the need to make a decision based on the results of one instruction while others are executing. Also called as **branch Hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the

instruction that was fetched is not the one that is needed.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner.

Consider the branch instruction we must begin fetching the instruction following the branch on the very next clock cycle. The pipeline cannot possibly know what the instruction should be. To avoid **STALL** we fetch a branch after that waiting until the pipelines determines the outcome of the branch and knows what instruction address to fetch from.

Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline.

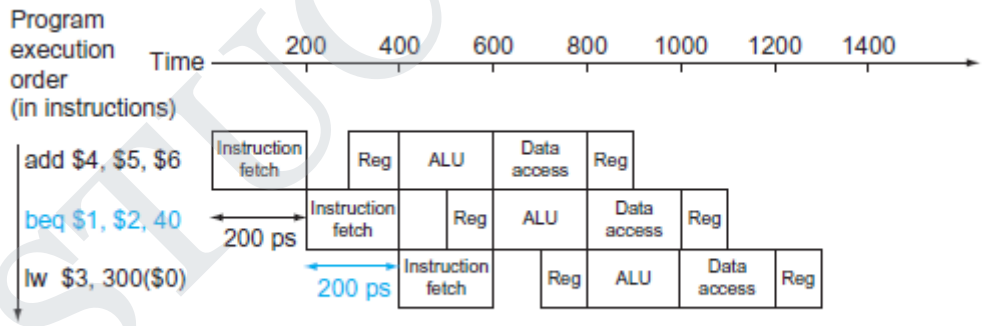


Figure 3.35 Predicting the branch not taken

Figure 3.35 shows the pipeline when a branch is not taken. The **lw** instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before starting.

If we cannot resolve the branch in the second stage, it cause longer pipelines, and it is too high cost for more computers.

Another method to resolve the control hazard is called **branch prediction**.

Branch prediction is a method of resolving a branch hazard that assume a given outcome for the branch and proceeds from that assumption rather that waiting to ascertain the actual outcome.

Figure 3.36 shows the pipeline when a branch is taken.

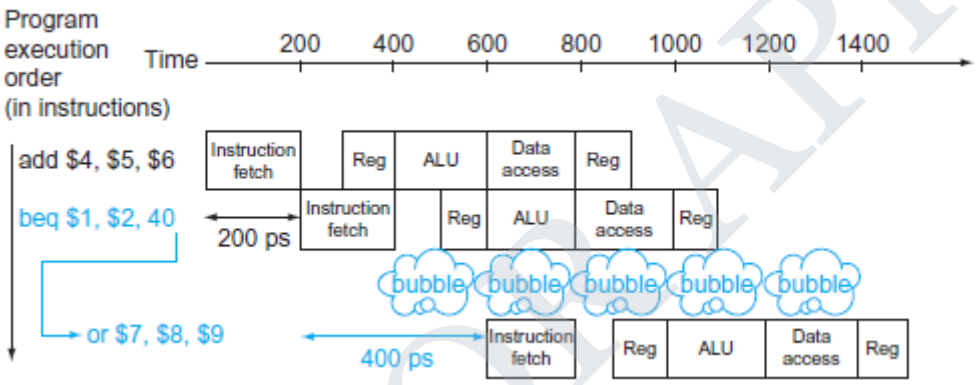


Figure 3.36 Predicting the branch taken

PART C

1.HANDLING DATA HAZARDS AND CONTROL HAZARDS

- **Describe the techniques for handling control hazards in pipelining (April/May 2013). (10)**
- **What is data hazard? How do you overcome it? (Nov/Dec 2011)(16)**

Data Hazards: Forwarding versus Stalling

Let us consider a sequence of instruction:

```
sub  $2,   $1, $3        # Register $2 written by sub
and  $12,$2, $5          # 1st operand($2) depends on sub
```

```
or   $13, $6, $2          # 2nd operand($2) depends on sub
add $14, $2, $2           # 1st($2) & 2nd($2) depend on sub
sw   $15,100($2)          # Base ($2) depends on sub
```

The last four instructions are all dependent on the result in register $2 of the first instruction. If register $2 had the value 10 before the subtract instruction and −20 afterwards, the programmer intends that −20 will be used in the following instructions that refer to register $2.

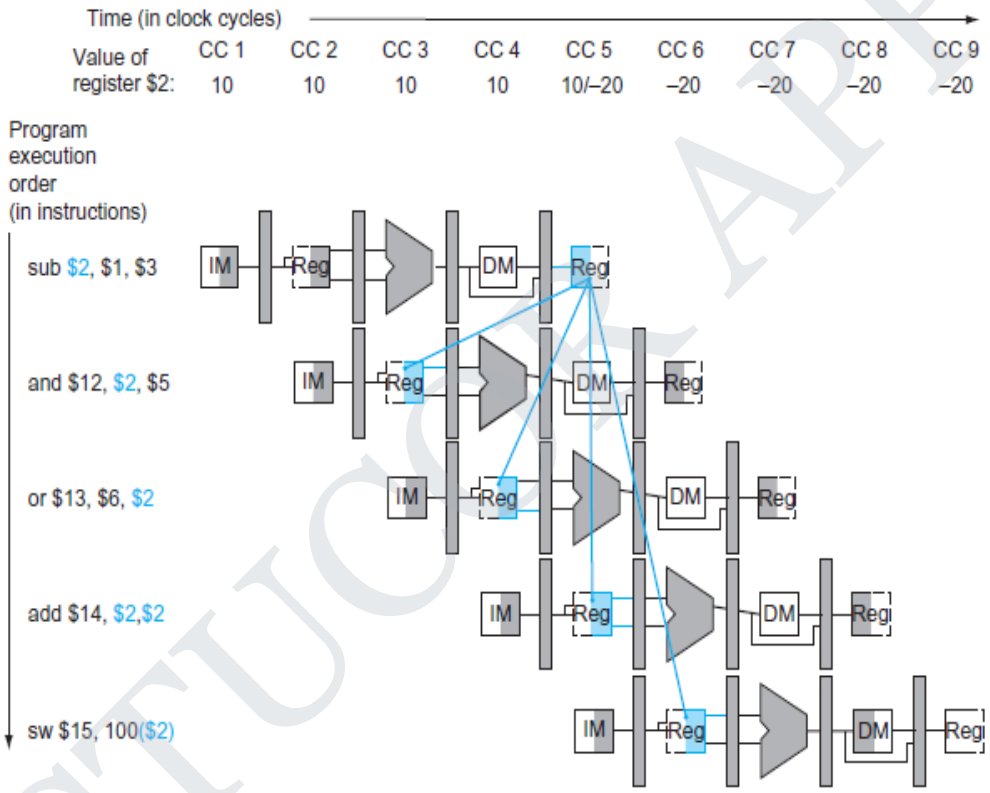


Figure 3.37 Pipelined dependences in a sequence using simplified datapath

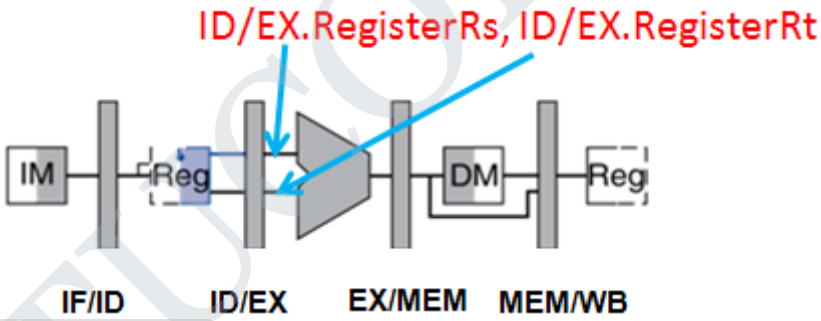
Figure 3.37 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of Figure shows the value of register $2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

The first instruction(Sub) will return the value of $2 and the remaining instructions will read the value from $2. The proper value of $2 will be available at the end of the

fifth clock cycle so remaining instruction must wait until fifth clock cycle it causes the **data hazard**. Data hazard can be resolved by using the design of the register file hardware. If the register is read and written in the same clock cycle means write is in first half of the clock cycle and read is in the second half of the clock cycle. We can execute this segment without stalls if we simply ***forward*** the data as soon as it is available to any units that need it before it is available to read from the register file.

Forwarding : It is applied only in the EX stage, which may be either an ALU operation or an effective address calculation. If EX stage wants to use the register value means it must write in the WB stage itself.

For example, "ID/EX.RegisterRs" refers to the number of one register whose value is found in the pipeline register ID/EX; that is, The first part of the name, is the name of the pipeline register; the second part is the name of the field in that register.



Using this notation, the two pairs of hazard conditions are

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

The first hazard in the sequence will occur in register $2, between the result of sub $2,$1,$3 and the first read operand of and $12,$2,$5. This hazard can be detected

when the and instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard

1a: EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

Because some instructions do not write registers, this policy is inaccurate; Check to see if the RegWrite signal will be active or not. Examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted.

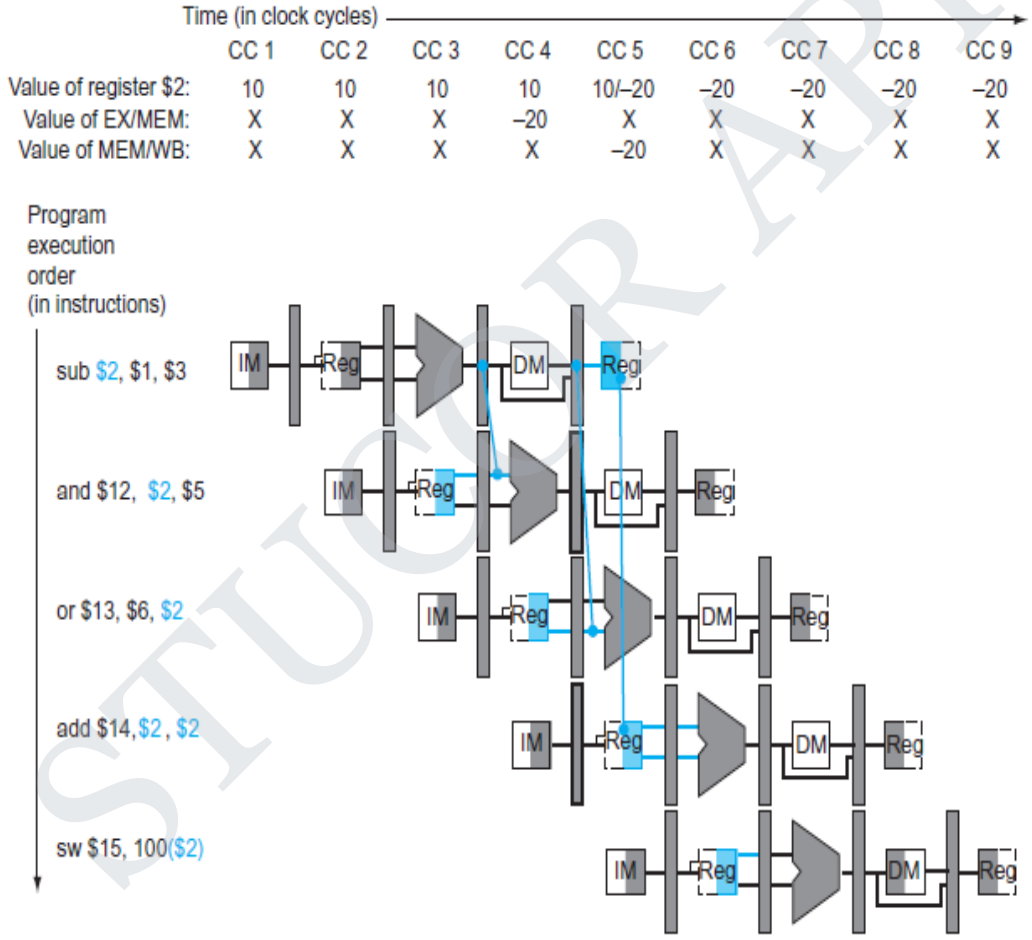


Figure 3.38 The dependences between the pipeline registers move forward

Figure 3.38 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence. The change is that the dependence begins from a p*ipeline* register, rather than waiting for the WB stage to write the register file.

Thus the pipeline registers holds the data to be forwarded. If we can take the inputs to the ALU from *any* pipeline register except ID/EX, then we can forward the proper data. By adding multiplexors to the input of the ALU, and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

Figure 3.39 shows a close-up of the ALU and pipeline register before and after adding forwarding. By adding multiplexers to the input of the ALU and with proper controls we can run the pipeline at full speed. This forwarding controls are in the Ex stage because the ALU forwarding multiplexers are found in that stage.

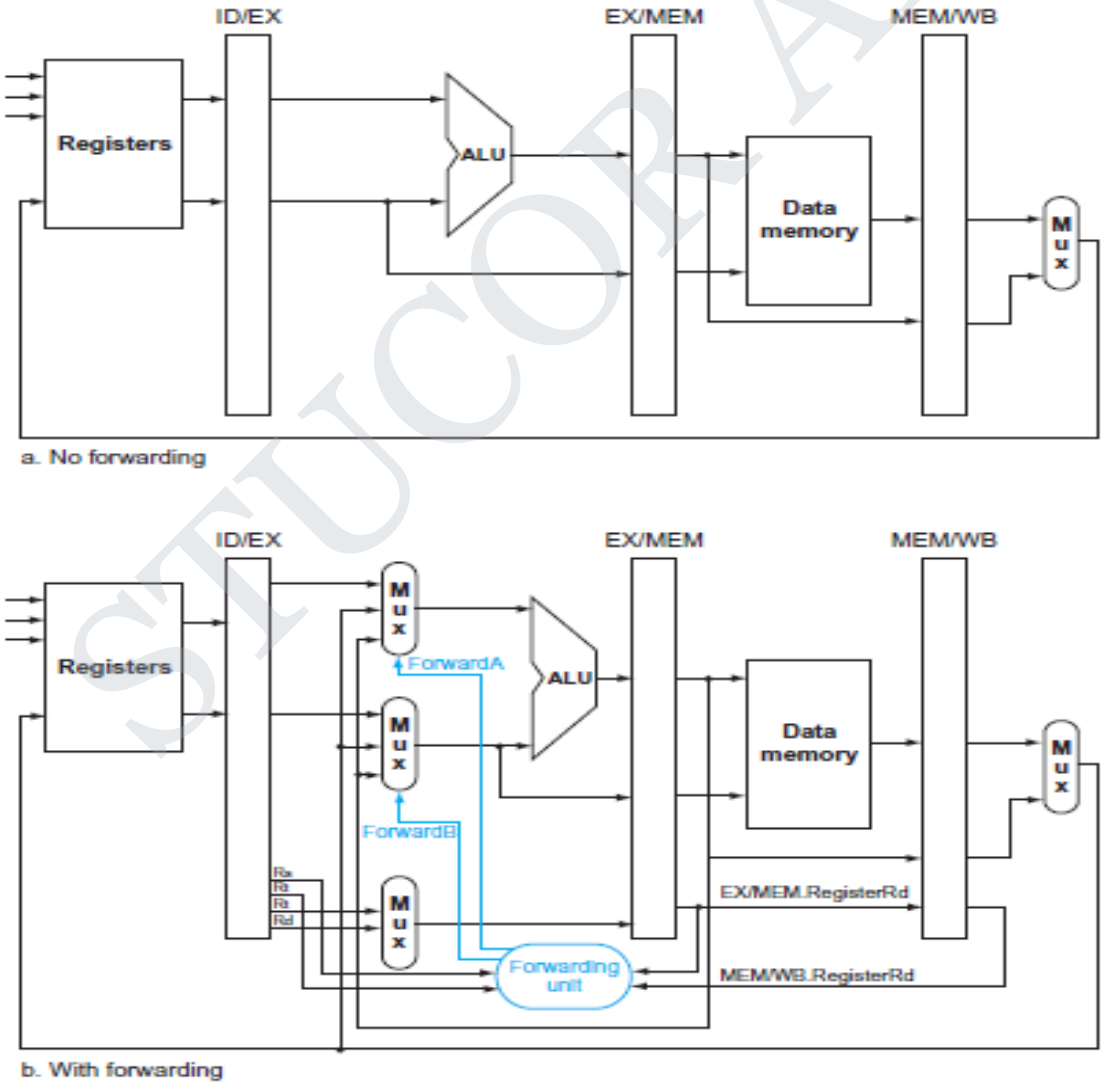


Figure 3.39 Pipeline Registers Forwarding

MUX control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Table shows a close-up of the ALU and pipeline register before and after adding forwarding. By adding multiplexers to the input of the ALU and with proper controls we can run the pipeline at full speed. This forwarding controls are in the EX stage because the ALU forwarding multiplexers are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. We have the rt field so before forwarding, the ID/EX register no need to include space to hold the rs field. Hence, rs is added to ID/EX.

Let's now write both the conditions for detecting hazards and the control signals to resolve them:

Condition 1. *EX hazard:*

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes

from the Rt field). This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

Condition 2. ***MEM hazard:***

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

There is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage.

For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

add $1,$1,$2
add $1,$1,$3
add $1,$1,$4

. . .

In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus, the control for the MEM hazard would be (with the additions highlighted):

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))

and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))

and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

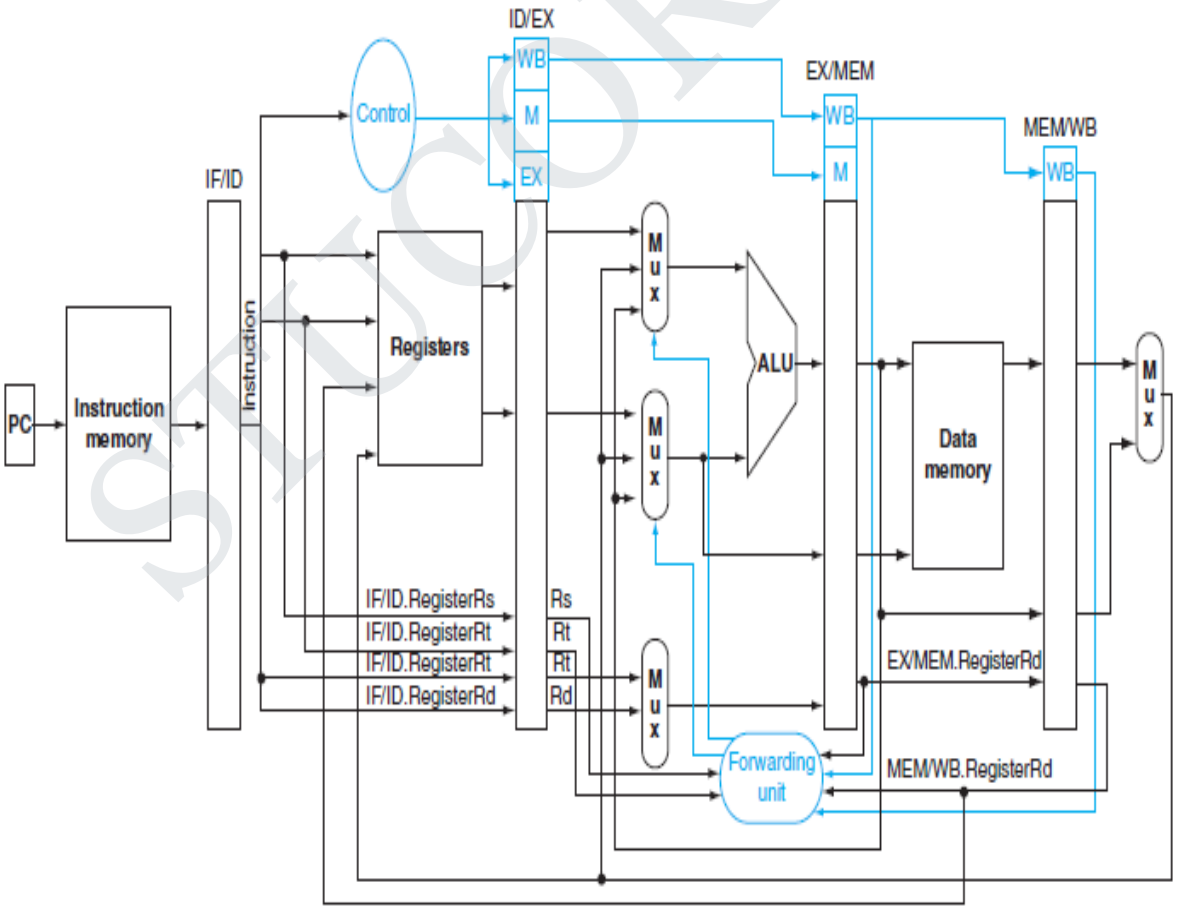


Figure 3.40 The datapath modified to resolve hazards via forwarding.

Figure 3.40 shows the hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

Data Hazards and Stalls

Forwarding method to resolve the problem occurred in data hazards. Consider the following set of instructions

lw $s2, 20($1)
and $4,$2,$5
or $8,$2,$6
add $9,$4,$2
slt $1,$6,$7

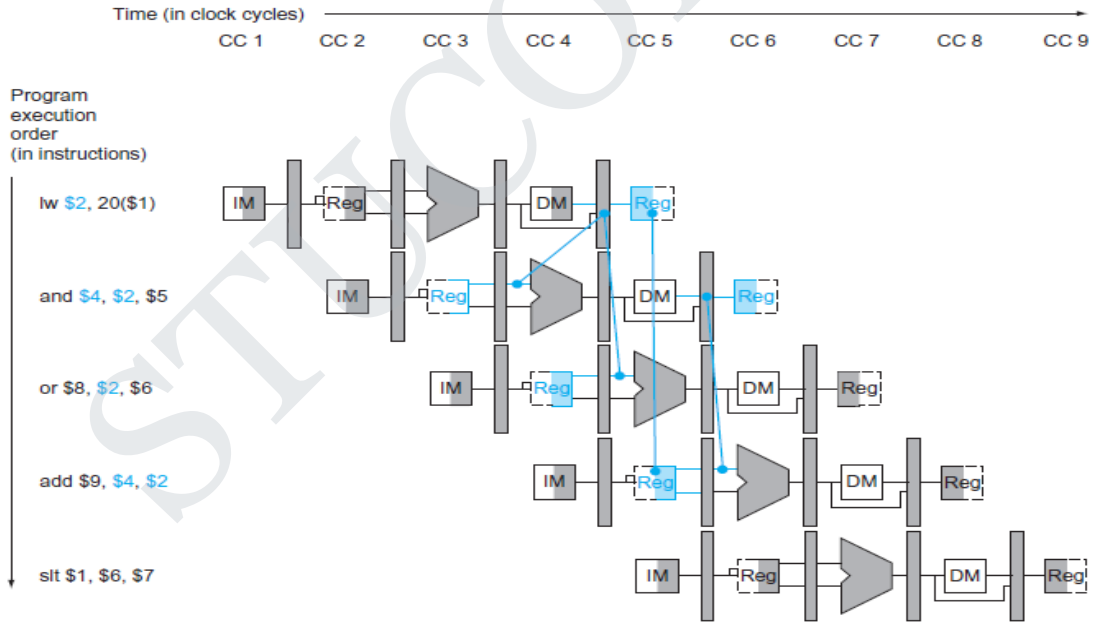


Figure 3.41 A Pipelined Sequence of Instructions.

Figure 3.41 illustrates the problem. Here dependence between the load and the following instruction goes backward in time this hazard cannot be solved by

forwarding technique. So in addition to a forwarding unit, we need a ***hazard detection unit***. It operates during the ID stage so that it can insert the stall between the load and its use.

Hazard detection unit has a single condition to provide control to load instructions

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)))
        stall the pipeline
```

- The **first line** tests to see if the instruction is a load or not. the only instruction that reads data memory is a load.
- The **next two lines** check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage.
- If the condition holds, the instruction **stalls** one clock cycle.

After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds.
If there were no forwarding, then the instructions would need another stall cycle.
If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled.

If IF stage not stalled means we lose the fetched instruction. To avoid stalled in these two stages we can prevent the PC registers and the IF/ID pipeline register from changing. The instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register.

NOP (No operation)

An instruction that does no operation to change state. NOP is a case where pipeline stage executing an instruction but that does not make any changes in the stage.. NOP acts like a bubble in the pipeline stage.

Inserting NOP's in pipeline

NOP act like bubbles, into the pipeline to identify hazard in ID stage. Once the bubble is inserted in the ID stage it will change the control field of EX, MEM, and WB field of the ID/EX pipeline register to 0. If the control values are 0 then no registers or memories are written.

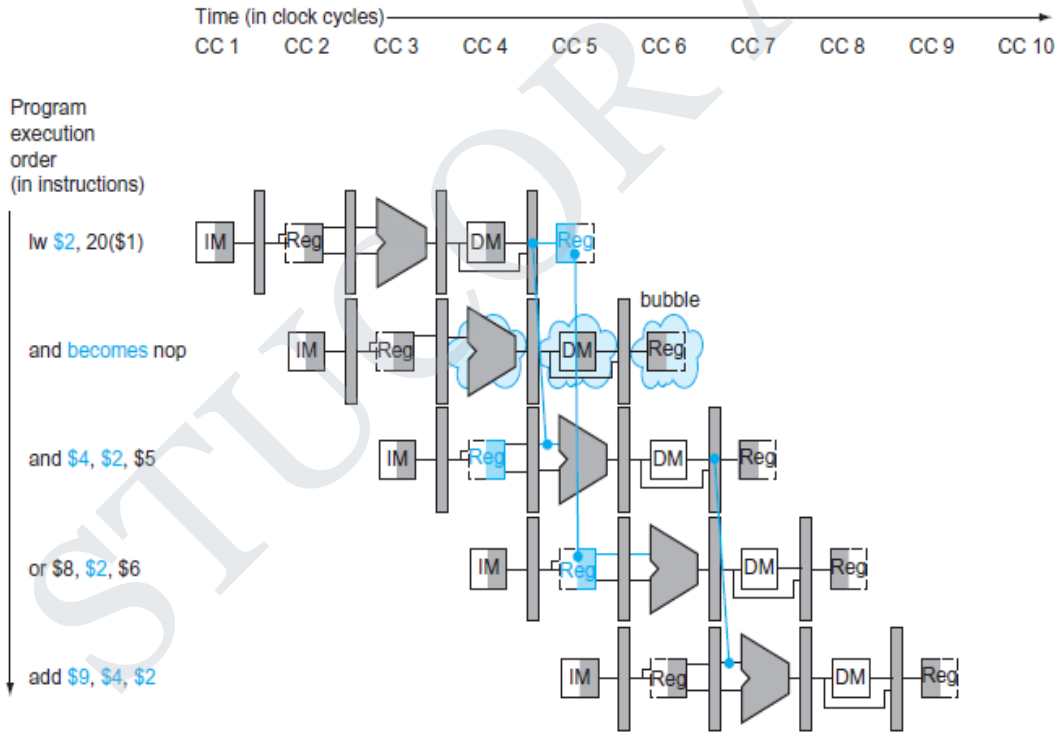


Figure 3.42 The way stalls are really inserted into the pipeline

Figure 3.42 shows the pipeline execution slot associated with the AND instruction is turned into a nop and all instructions beginning with the AND instruction are delayed one cycle.

In this example, the hazard forces the AND and OR instructions to repeat in clock cycle 4 what they did in clock cycle 3: AND reads registers and decodes, and OR is refetched from instruction memory. After inserting bubble all the dependencies go forward in time and no further hazards occur.

Hazard Detection Unit

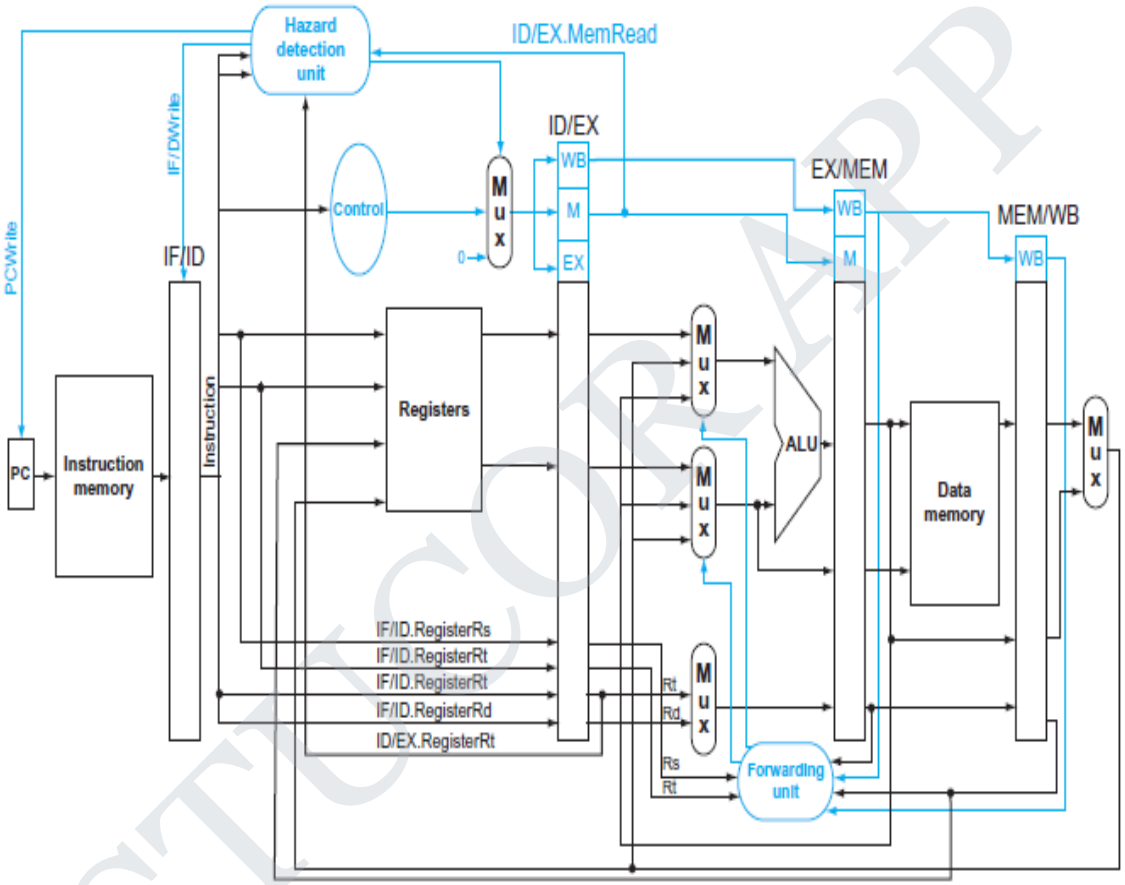


Figure 3.43 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit.

Figure 3.43 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s.

The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true

Control Hazards

It occurs when we execute the branch instruction in pipeline process. It occurs at less frequently.

For Example

40 beq $1, $3, 28
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)

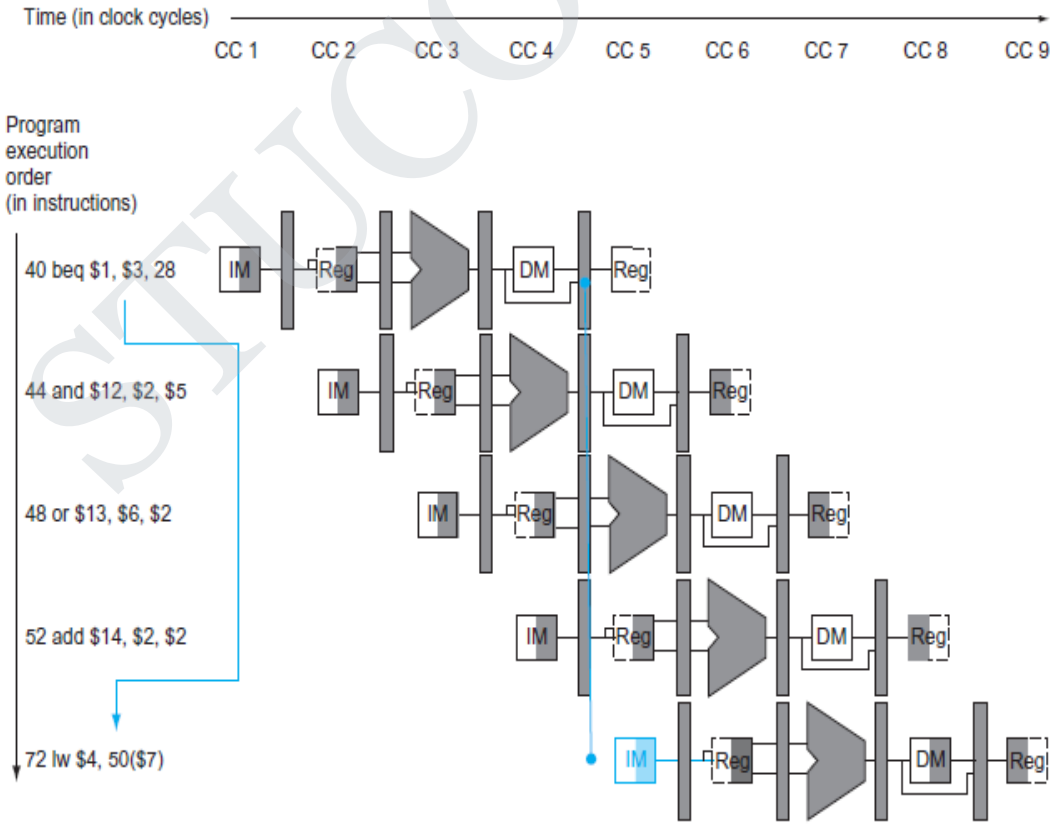


Figure 3.44 The impact of the pipeline on the branch instruction.

Figure 3.44 shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. The beq instruction is executed at MEM stage only.

Two schemes for resolving control hazards are.

- Branch not taken
- Branch Prediction

Branch Not Taken

If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. Discarding instructions, means we must be able to **flush** instructions in the IF, ID, and EX stages of the pipeline. If branch instruction is fetched immediately it stall the execution until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

If branches are not taken no need to discard the instructions the pipeline will execute the instructions continuously.

Reducing the Delay of Branches

Way to improve branch performance is to reduce the cost of the taken branch.The next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. Moving the **branch decision earlier** will increase the speed of the performance. Moving the **branch decision earlier** requires two actions to occur earlier:

- Computing the branch target address
- Evaluating the branch decision.

Computing the branch target address

We need the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

Evaluating the branch decision.

The branch decision is harder part. Branch decisions are **branch equal and branch not equal.**

For **branch equal**, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware. Branch dependent on a result still in the pipeline must still work properly with this optimization.

For example, to implement branch on equal (and its inverse), we will need to forward results to the equality test logic that operates during ID.

There are two complicating factors:

1. During ID stage we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.

2. The values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed.

To overcome these difficulties we can move the branch execution to the ID stage because it reduces the penalty of a branch to only one instruction if the branch is taken.

Branch prediction

Branch prediction is a method of resolving a branch hazard that assume a given outcome for the branch and proceeds from that assumption rather that waiting to ascertain the actual outcome. A simple form of branch prediction is we have to assume branch is not taken. This assumption is possible only for simple five stage pipeline. For deeper pipelines this assumption is not suitable because it will increase the branch penalty when measured in clock cycles. For such kind of pipelines we have to add more hardware to predict branch behavior during program execution. Solution for increasing branch penalty we have new technique called **dynamic branch prediction**.

Dynamic Branch Prediction

One implementation of that approach is a **branch prediction buffer** or **branch history table**. **A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction**. The memory contains a bit that says whether the branch was recently taken or not.

Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

Drawback

The accuracy of the predictor match the taken branch frequency for higher regular branches.

Two bit prediction scheme

To overcome this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed.

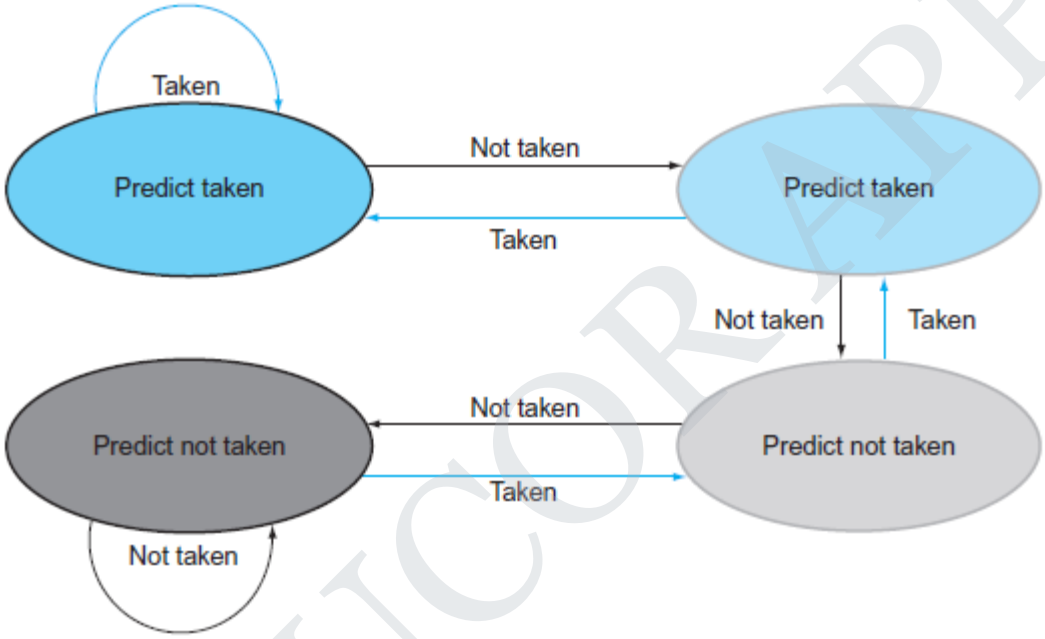


Figure 3.45 The states in a 2-bit prediction scheme.

Figure 3.45 shows the finite-state machine for a 2-bit prediction scheme. A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure

2. Explain in detail how exceptions are handled in MIPS architecture. (April/May 2015).

Exception an unscheduled event that disrupts program execution and used to

detect overflow. It is also called an interrupt. **Interrupt** is an exception that comes from outside of the processor.

Distinguishing interrupts and exception is more important. Exception refers to any unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term interrupt only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated

Type of event	From Where?	MIPS Terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

To detect exception we need to implement control. Two kinds of exception arise

- From the portion of instruction set
- During implementation

First we have to **detect** the condition of exception. Next we need to take appropriate **action** for it. Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce the performance, as well as complicate the task of getting the design correct.

Exceptions Handling in the MIPS Architecture

In MIPS architecture two types of exceptions occur when

- Executing an undefined instruction
- An arithmetic overflow occur.

Once exception occurs the processor must **save** the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address. The operating system can then take the appropriate action. The actions are

- Taking some predefined action in response to an overflow
- Stopping the execution of the program and reporting an error.

After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it.

There are two main methods used to communicate the reason for an exception.

- To include a **status register** (called the Cause register), which holds a field that indicates the reason for the exception.
- To use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined Instruction	$8000\ 0000_{hex}$
Arithmetic Overflow	$8000\ 0180_{hex}$

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause. We will need to add two additional registers to the MIPS implementation:

EPC: A 32bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)

Cause Register: A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an undefined instruction and 12 representing arithmetic overflow.

Exceptions in a Pipelined Implementation

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. We must flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address.

To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is ORed with the stall signal from the hazard Detection unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called EX. Flush to cause new multiplexors to zero the control lines. To start fetching instructions from location 8000

0180hex, which is the MIPS exception address, we simply add an additional input to the PC multiplexor that sends 8000 0180hex to the PC.

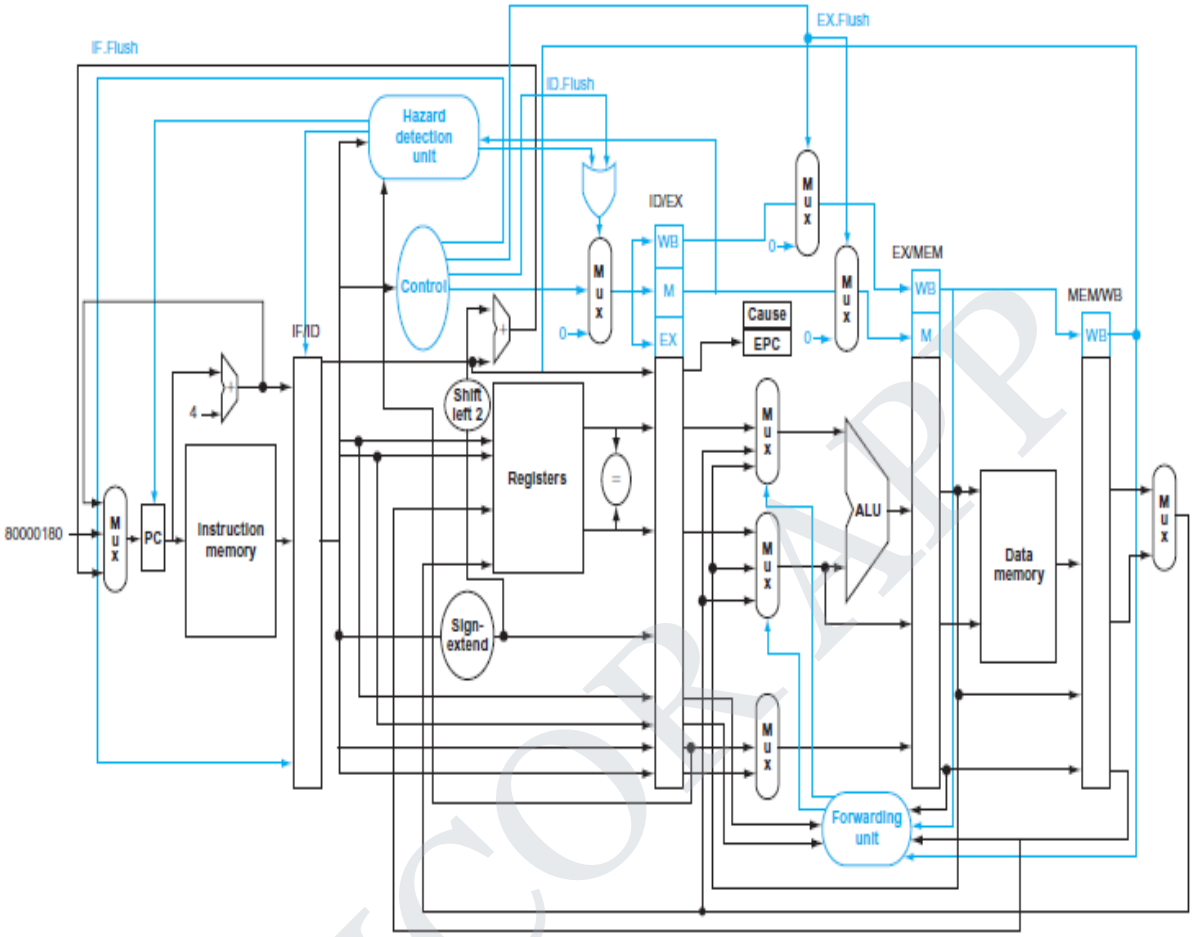


Figure 3.46 The datapath with controls to handle exceptions.

Figure 3.46 shows these changes. This example points out a problem with exceptions: if we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register $1 that helped cause the overflow because it will be clobbered as the Destination register of the add instruction.

Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally. The

easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled.

The final step is to save the address of the off ending instruction in the *exception program counter* (EPC). In reality, we save the address +4, so the exception handling the soft ware routine must first subtract 4 from the saved value.

UNIT IV

UNIT IV PARALLELISM 9

Instruction-level-parallelism – Parallel processing challenges – Flynn's classification – Hardware multithreading – Multicore processors

PART A

1. Define speculation (NOV/DEC 14)

It is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier

2. Differentiate between strong scaling and week scaling. (APR/MAY 15)

Strong Scaling : Speed up achieved on a multiprocessor without increasing the size of the problem is called strong scaling.

Weak Scaling : Speedup achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processor is called weak scaling.

3. What is Flynn's Classification? (Nov/Dec 2014).

Flynn uses the stream concept for describing a machine's structure. A stream simply means a sequence of items (data or instructions). The classification of computer architectures based on the number of instruction steams and data streams is called Flynn's Taxonomy.

SISD : Single Instruction Single Data

MISD : Multiple Instructions Single Data

SISD : Singe-Instruction stream, Singe Data stream

SIMD : Single-Instruction stream, Multiple Data streams

4. **What is meant by hardware multithreading? (Nov/Dec 2014).**

Hardware multithreading : Increases the utilization of a processor by switching to another thread when one thread is stalled. It allows multiple threads to share the functional units of a single processor in an overlapping fashion to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread.

5. What is ILP?(Nov/Dec 2015)

Pipelining exploits the potential parallelism among instructions. This parallelism is called instruction-level parallelism (ILP). There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage.

6. Define loop unrolling

An important compiler technique to get more performance from loops is **loop unrolling**, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

7. What is Anti-dependence?

It is an ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

8. Define – Static Multiple Issue, Issue Slots and Issue Packet.

Static multiple issue is an approach to implement a multiple-issue processor where many decisions are made by the compiler before execution.

Issue slots are the positions from which instructions could be issued in a given clock cycle. By analogy, these correspond to positions at the starting blocks for a sprint.

Issue packet is the set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

9. Define – Superscalar Processor and VLIW.(Nov/Dec 2016)

Superscalar is an advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution. Dynamic multiple-issue processors are also known as superscalar processors.
Very Long Instruction Word (VLIW) is a style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

10. Compare UMA and NUMA multiprocessors. (Apr/May 15)

- **Uniform Memory Access (UMA) :** Latency to a word in memory does not depend on which processor asks for it. Such machines are called **Uniform Memory Access multiprocessors.** That is the time taken to access a word is uniform for all processors.

- **Non – Uniform Memory Access (NUMA) :** Some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different

microprocessors or to different memory controllers on the same chip. Such machines are called **nonuniform memory access (NUMA)** multiprocessors.

- Programming challenges are harder for NUMA than UMA.
- NUMA machines can scale to larger size.
- NUMA s can have lower latency to nearby memory.

PART B

1. INSTRUCTION LEVEL PROCESSING

❖ **Explain in detail about Instruction Level Processing. (Nov/Dec 2014)**

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism.

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel. The amount of parallelism available within a basic block (a straight-line code sequence with no branches in and out except for entry and exit) is quite small. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called **loop-level parallelism**.

Example 1

```
for (i=1; i<=1000; i= i+1)
        x[i] = x[i] + y[i];
```

This is a parallel loop. Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap.

Example 2

```
for (i=1; i<=100; i= i+1)
{
        a[i] = a[i] + b[i];         //s1
        b[i+1] = c[i] + d[i];       //s2
}
```

Is this loop parallel? If not how to make it parallel?

Statement s1 uses the value assigned in the previous iteration by statement s2, so there is a **loop-carried dependency** between s1 and s2. Despite this dependency, this loop can be made parallel because the dependency is not circular:

- Neither statement depends on itself;
- While s1 depends on s2, s2 does not depend on s1.

A loop is parallel unless there is a cycle in the dependencies, since the absence of a cycle means that the dependencies give a partial ordering on the statements. To expose the parallelism the loop must be transformed to conform to the partial order. Two observations are critical to this transformation: There is no dependency from s1 to s2. Then, interchanging the two statements will not affect the execution of s2.

On the first iteration of the loop, statement s1 depends on the value of b[1] computed prior to initiating the loop. This allows to replace the loop above with the following code sequence, which makes possible overlapping of the iterations of the loop:

```
a[1] = a[1] + b[1];
for (i=1; i<=99; i= i+1)
{
        b[i+1] = c[i] + d[i];
        a[i+1] = a[i+1] + b[i+1];
}
b[101] = c[100] + d[100];
```

Example 3

```
for (i=1; i<=100; i= i+1)
{
        a[i+1] = a[i] + c[i];       //S1
        b[i+1] = b[i] + a[i+1];     //S2
}
```

This loop is not parallel because it has cycles in the dependencies, namely the statements S1 and S2 depend on themselves.

Parallelism and Advanced Instruction-Level Parallelism:

Pipelining exploits the potential parallelism among instructions. This parallelism is called **instruction-level parallelism** (ILP). There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple**

issue. It is a scheme whereby multiple instructions are launched in one clock cycle Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle. There are many constraints on what types of instructions may be executed simultaneously, and what happens when dependences arise.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue and dynamic multiple issue**.

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

1.**Packaging instructions into issue slots**: In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will oft en have already tried to help improve the issue rate by placing the instructions in a beneficial order.

2.**Dealing with data and control hazards**: In static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

2. CHALLENGES OF PARALLEL PROCESSING

❖ **State the challenges of Parallel Processing.(Nov/Dec 2014)**

Static Multiple Issue

Static multiple-issue processors use the compiler to assist with packaging

instructions and handling hazards. In a static issue processor, the set of instructions issued in a given clock cycle, which is called an issue packet, is one large instruction with multiple operations. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This led to the original name for this approach: **Very Long Instruction Word** (VLIW).

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards.

Consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue.

Instruction Type	**Pipe Stages**							
ALU or Branch Instruction	IF	ID	EX	MEM	WB			
Load or Store Instruction	IF	ID	EX	MEM	WB			
ALU or Branch Instruction		IF	ID	EX	MEM	WB		
Load or Store Instruction		IF	ID	EX	MEM	WB		
ALU or Branch Instruction			IF	ID	EX	MEM	WB	
Load or Store Instruction			IF	ID	EX	MEM	WB	
ALU or Branch Instruction				IF	ID	EX	MEM	WB
Load or Store Instruction				IF	ID	EX	MEM	WB

Figure 4.1 An Example: Static Multiple Issue with the MIPS

Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a nop. Thus, the instructions always issue in pairs, possibly with a nop in one slot. Figure 4.1 shows how the instructions look as they go into the pipeline in pairs.

Dynamic Multiple-Issue Processors:

Dynamic multiple-issue processors are also known as superscalar processors, or simply Superscalar. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor.

Many superscalars extend the basic framework of dynamic issue decisions to include dynamic pipeline scheduling. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
lw $t0, 20($s2)
addu $t1, $t0, $t2
sub $s4, $s4, $t3
slti $t5, $s4, 20
```

Even though the sub instruction is ready to execute, it must wait for the lw and addu to complete first, which might take many clock cycles if memory is slow. Dynamic pipeline scheduling allows such hazards to be avoided either fully or partially.

Dynamic Pipeline Scheduling:

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (a dozen or more in high-end designs in 2008), and a commit unit. Figure shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called reservation stations, which hold the operands and the operation.

When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the reorder buffer, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

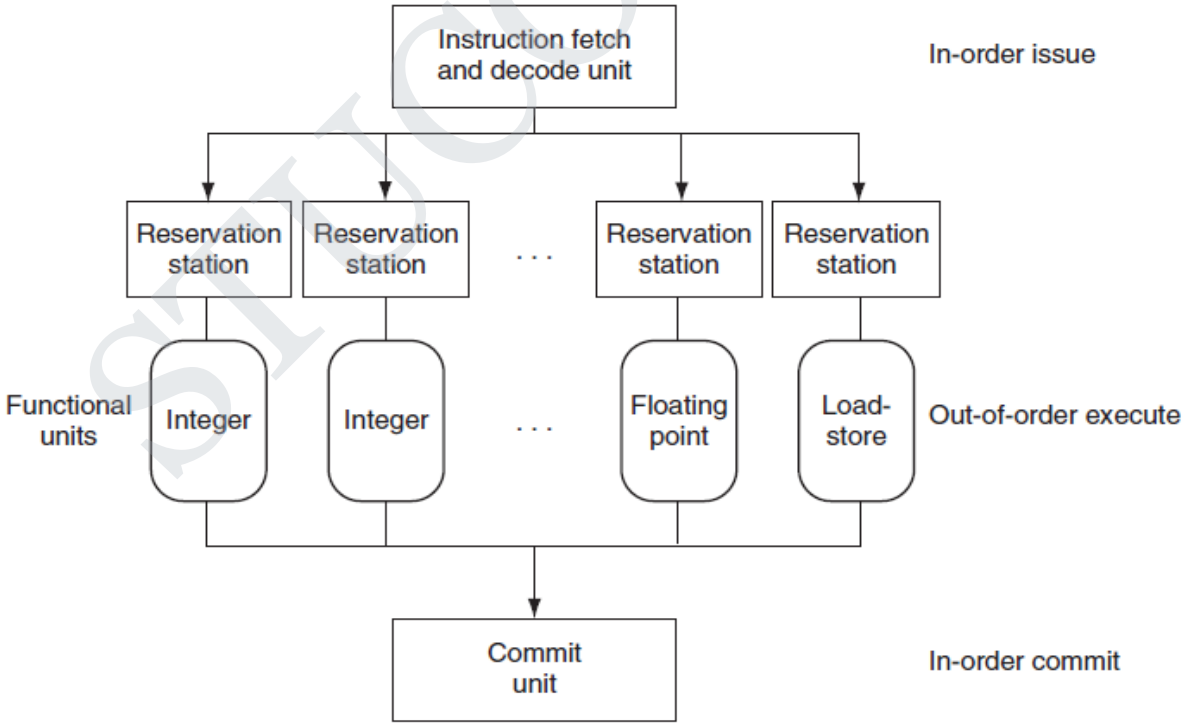


Figure 4.2 The three primary units of a dynamically scheduled pipeline

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called in-order commit.

There are at least two different kinds of parallelism in computing.

- Using multiple processors to work toward a given goal, with each processor running its own program.
- Using only a single processor to run a single program, but allowing instructions from that program to execute in parallel.

The latter is called *instruction-level parallelism*, or ILP.

Limitations of ILP:

The Hardware Model

An ideal processor is one where all constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows through either registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. Register Renaming:

There are an infinite number of virtual registers available, and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

2. Branch Prediction:

Branch prediction is perfect. All conditional branches are predicted exactly.

3. Jump Prediction:

All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

4. Memory Address Alias Analysis:

All memory addresses are known exactly, and a load can be moved before a store provided that the addresses are not identical. Note that this implements perfect address alias analysis.

5. Perfect Caches:

All memory accesses take 1 clock cycle. In practice, superscalar processors will typically consume large amounts of ILP hiding cache misses, making these results highly optimistic. To measure the available parallelism, a set of programs was compiled and optimized with the standard MIPS optimizing compilers. The programs were instrumented and executed to produce a trace of the instruction and data references. Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences. Since a trace is used, perfect branch prediction and perfect alias analysis are easy to do.

Limitations on the Window Size and Maximum Issue Count

To build a processor that even comes close to perfect branch prediction and perfect alias analysis requires extensive dynamic analysis, since static compile time

schemes cannot be perfect. Of course, most realistic dynamic schemes will not be perfect, but the use of dynamic schemes will provide the ability to uncover parallelism that cannot be analyzed by static compile time analysis. Thus, a dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor.

The Effects of Realistic Branch and Jump Prediction:

The processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed! Of course, no real processor can ever achieve this. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates.

a. Perfect : All branches and jumps are perfectly predicted at the start of execution.

b. Tournament-based branch predictor: The prediction scheme uses a correlating 2-bit predictor and a noncorrelating 2-bit predictor together with a selector, which chooses the best predictor for each branch.

The Effects of Finite Registers:

The ideal processor eliminates all name dependences among register references using an infinite set of virtual registers. To date, the IBM Power5 has provided the largest numbers of virtual registers: 88 additional floating-point and 88 additional integer registers, in addition to the 64 registers available in the base architecture. All 240 registers are shared by two threads when executing in multithreading mode, and all are available to a single thread when in single-thread mode.

The Effects of Imperfect Alias Analysis:

The optimal model assumes that it can perfectly analyze all memory dependences, as well as eliminate all register name dependences. Of course, perfect alias analysis is not possible in practice: The analysis cannot be perfect at compile time, and it requires a potentially unbounded number of comparisons at run time (since the number of simultaneous memory references is unconstrained).

3. FLYNN'S CLASSIFICATION

❖ **Discuss about SISD,MIMD,SIMD,SPMD and Vector systems. (April/May 2015, Nov/Dec 2015, May/June 2016) (16)**

Flynn uses the stream concept for describing a machine's structure. A stream simply means a sequence of items (data or instructions). The classification of computer architectures based on the number of instruction steams and data streams (Flynn's Taxonomy).

SISD (Singe-Instruction stream, Singe-Data stream):

SISD corresponds to the traditional mono-processor (von Neumann computer). A single data stream is being processed by one instruction stream .A single-processor computer (uni-processor) in which a single stream of instructions is generated from the program.

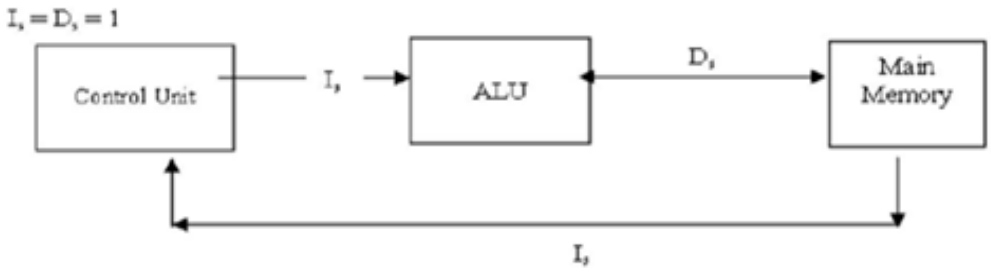


Figure 4.3 SISD

SIMD (Single-Instruction stream, Multiple-Data streams):

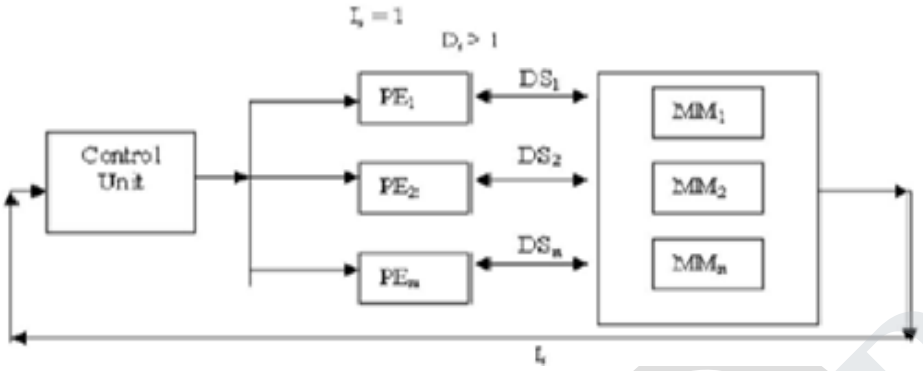


Figure 4.4 SIMD

Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams. This group is dedicated to array processing machines. Sometimes, vector processors can also be seen as a part of this group.

MISD (Multiple-Instruction streams, Singe-Data stream):

Each processor executes a different sequence of instructions. In case of MISD computers, multiple processing units operate on one single-data stream. In practice, this kind of organization has never been used.

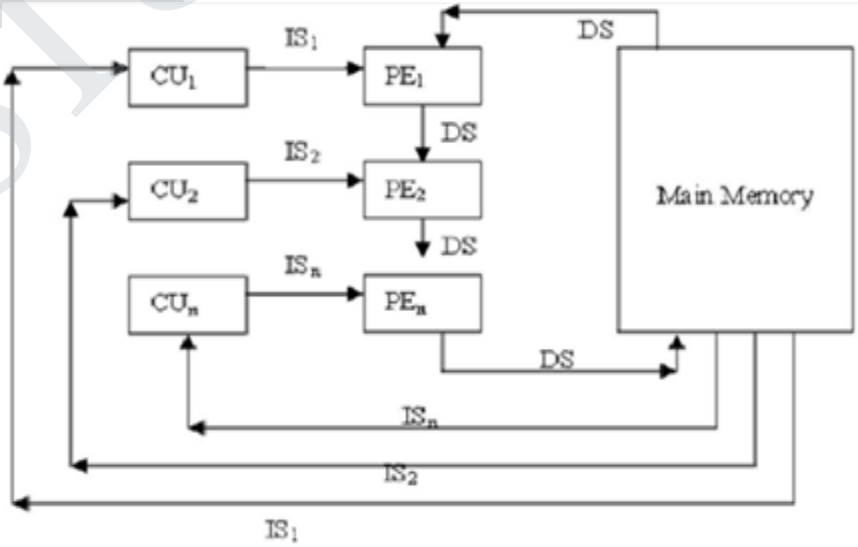


Figure 4.5 MIMD

MIMD(Multiple-Instruction streams, Multiple-Data streams):

Each processor has a separate program. An instruction stream is generated from each program. Each instruction operates on different data. This last machine type builds the group for the traditional multi-processors. Several processing units operate on multiple-data streams.

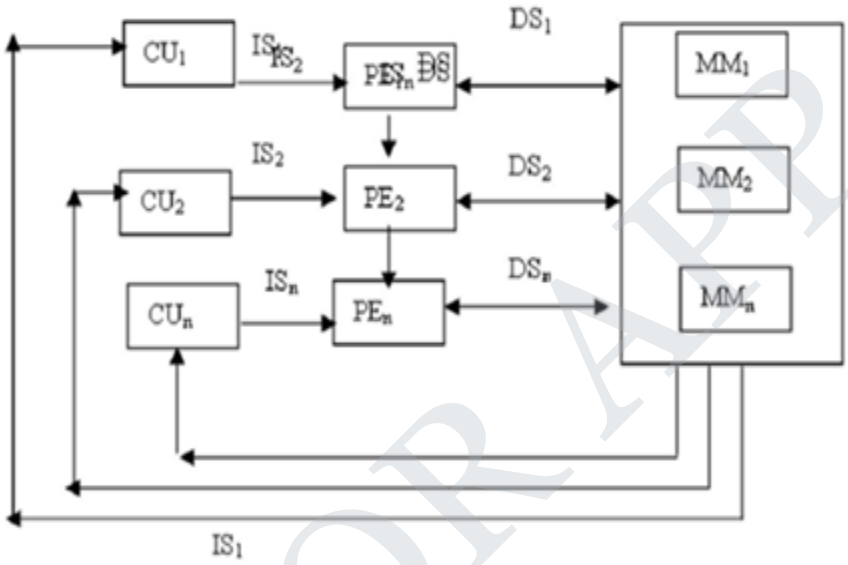


Figure 4.6 MIMD

SISD, MIMD, SIMD, SPMD, and Vector:

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

Based on the number of instruction streams and the number of data streams shown in above table. The conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated SISD and MIMD, respectively.

While it is possible to write separate programs that run on different processors on a MIMD computer and yet work together for a grander, coordinated goal, programmers normally write a single program that runs on all processors of an MIMD computer, relying on conditional statements when different processors should execute different sections of code. This style is called **Single Program Multiple Data (SPMD)**, but it is just the normal way to program a MIMD computer.

While it is hard to provide examples of useful computers that would be classified as multiple instruction streams and single data stream (MISD), the inverse makes much more sense. SIMD computers operate on vectors of data. For example, a single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced size of program memory—SIMD needs only one copy of the code that is being simultaneously executed, while message-passing MIMDs may need a copy in every processor, and shared memory MIMD will need multiple instruction caches.

SIMD works best when dealing with arrays in for loops. Hence, for parallelism to work in SIMD, there must be a great deal of identically structured data, which is called data-level parallelism. SIMD is at its weakest in case or switch statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data are disabled so that units with proper data may continue. Such situations essentially run at 1/nth performance, where n are the number of cases.

SIMD in x86: Multimedia Extensions:

The most widely used variation of SIMD is found in almost every microprocessor today, and is the basis of the hundreds of MMX and SSE instructions

of the x86 microprocessor (see Chapter 2). They were added to improve performance of multimedia programs. These instructions allow the hardware to have many ALUs operate simultaneously or, equivalently, to partition a single, wide ALU into many parallel smaller ALUs that operate simultaneously.

Vector:

An older and more elegant interpretation of SIMD is called a vector architecture, which has been closely identified with Cray Computers. It is again a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. A key feature of vector architectures is a set of vector registers. Thus, vector architecture might have 32 vector registers, each with 64 64-bit elements.

Vector versus Scalar:

Vector instructions have several important properties compared to conventional instruction set architectures, which are called scalar architectures in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.

- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.

- Vector architectures and compilers have a reputation of making it much easier than MIMD multiprocessors to write efficient applications when

they contain data-level parallelism.

- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save power as well.

- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.

- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.

- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

4. HARDWARE MULTITHREADING

- **Explain about Hardware Multithreading.(Nov/Dec 2014)**
- **What is hardware multithreading? Compare and contrast Fine grained multi threading and coarse grained multi threading.(April/May 2015, Nov/Dec 2015, May/June 2016)**

Multithreading is defined as the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests

by the same user without having to have multiple copies of the programming, running in the computer or console.

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.

A **thread** includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

A **process** includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

Increasing utilization of a processor by switching to another thread when one thread is stalled is called **Hardware multithreading**.

There are two main approaches to hardware multithreading.

- Fine Grained multithreading
- Coarse-grained multithreading

Fine-grained multithreading switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.

Advantage of fine-grained multithreading:

It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls.

Disadvantage of fine-grained multithreading:

It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

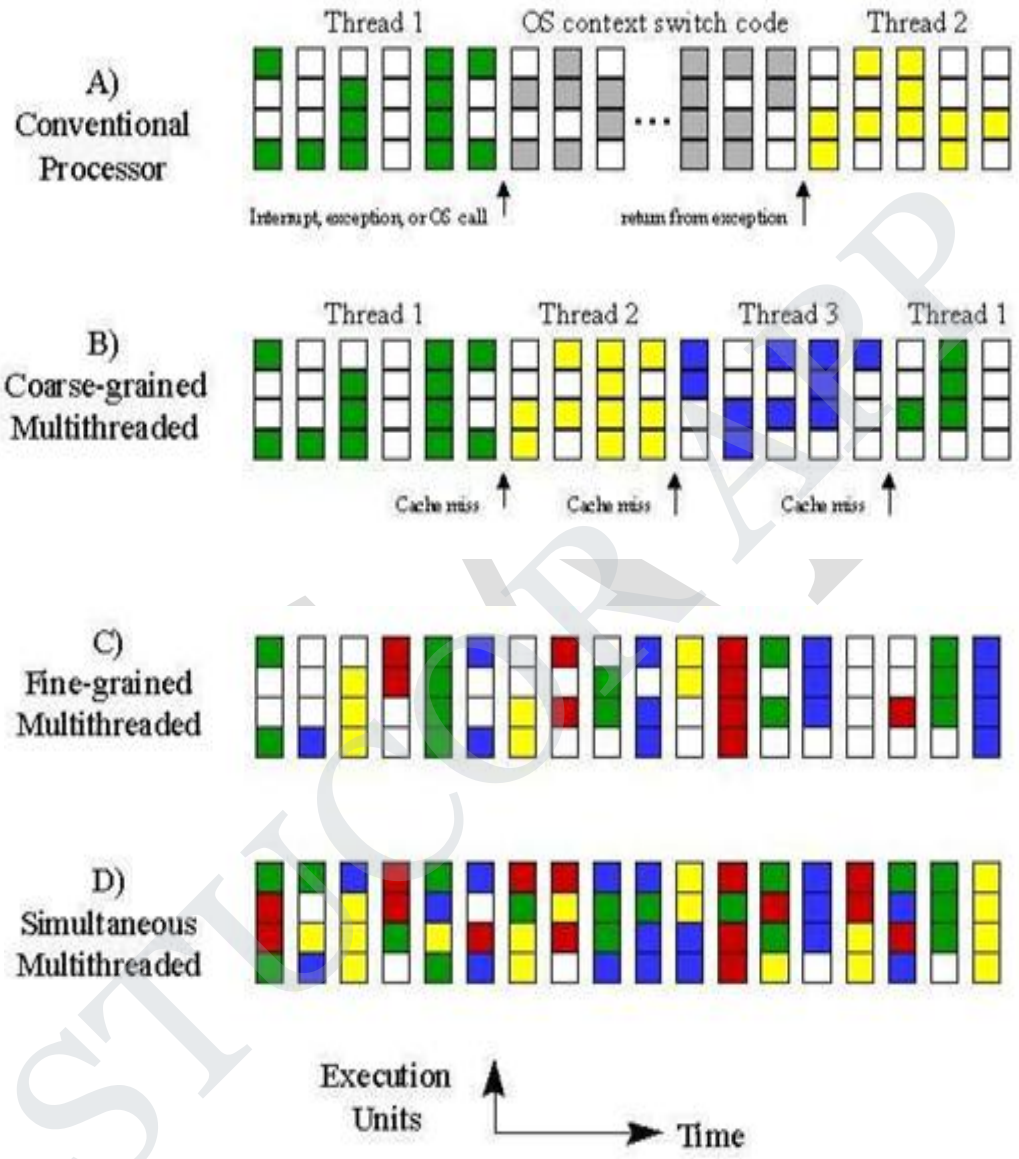


Figure 4.7 Multithreading

Coarse-Grained Multithreading (CGMT):

Coarse-grained multithreading switches threads only on costly stalls, such as last-level cache misses.

Advantages:

- Relieves need to have very fast thread switching.
- Doesn't slow down thread, since instructions from other threads.
- Threads issued only when the thread encounters a costly stall.

Disadvantages:

- It is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs
- Processor issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen.
- The new thread must fill the pipeline before instructions will be able to complete.
- Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous Multithreading:

Simultaneous multithreading (SMT) is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction level parallelism

The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use. Register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability. Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is always executing instructions from multiple threads, leaving it up to

the hardware to associate instruction slots and renamed registers with their proper threads.

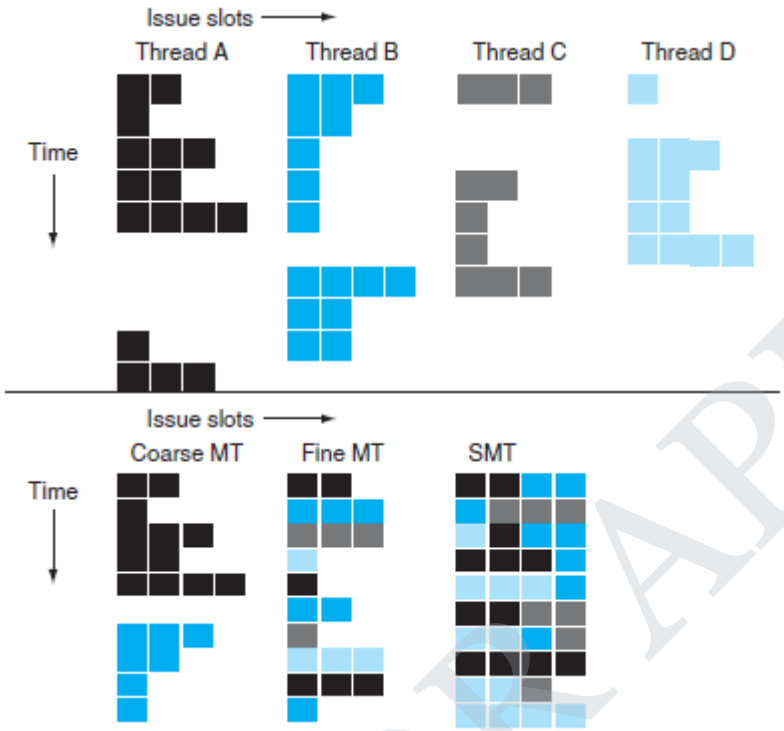


Figure 4.8 How four threads use the issue slots of a superscalar processor in different approaches.

The figure 4.8 illustrates the differences in a processor's ability to exploit Superscalar resources for the following processor configurations. The top portion shows how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

■ A superscalar with coarse-grained multithreading

Switches threads only on costly stalls.

Advantages: No switching each clock cycle, no slow down for ready-to-go threads. Reduces no of completely idle clock cycles.

Disadvantages: Limitations in hiding shorter stalls.

■ A superscalar with fine-grained multithreading

Switches threads on every clock cycle.

Advantages : Hide latency of from both short and long stalls.

Disadvantages: Slows down execution of individual threads.

■ A superscalar with simultaneous multithreading

Impact of fine grained scheduling on single thread performance.

A preferred thread approach sacrifices throughout and single threaded performance.

Part - C

5. MULTICORE PROCESSORS

❖ **Explain about Multicore Processors. (Nov/Dec 2014)(May/June 2016) (8)**

Multicore processors that contain any number of multiple CPUs on a single chip, such as 2, 4, 8. A multicore processor is composed of two or more independent cores. It is an integrated circuit which has two or more individual processors. These processors are called as cores. When these cores are integrated into a single integrated circuit die then they are known as a chip multiprocessor or CMP. Dual-core processors are among the multicore processors that consist of two CPUs on a single chip.

Advantage:

- Improves processor performance by performing more task simultaneously.
- It is inexpensive compared to having individual systems.
- Power consumption is less.
- Time to perform a particular task decreases
- Space in motherboard is utilized effectively.
- Intel Dual-core system has 2 cores onto a single chip.

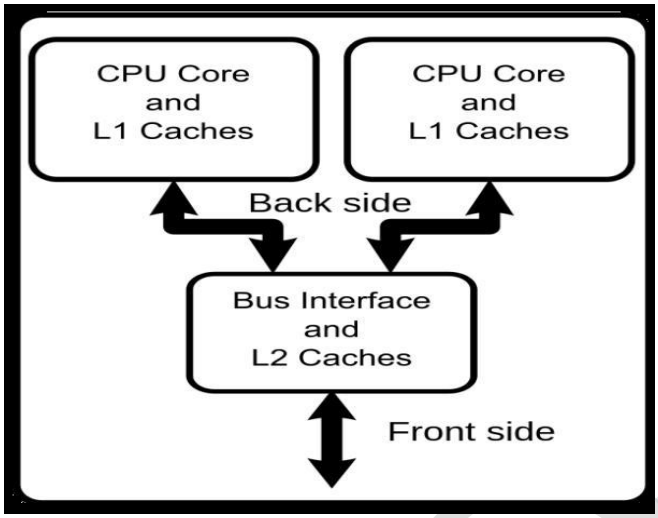


Figure 4.9 Dual CPU Core Chip

There are two types of memory used in multicore processor

a) Centralized shared Memory ("Uniform Memory Access" or "Shared Memory Processor")

b) Physically Distributed Memory Multiprocessor ("Decentralized memory" or Memory Module with CPU)

Centralized shared Memory Architecture

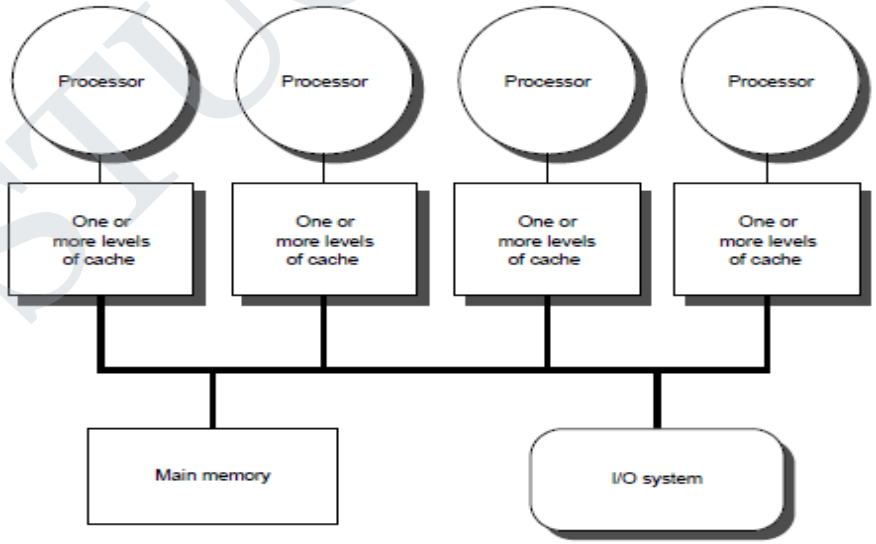


Figure 4.10 structure of centralized shared memory Architecture

Centralized shared memory Architecture share a single centralized memory, interconnect processors & memory by a bus. It is known as ―uniform Memory Access‖ (UMA) or Symmetric shared memory multiprocessor (SMP) because there is a single main memory that has,

- A symmetric relationship to all processors
- A uniform memory access time for any processor

Advantages:

Large caches can satisfy the memory demands of a small number of processor

Disadvantages:

Scalability problem: Ability to scale less number of processor

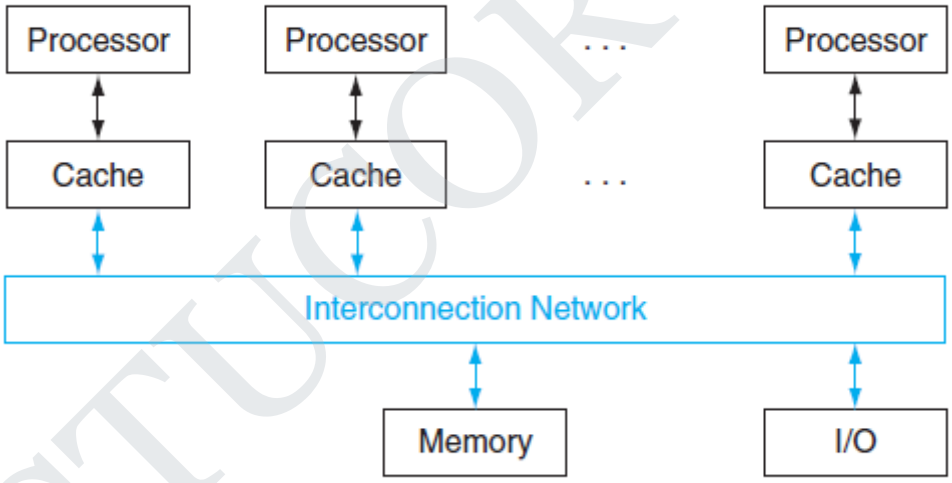


Figure 4.11 Classic Organization of an Shared Memory Processor

Figure 4.11 shows the classic organization of an SMP. Single address space multiprocessors come in two styles. The first takes about the same time to access main memory no matter which processor requests it and no matter which word is requested. Such machines are called uniform memory access (UMA) multiprocessors.

In the second style, some memory accesses are much faster than others, depending on which processor asks for which word. Such machines are called

nonuniform memory access (NUMA) multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor, but NUMA machines can scale to larger sizes and NUMAs can have lower latency to nearby memory.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called synchronization. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a lock for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable.

Clusters and other Message-Passing Multiprocessors:

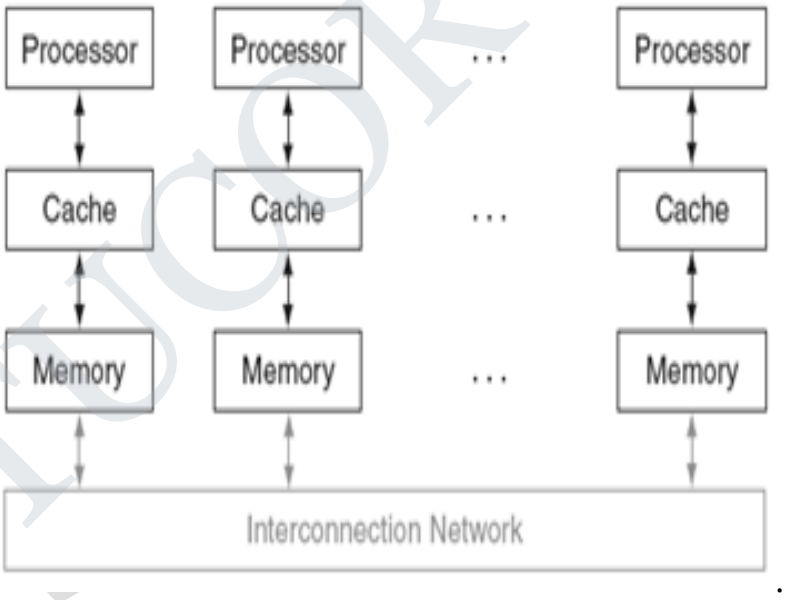


Figure 4.12 Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor

The alternative approach to sharing an address space is for the processors to each have their own private physical address space. Figure shows the classic organization of a multiprocessor with multiple private address spaces. This alternative multiprocessor must communicate via explicit message passing, which traditionally is

the name of such style of computers. Provided the system has routines to send and receive messages, coordination is built in with message passing, since one processor knows when a message is sent, and the receiving processor knows when a message arrives. If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the senderSome concurrent applications run well on parallel hardware, independent of whether it offers shared addresses or message passing. In particular, job-level parallelism and applications with little communication like Web search, mail servers, and file servers do not require shared addressing to run well. There were several attempts to build high-performance computers based on high performance message-passing networks, and they did offer better absolute communication performance than clusters built using local area networks.

The problem was that they were much more expensive. Few applications could justify the higher communication performance, given the much higher costs. Hence, clusters have become the most widespread example today of the message-passing parallel computer. Clusters are generally collections of commodity computers that are connected to each other over their I/O interconnect via standard network switches and cables. Each runs a distinct copy of the operating system. Virtually every Internet service relies on clusters of commodity servers and switches.

Limitations:

1. Clusters has been that the cost of administering a cluster of n machines is about the same as the cost of administering n independent machines, while the cost of administering a shared memory multiprocessor with n processors is about the same as administering a single machine. This weakness is one of the reasons for the popularity of virtual machines, since VMs make clusters easier to administer.

   For example, VMs make it possible to stop or start programs atomically, which simplifies software upgrades. VMs can even migrate a program from one computer in a cluster to another without stopping the program, allowing a program to migrate from failing hardware.

2. The processors in a cluster are usually connected using the I/O interconnect of each computer; whereas the cores in a multiprocessor are usually connected on the memory interconnect of the computer. The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance.
3. The overhead in the division of memory: a cluster of n machines has n independent memories and n copies of the operating system, but a shared memory multiprocessor allows a single program to use almost all the memory in the computer, and it only needs a single copy of the OS.

Properties of Multi-Core Systems:

- Cores will be shared with a wide range of other applications dynamically. Load can no longer be considered symmetric across the cores.
- Cores will likely not be asymmetric as accelerators become common for scientific hardware.
- Source code will often be unavailable, preventing compilation against the specific hardware configuration.

Applications that Benefit from Multi-Core:

- Database servers
- Web servers
- Telecommunication markets
- Multimedia applications
- Scientific applications

Distributed Memory Multicore Processor

In this every processor has its own memory space. In this system each processor core shares the entire memory; In distributed shared memory architecture directory based

cache coherence protocol is used. And every processor has its own memory space and the memory space can be shared among several processors. When the number of processor increases, the number of entries in the directory also increases. Both direct network (switches) & indirect network (multidimensional meshes) is used. Distributing the memory among the nodes has two major benefits,

- It is a cost- effective way to scale the memory bandwidth if most of scale the memory bandwidth if most of the accesses are to the local memory in the mode.
- It reduces the latency for accesses to the local memory

Advantages of Multicore Processor

- Increased responsiveness & worker productivity
- Improved performance in parallel environments when running computations on multiple processors

Classification based on communication models

Distributed shared Memory (DSM)

It is a shared address space in which communication occurs through a shared address space

Shared Address Space: the same physical address on two processor refers to the same location in memory

Architecture of Multicore processor

The major components of processor are

ALU : To perform arithmetic and logical operation.

Register file : A large number of register to hold the data and result during processing

Bus Interface : To interface the processor with the outside world.

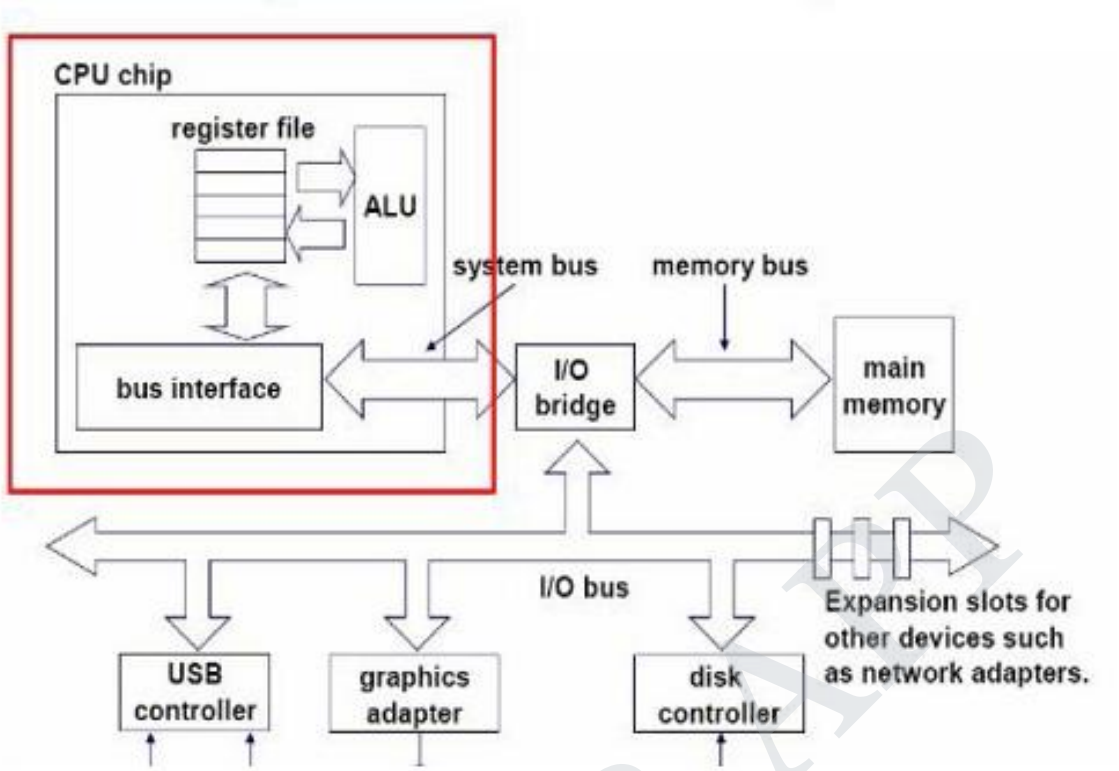


Figure 4.13 Interfacing the peripherals

To interface with peripheral devices, system bus is used. Expansion slot are used to expand the I/O bus so that more number of I/O devices can be connected

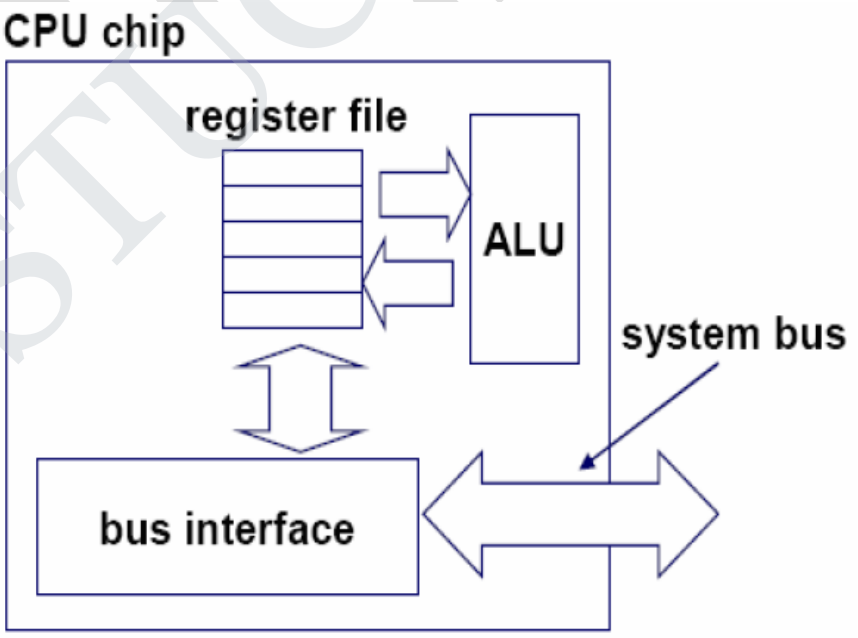


Figure 4.14 Single Core CPU Chip

Single Core CPU Chip:

The single processor architecture is shown in figure 4.14. Here only one processing unit is present in the chip for performing the arithmetic or logical operations. At any particular time, only one operation can be performed.

Chip Multiprocessor Architecture:

In multicore or chip multiprocessor architecture, multiple processing units or chips are present on a single die. Figure shows a multicore architecture with 4 cores in a single CPU chip. Here all the cores are fit on a single processor socket called as Chip Multiprocessor. The cores can run in parallel. Within each cores, threads can be time sliced similar to single processor systems

Applications that Benefit from Multi-Core:

- Database servers
- Web servers
- Telecommunication markets
- Multimedia applications
- Scientific applications

UNIT – V

UNIT V MEMORY AND I/O SYSTEMS 9

Memory hierarchy - Memory technologies – Cache basics – Measuring and improving cache performance - Virtual memory, TLBs - Input/output system, programmed I/O, DMA and interrupts, I/O M processors.

PART A

1. What is the need to implement memory as a hierarchy? (April/May 2015)

Computers use different types of memory units for different types of purposes. Each memory unit has its own advantages and disadvantages. A structure that uses multiple levels of memories is called hierarchy. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller. As the distance from the processor increases, the size of the memories and the access time both increases.

2. Compare SRAM from DRAM. (Nov/Dec 2013)

SRAMs are simply integrated circuits that are memory arrays with a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum. SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed.

3. What is meant by interleaved memory? (May/June 2012)

In computing, interleaved memory is a design made to compensate for the relatively slow speed of dynamic random-access memory (DRAM) or core memory, by spreading memory addresses evenly across memory banks. That way, contiguous memory reads and writes are using each memory bank in turn, resulting in higher memory throughputs due to reduced waiting for memory banks to become ready for desired operations.

It is different from multi-channel memory architectures, primarily as interleaved memory is not adding more channels between the main memory and the memory controller. However, channel interleaving is also possible, for example in free scale i.MX6 processors, which allow interleaving to be done between two channels.

4. **What is the purpose of Dirty /Modified bit in cache memory? (Nov/Dec 2014)**

A dirty bit or modified bit is a bit that is associated with a block of computer memory and indicates whether or not the corresponding block of memory has been modified. The dirty bit is set when the processor writes to (modifies) this memory. The bit indicates that its associated block of memory has been modified and has not yet been saved to storage. When a block of memory is to be replaced, its corresponding dirty bit is checked to see if the block needs to be written back to secondary memory before being replaced or if it can simply be removed. Dirty bits are used by the CPU cache and in the page replacement algorithms of an operating system.

5. Define cache hit and cache miss. (Nov/Dec 2013)

A cache hit is a state in which data requested for processing by a component or application is found in the cache memory. It is a faster means of delivering data to the processor, as the cache already contains the requested data.

Cache miss is a state where the data requested for processing by a component

or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory

6. What are the temporal and spatial localities of references? (May/June 2014)

Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon.

Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.

7. What is DMA? Mention its advantages.(Nov/Dec 2013, Nov/Dec 2014)

Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (RAM) independently of the central processing unit (CPU).

The need to handle more data and at higher rates means DMA is now an important part of hardware and software design. A dedicated DMA controller, often integrated in the processor, can be configured to move data between main memory and a range of subsystems, including another part of main memory.

8. What is the use of TLB? (April/May 2015)

A translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.Is is used to avoid an access to the page table.

9. **Point out how DMA can improve I/O speed. (April/May 2015)**

The CPU still responsible for initiating each block transfer. Then the DMA

interface controller can take the control and responsibility of transferring data. So that data can be transferred without the intervention of CPU. The CPU and I/O controller interacts with each other only when the control of bus is required.

10 **.Difference between Programmed I/O and Interrupt I/O. (Nov/Dec 2014)**

Programmed I/O	Interrupt I/O
The program is polling or checking some hardware item e.g. Mouse within a loop.	The same mouse will trigger a signal to the program to process the mouse event.
Slow and inefficient	Fast and efficient
Easy to program and understand	Can be tricky to write if you are using a low level language.

11. Define Hit Rate , Miss Rate ((Nov/Dec 2015)

Hit rate means the fraction of memory accesses found in a level of the memory hierarchy. Miss rate means the fraction of memory accesses not found in a level of the memory hierarchy.

PART B

1. MEMORY HIERARCHY

❖ **Explain about Memory Hierarchy in detail.**

The Memory Hierarchy Pyramid

IC Memory Types and Cycles:

ROM Read-only Memory (NEC)

Permanent storage (boot code, embedded code)

SRAM	Static Random Access Memory cache and high speed access
DRAM	Dynamic Random Access Memory (Micron) Main Memory
EPROM	Electrically programmable read-only memory (Atmel) Replace ROM when reprogramming required
EEPROM	Electrically erasable, programmable read-only memory (Atmel) Alternative to EPROM, limited but regular reprogramming, Device configuration info during power down (USB memories)
FLASH	advancement on EEPROM technology allowing blocks of memory location to be written and cleared at one time instead. (Samsung). Found in thumb drives/memory stick or as solid-state hard disks.

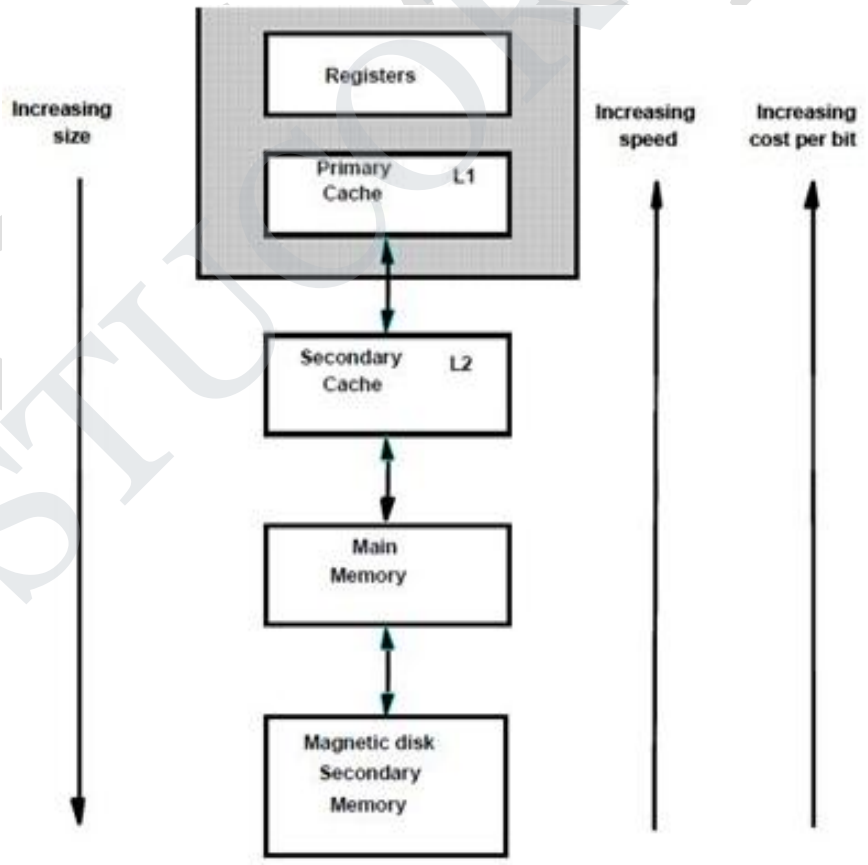


Figure 5.1 Memory Hierarchy

Average Memory Access Time (Registers and Main Memory):

The entire computer memory can be viewed as the hierarchy depicted in Figure. The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of the speed of access. The registers provide only a minuscule portion of the required memory.

At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the 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 levels of caches.

A primary cache is always located on the processor chip. This cache is small because it competes for space on the processor chip, which must implement many other functions. The primary cache is referred to as level (L1) cache. A larger, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as level 2 (L2) cache. It is usually implemented using SRAM chips. It is possible to have both Ll and L2 caches on the processor chip.

The next level in the hierarchy is called the main memory. This rather large memory is implemented using dynamic memory components, typically in the form of SIMMs, DIMMs, or RIMMs. The main memory is much larger but significantly slower than the cache memory. In a typical computer, the access time for the main memory is about ten times longer than the access time for the L1 cache.

Disk devices provide a huge amount of inexpensive storage. They are very slow compared to the semiconductor devices used to implement the main memory. A hard disk drive (HDD; also hard drive, hard disk, magnetic disk or disk drive) is a device for storing and retrieving digital information, primarily computer data. It consists of one or more rigid (hence "hard") rapidly rotating discs (often referred to as

platters), coated with magnetic material and with magnetic heads arranged to write data to the surfaces and read it from them.

2. MEMORY TECHNOLOGIES

- **Draw different memory access layouts and brief about the technique used to increase the average rate of fetching words from the main memory. (Nov/Dec 2014) (8)**
- **Elaborate on the various memory technologies and its relevance.(April/May 2015) (16)**

Memory latency is traditionally quoted using two measures access time and cycle time. Access time is the time between when a read is requested and when the desired word arrives, cycle time is the minimum time between requests to memory. One reason that cycle time is greater than access time is that the memory needs the address lines to be stable between accesses.

DRAM TECHNOLOGY:

The solution was to multiplex the address lines, thereby cutting the number of address pins in half. Figure shows the basic DRAM organization. One-half of the address is sent first, called the row access strobe (RAS). The other half of the address, sent during the column access strobe (CAS), follows it. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, D, for dynamic. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit destroys the information, so it must be restored. This is one reason the DRAM cycle time is much longer than the access time. In addition, to prevent loss of information when a bit is not read or written, the

bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row. Hence, every DRAM in the memory system must access every row within a certain time window, such as 8 ms. Memory controllers include hardware to refresh the DRAMs periodically

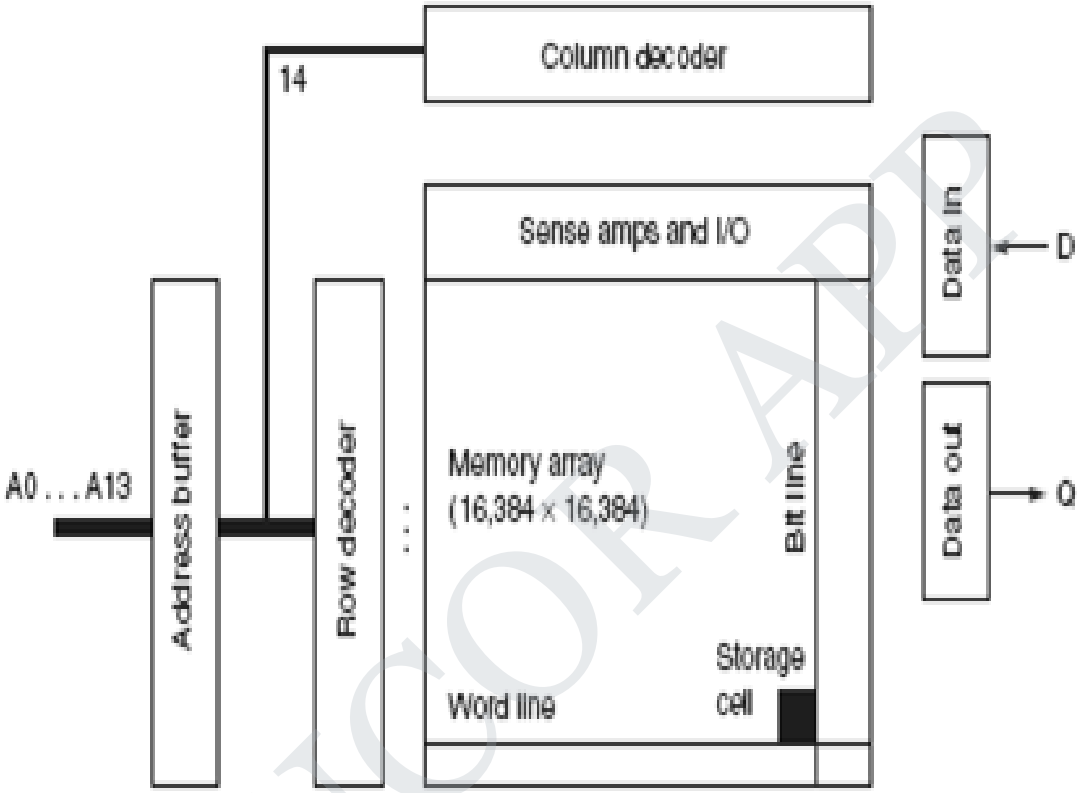


Figure 5.2 DRAM Technology

SRAM TECHNOLOGY:

The first letter of SRAM stands for static. The dynamic nature of the circuits in DRAM requires data to be written back after being read—hence the difference between the access time and the cycle time as well as the need to refresh. SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode. SRAM designs are concerned with speed and capacity, while in DRAM designs the

emphasis is on cost per bit and capacity. For memories designed in comparable technologies, the capacity of DRAMs is roughly 4–8 times that of SRAMs. The cycle time of SRAMs is 8–16 times faster than DRAMs, but they are also 8–16 times as expensive**.**

RAM BUS: This is an interface improvement using a pipelined bus interface sometimes called a split-transaction

- Bus comprises row and column address line + 18 bits of data
- transactions on bus simultaneously (RAS/CAS/Data)
- High clock rate (400MHz) with data transfers on both edges

ROM: Memory is the third key component of a microprocessor-based system (besides the CPU and I/O devices). More specifically, the primary storage directly addressed by the CPU is referred to as main memory to distinguish it from other "memory" structures such as CPU registers, caches, and disk drives. Main memory is typically built from several discrete semiconductor memory devices. Most systems contain two or more types of main memory.

All memory types can be categorized as ROM or RAM, and as volatile or non-volatile:

- Read-Only Memory (**ROM**) cannot be modified (written), as the name implies. A ROM chip's contents are set before the chip is placed in the system.
- Read-Write Memory is referred to as **RAM** (for Random-Access Memory). This distinction is inaccurate, since ROMs are also random access, but we are stuck with it for historical reasons.
- **Volatile** memories lose their contents when their power is turned off.
- **Non-volatile** memories do not.

The memory types currently in common usage are:

	ROM	RAM
Volatile	(nothing)	Static RAM (SRAM) Dynamic RAM (DRAM)
Non-volatile	Mask ROM PROM EPROM	EEPROM Flash memory BBSRAM

Every system requires some non-volatile memory to store the instructions that get executed when the system is powered up (the boot code) as well as some (typically volatile) RAM to store program state while the system is running.

PROGRAMMABLE ROM (PROM):

- Replace diode with diode + fuse, put one at every cell (a.k.a. "fusible-link" PROM)
- Initial contents all 1s; users program by blowing fuses to create 0s
- Plug chip into PROM programmer ("burner") device, download data file
- One-time programmable (OTP), bug throw it away

UV ERASABLE PROM (UV EPROM, OR JUST EPROM):

Replace PROM fuse with pass transistor controlled by "floating" (electrically isolated) gate Program by charging gate; switches pass transistor Careful application of high voltages to overcome gate insulation (again using special "burner") Erase by discharging all gates using ultraviolet light UV photons carry electrons across insulation Chip has window to let light in Insulation eventually breaks down limited number of erase/reprogram cycles (100s/1000s).

NON-VOLATILE RAM TYPES:

Three basic types:

- **EEPROM,**
- **Flash**
- **BBSRAM**

ELECTRICALLY ERASABLE PROM (EEPROM, E^2PROM):

Similar to UV EPROM, but with on-chip circuitry to electrically charge/discharge floating gates (no UV needed) Writable by CPU it's RAM, not ROM (despite name) Reads & writes much like generic RAM on writes, internal circuitry transparently erases affected byte/word, then reprograms to new value Write cycle time on the order of a millisecond typically poll status pin to know when write is done

- High-voltage input (e.g. 12V) often required for writing
- Limited number of write cycles (e.g. 10,000)
- Selective erasing requires extra circuitry (additional transistor) per memory cell lower density, higher cost than EPROM

FLASH MEMORY:

Again, floating-gate technology like EPROM, EEPROM Electrically erasable like EEPROM, but only in large 8K-128K blocks (not a byte at a time) Moves erase circuitry out of cells to periphery of memory array Back to one transistor/cell excellent density Reads just like memory Writes like memory for locations in erased blocks typ. write cycle time is a few microseconds slower than volatile RAM, but faster than EEPROM To rewrite a location, software must explicitly erase entire block initiated via control registers on flash memory device erase can take several seconds erased blocks can be written (programmed) a byte at a time Still have erase/reprogram cycle limit (10K-100K cycles per block).

FLASH APPLICATIONS:

Flash technology has made rapid advances in last few years cell density rivals DRAM; better than EPROM, much better than EEPROM. Multiple gate voltage levels can encode 2 bits per cell 64 Mbit devices available ROMs & EPROMs rapidly becoming obsolete as cheap or cheaper, allows field upgrades Replacing hard disks in some applications smaller, lighter, faster more reliable (no moving parts) cost-effective up to tens of megabytes block erase good match for file-system type interface.

BATTERY-BACKED STATIC RAM (BBSRAM):

Take volatile static RAM device and add battery backup Key advantage: write performance write cycle time same as read cycle time Need circuitry to switch to battery on power-off have to worry about battery running out Effective for small amount of storage when you need battery anyway (e.g. PC built-in clock)

VOLATILE RAM TYPES:

Two basic types:

- **Static**
- **Dynamic**

STATIC RAM (SRAM):

- Each cell is basically a flip-flop
- Four or six transistors (4T/6T) relatively poor density
- Very simple interfacing; writes & reads at same speed
- Very fast (access times under 10 ns available)

DYNAMIC RAM (DRAM):

- One transistor per cell (drain acts as capacitor)
- Highest density memory available
- Very small charges involved

    - bit lines must be precharged to detect bit values: cycle time > access time
    - reads are destructive; internally does writeback on read
    - values must be refreshed (rewritten) periodically by touching each row of array or charge will leak away

- External row/column addressing saves pins, $
- Row/column addressing + refresh complex interfacing

EMBEDDED PROCESSOR MEMORY TECHNOLOGY: ROM AND FLASH

Embedded computers usually have small memories, and most do not have a disk to act as non-volatile storage. Two memory technologies are found in embedded computers to address this problem.

The first is Read-Only Memory (ROM). ROM is programmed at time of manufacture, needing only a single transistor per bit to represent 1 or 0. ROM is used for the embedded program and for constants, often included as part of a larger chip. In addition to being non-volatile, ROM is also non-destructible; nothing the computer can do can modify the contents of this memory. Hence, ROM also provides a level of protection to the code of embedded computers. Since address based protection is often not enabled in embedded processors, ROM can fulfill an important role.

The second memory technology offers non-volatility but allows the memory to be modified. Flash memory allows the embedded device to alter nonvolatile memory after the system is manufactured, which can shorten product development.

IMPROVING MEMORY PERFORMANCE IN A STANDARD DRAM CHIP:

1. **To improve bandwidth**, there have been a variety of evolutionary innovations over time. The first was timing signals that allow repeated accesses to the row buffer without another row access time, typically called fast page mode.

2. **The second major change** is that conventional DRAMs have an asynchronous interface to the memory controller, and hence every transfer involves overhead to synchronize with the controller. This optimization is called Synchronous DRAM (SDRAM).

3. **The third major DRAM innovation** to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate. This optimization is called Double Data Rate (DDR).

3. CACHE BASICS – MEASURING AND IMPROVING CACHE PERFORMANCE

❖ **Explain the features of cache memory and its accesses. (May/June 2014).**

BASICS Basic Ideas:

The cache is a small mirror-image of a portion (several "lines") of main memory. cache is faster than main memory ==> so maximize its utilization cache is more expensive than main memory ==> so it is much smaller .

Locality of reference:

The principle that the instruction currently being fetched/executed is very close in memory to the instruction to be fetched/executed next. The same idea applies to the

data value currently being accessed (read/written) in memory.

If we keep the most active segments of program and data in the cache, overall execution speed for the program will be optimized. Our strategy for cache utilization should maximize the number of cache read/write operations, in comparison with the number of main memory read/write operations.

Example:

A line is an adjacent series of bytes in main memory (that is, their addresses are contiguous). Suppose a line is 16 bytes in size. For example, suppose we have a 2^{12}= 4K-byte cache with 2^8 = 256 16-byte lines; a 2^{24} = 16M-byte main memory, which is 2^{12} = 4K times the size of the cache; and a 400-line program, which will not all fit into the cache at once.

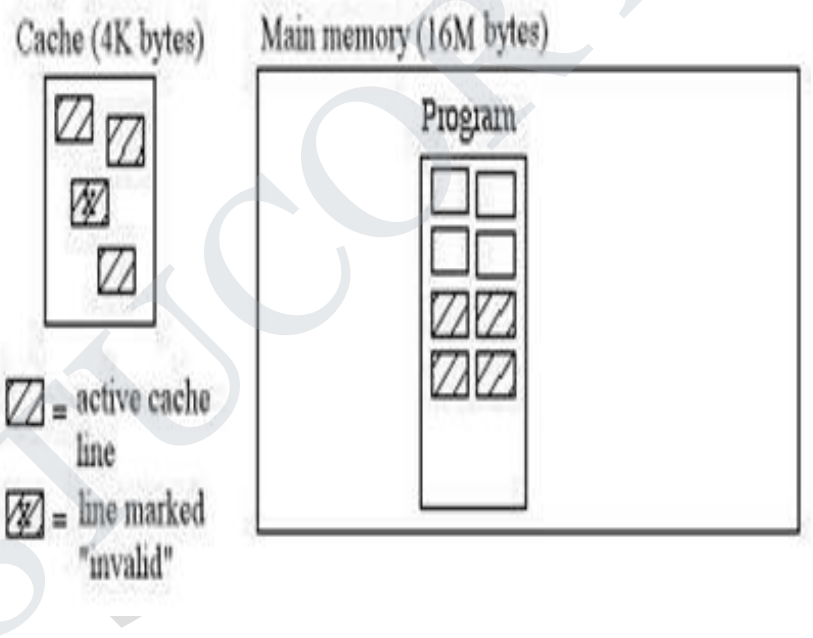


Figure 5.3 Cache Memory

Each active cache line is established as a copy of a corresponding memory line during execution. Whenever a memory write takes place in the cache, the "Valid" bit is reset (marking that line "Invalid"), which means that it is no longer an exact image of its corresponding line in memory.

Cache Dynamics:

When a memory read (or fetch) is issued by the CPU:

1. If the line with that memory address is in the cache (this is called a cache hit), the data is read from the cache to the MDR.
2. If the line with that memory address is not in the cache (this is called a miss), the cache is updated by replacing one of its active lines by the line with that memory address, and then the data is read from the cache to the MDR.

When a memory write is issued by the CPU:

1. If the line with that memory address is in the cache, the data is written from the MDR to the cache, and the line is marked "invalid" (since it no longer is an image of the corresponding memory line).
2. If the line with that memory address is not in the cache, the cache is updated by replacing one of its active lines by the line with that memory address. The data is then written from the MDR to the cache and the line is marked "invalid."

Cache updation is done in the following way:

1. A candidate line is chosen for replacement using an algorithm that tries to minimize the number of cache updates throughout the life of the program run. Two algorithms have been popular in recent architectures: Choose the line that has been least recently used - "LRU" for short (e.g., the PowerPC) Choose the line randomly (e.g., the 68040)
2. If the candidate line is "invalid," write out a copy of that line to main memory (thus bringing the memory up to date with all recent writes to that line in the cache).
3. Replace the candidate line by the new line in the cache.

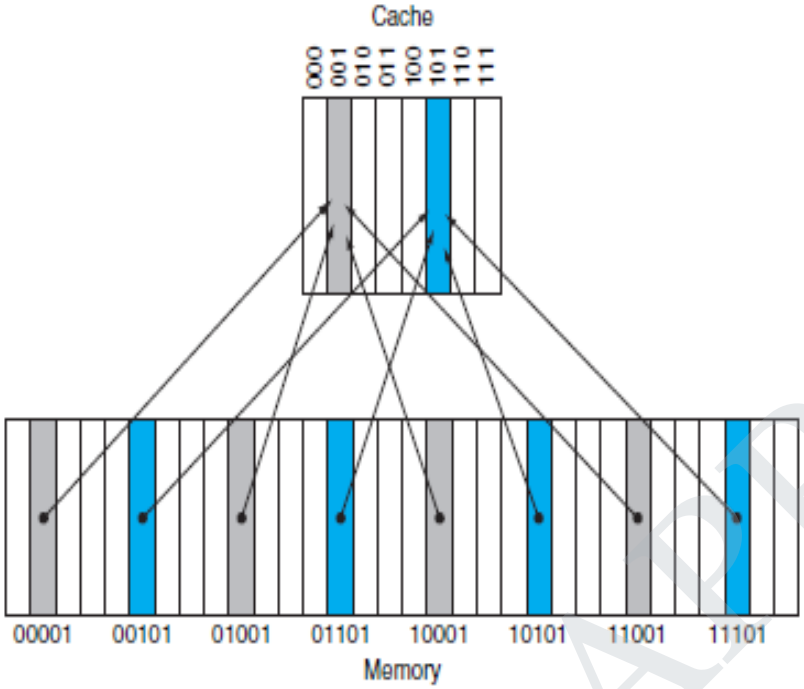


Figure 5.4 Direct-mapped cache with eight entries showing the addresses of memory words

❖ **Explain mapping functions in cache memory to determine how memory blocks are placed in cache. (Nov/Dec 2014, May/June 2016)**

As a working example, suppose the cache has 2^7 = 128 lines, each with 2^4 = 16 words. Suppose the memory has a 16-bit address, so that 2^{16} = 64K words are in the memory's address space.

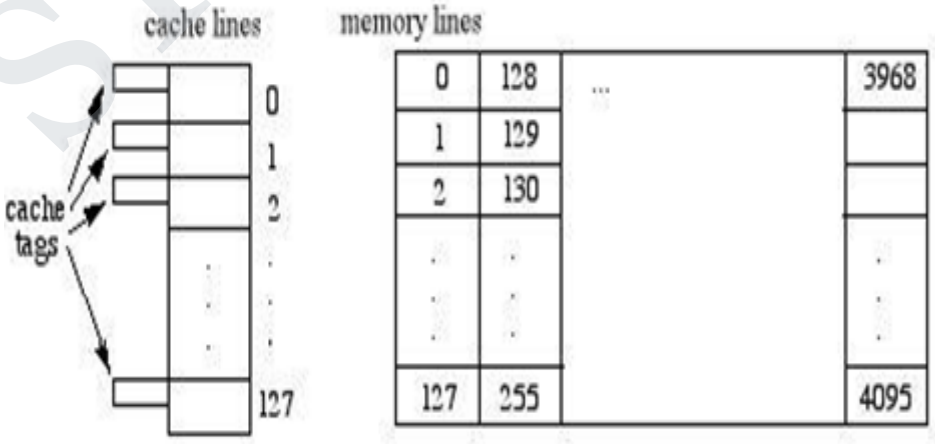
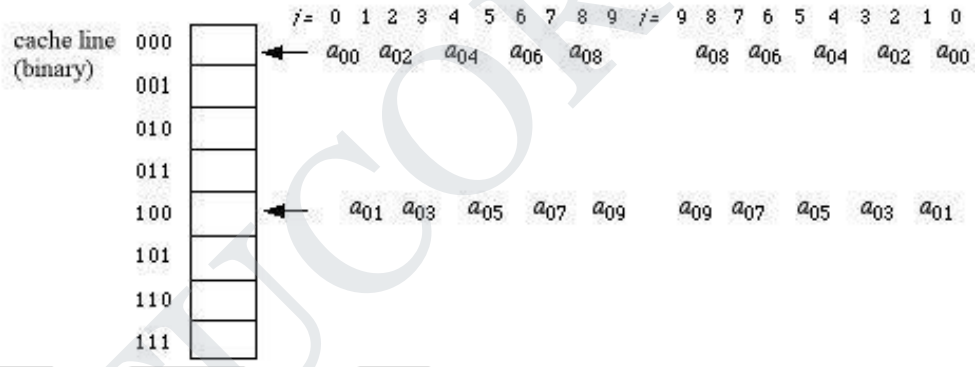


Figure 5.5 Mapping

Direct Mapping:

Direct mapping of the cache for this model can be accomplished by using the rightmost 3 bits of the memory address. For instance, the memory address 7A00 = 0111101000000 000, which maps to cache address 000. Thus, the cache address of any value in the array a is just its memory address modulo 8.

Using this scheme, we see that the above calculation uses only cache words 000 and 100, since each entry in the first row of a has a memory address with either 000 or 100 as its rightmost 3 bits. The hit rate of a program is the number of cache hits among its reads and writes divided by the total number of memory reads and writes. There are 30 memory reads and writes for this program, and the following diagram illustrates cache utilization for direct mapping throughout the life of these two loops:

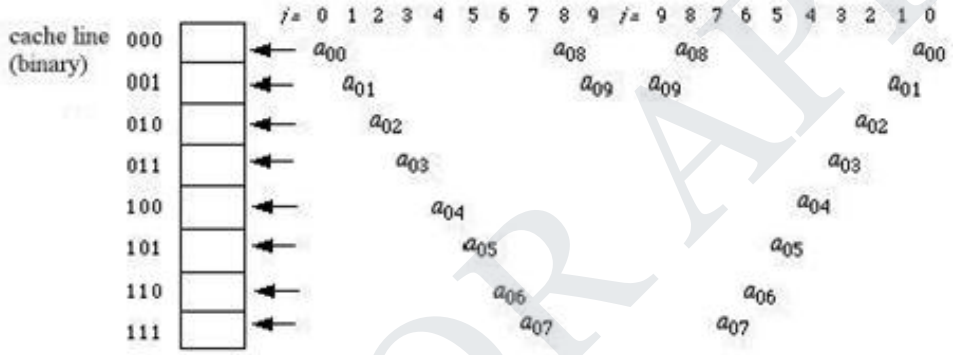


Reading the sequence of events from left to right over the ranges of the indexes i and j, it is easy to pick out the hits and misses. In fact, the first loop has a series of 10 misses (no hits). The second loop contains a read and a write of the same memory location on each repetition (i.e., a[0][i] = a[0][i]/Ave;),

so that the 10 writes are guaranteed to be hits. Moreover, the first two repetitions of the second loop have hits in their read operations, since a_{09} and a_{08} are still in the cache at the end of the first loop. Thus, the hit rate for direct mapping in this algorithm is 12/30 = 40%.

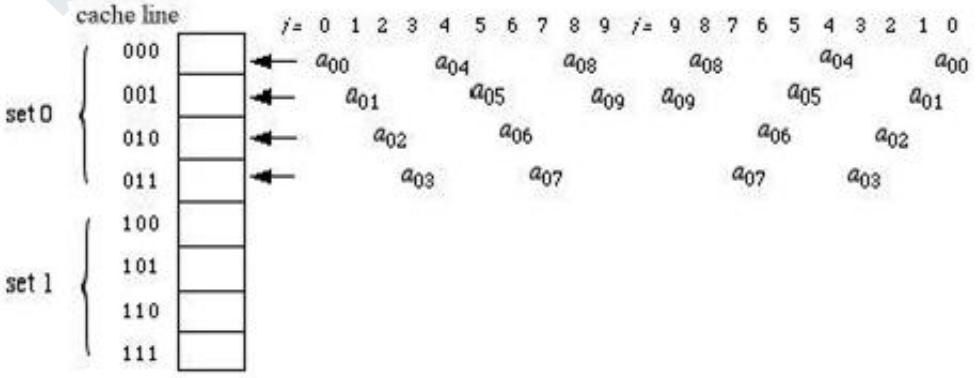
Associative Mapping:

Associative mapping for this problem simply uses the entire address as the cache tag. If we use the least recently used cache replacement strategy, the sequence of events in the cache after the first loop completes is shown in the left-half of the following diagram. The second loop happily finds all of a_{09} - a_{02} already in the cache, so it will experience a series of 16 hits (2 for each repetition) before missing on a $_{01}$ when i=1. The last two steps of the second loop therefore have 2 hits and 2 misses.



Set-Associative Mapping:

Set associative mapping tries to compromise these two. Suppose we divide the cache into two sets, distinguished from each other by the rightmost bit of the memory address, and assume the least recently used strategy for cache line replacement. Cache utilization for our program can now be pictured as follows:

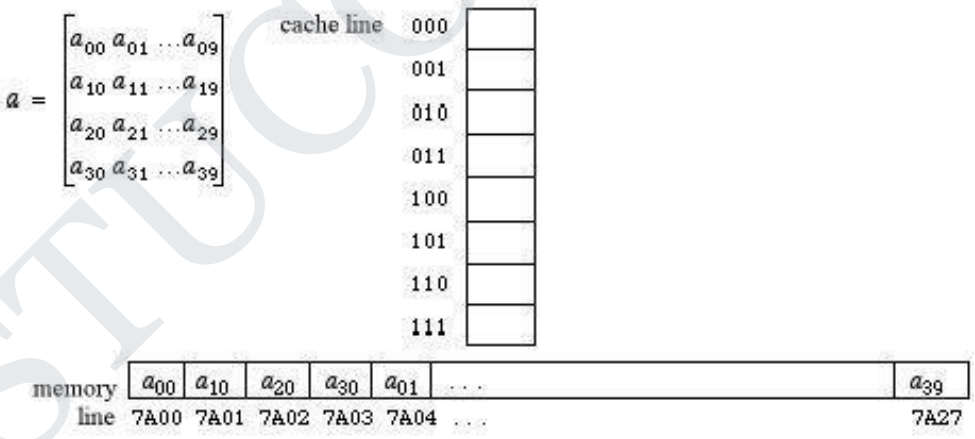


Here all the entries in that are referenced in this algorithm have even-numbered addresses (their rightmost bit = 0), so only the top half of the cache is utilized. The hit rate is therefore slightly worse than associative mapping and slightly better than direct. That is, set-associative cache mapping for this program yields 14 hits out of 30 read/writes for a hit rate of 46%.

Example:

Suppose we have an 8-word cache and a 16-bit memory address space, where each memory "line" is a single word (so the memory address needs not have a "Word" field to distinguish individual words within a line). Suppose we also have a 4x10 array a of numbers (one number per addressable memory word) allocated in memory column-by-column, beginning at address 7A00. That is, we have the following declaration and memory allocation picture for the array a.

float [][] a = new float [4][10];



Here is a simple equation that recalculates the elements of the first row of a:

$$a_{0,i} = \frac{a_{0,i}}{\sum_{j=0}^{9} a_{0,j} / 10} \qquad \text{for all } i = 0, 1, \ldots, 9$$

This calculation could have been implemented directly in C/C++/Java as follows:

```
Sum = 0;
for (j=0; j<=9; j++)
   Sum = Sum + a[0][j];
   Ave = Sum /10;
for (i=9; i>=0; i--)
   a[0][i] = a[0][i] / Ave;
```

The emphasis here is on the underlined parts of this program which represent memory read and write operations in the array a. Note that the 3rd and 6th lines involve a memory read of a[0][j] and a[0][i], and the 6th line involves a memory write of a[0][i]. So altogether, there are 20 memory reads and 10 memory writes during the execution of this program. The following discussion focuses on those particular parts of this program and their impact on the cache.

4. BUS ARBITRATION

- **Explain in detail about any two standard input and output interface required to connect the I/O device to the bus.(Nov/Dec 2014).**
- **Explain in detail about the Bus Arbitration techniques in DMA. (Nov /Dec 2014)**

Multiple devices may need to use the bus at the same time so it must have a way to arbitrate multiple requests. Bus arbitration schemes usually try to balance:

- Bus priority – the highest priority device should be serviced first
- Fairness – even the lowest priority device should never be completely locked out from the bus

Bus arbitration schemes can be divided into three classes

1.Daisy chaining.

2.Polling.

3.Independent requesting.

Daisy Chaining

In Daisy Chaining method all masters make use of the same line for bus request. In response to a bus request, the controller sends a bus grant if the bus is free. The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This master blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus. Therefore any other requesting module will not receive the grant signal and hence cannot get the bus access. This bus allocation scheme is simple and cheaper But failure of any one master causes the whole system to fail and arbitration is slow due to the propagation delay of bus grant signal is proportional to the number of masters

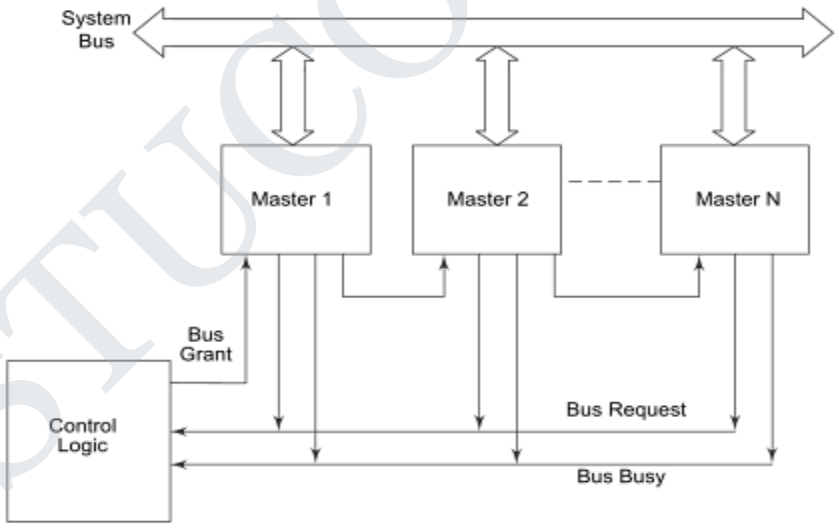


Figure 5.6 Daisy Chaining

Polling

In polling method, the controller sends address of device to grant bus access. The number of address lines required is depend on the number of masters connected in the system. For example, if 3

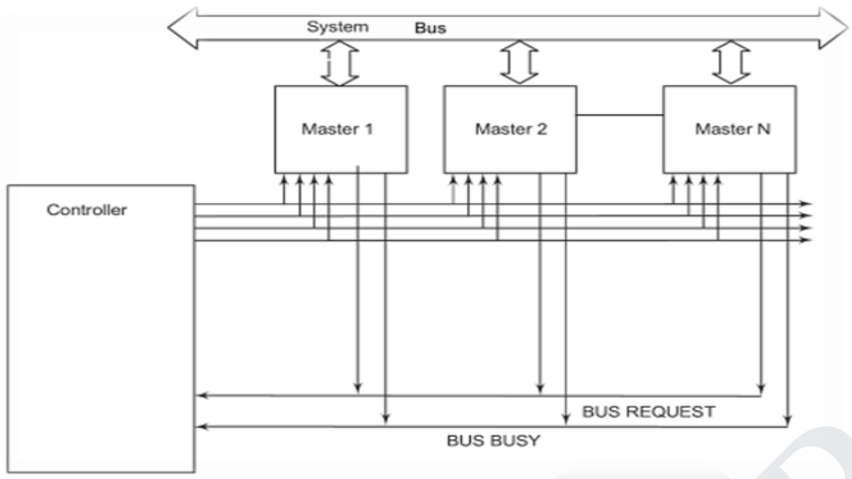


Figure 5.7 Polling

masters are connected in the system, one address line is required. In response to a bus request, controller generates a sequence of master addresses When the requesting master recognizes the address, it activates the busy line and begins to use the bus. The priority can be changed by altering the polling sequence stored in the controller Another one advantage of this method is, if one module fails entire system does not fail.

Independent Priority

In the independent priority scheme each master has a separate pair of bus request (BRQ) and bus grant (BGR) lines and each pair has a priority assigned to it. The built in priority decoder within the controller selects the highest priority request and asserts the.

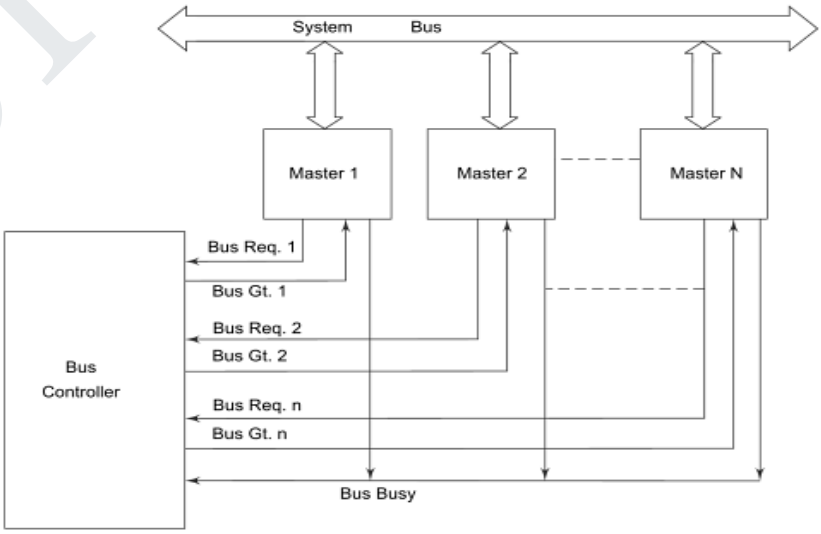


Figure 5.8 Independent Priority

corresponding bus grant signal. Synchronization of clocks must be performed once a master is recognized, Master will receive a common clock from one side and pass it to the controller which will derive a clock for transfer. Due to separate pairs of bus request and bus grant signals, arbitration is fast

Bus Grant : This signal means that a unit has been granted bus access and can take control.

Bus Request : This signal means that the unit has requested for the grant of the bus access and requests to take control of the bus.

Busy: This signal is to and from a bus master to enables all other units with the bus to note that presently bus access is not possible as one of the units is busy using the bus or has been granted control over the bus. The unit, which accepts the Bus Grand issues the Busy.

Centralized Arbitration:

Processor is normally the bus master grants bus mastership to DMA. DMA controller 2 requests and acquires bus and later releases the bus. During its tenure as the bus master, it ma or more data transfer operations, depending on operating in the cycle stealing or block mode. After it releases the bus, the processor.

Distributed Arbitration:

All devices waiting to use the bus have to the arbitration process - no central arbiter each device on the bus is assigned wit identification number. One or more devices request the bus by the start-arbitration signal and pl identification number on the four open lines . ARB0 through ARB3 are the four open lines .One among the four is selected using the lines and one with the highest ID.

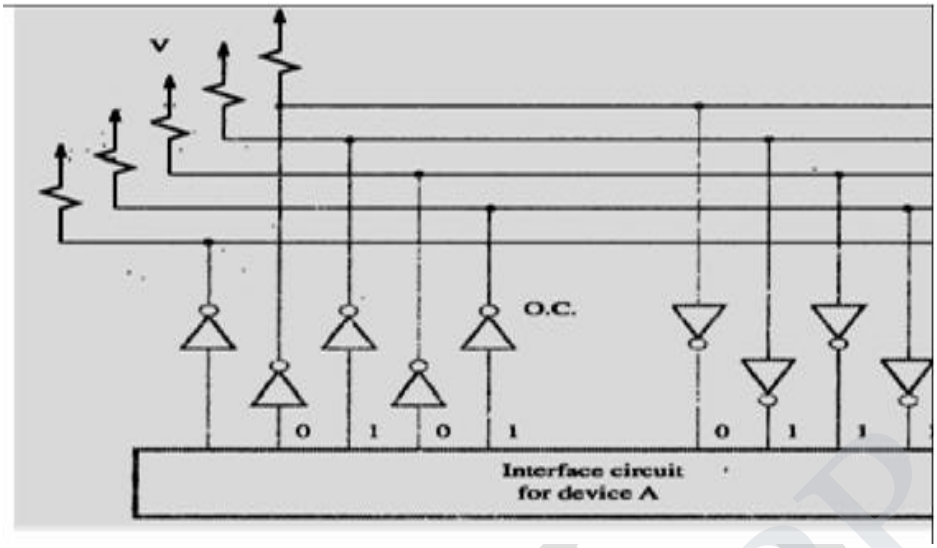


Figure 5.9 Distributed Arbitration

Assume that two devices, A and B, having 5 and 6, respectively, are requesting the use of .Device A transmits the pattern 0101, and transmits the pattern 0110.The code seen by both devices is 0111.Each device compares the pattern on the lines to its own ID, starting from the most significant. If it detects a difference at any bit position its drivers at that bit position and for all lower-does so by placing a 0 at the input of these driver. In the case of our example, device A dete difference on line ARB I. Hence, it disables its d lines ARB 1 and ARBO.

❖ **Explain the techniques for measuring and improving cache performance. (May/June 2014).**

We then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called multilevel caching.

Thus, CPU time = (CPU execution clock cycles + Memory-stall clock cycles) × Clock cycle time. The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified

model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes: Memory-stall clock cycles = Read-stall cycles + Write-stall cycles. The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

Read-stall cycles = Reads Program × Read miss rate × Read miss penalty Writes are more complicated.

Write-stall cycles = (Writes Program × Write miss rate × Write miss penalty) + Write buffer stalls.

Because the write buffer stalls depend on the proximity of writes, and not just the frequency, it is not possible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them.

If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

Memory-stall clock cycles = Memory accesses Program × Miss rate × Miss penalty

Memory-stall clock cycles = Instructions Program ×Misses Instruction × Miss Penalty

Calculating Cache Performance:

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

Instruction miss cycles = I × 2% × 100 = 2.00 × I

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references.

Data miss cycles = I × 36% × 4% × 100 = 1.44 × I

The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is CPU time with stalls.

CPU time with perfect cache=I × CPIstall × Clock cycle (I) × CPIperfect × Clock cycle

=CPIstall x CPIperfect

= 5.44

The performance with the perfect cache is better by 5.44

Calculating Average Memory Access Time:

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of

20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls. The average memory access time per instruction is

AMAT = Time for a hit + Miss rate × Miss penalty = 1 + 0.05 × 20= 2 clock cycles or 2 ns.

Reducing the Miss Penalty Using Multilevel Caches:

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is usually on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

Performance of Multilevel Caches Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz. Assume a main memory access time of 100 ns, including all the miss handling. Suppose the miss rate per instruction at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5 ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%? The miss penalty to main memory is

$$\frac{100 \text{ ns}}{0.25 \dfrac{\text{ns}}{\text{clock cycle}}} = 400 \text{ clock cycles}$$

$$\frac{5 \text{ ns}}{0.25 \dfrac{\text{ns}}{\text{clock cycle}}} = 20 \text{ clock cycles}$$

REDUCING CACHE MISS RATE:

The average memory access time formula gave us a framework to present cache optimizations for improving cache performance:

Average memory access time = Hit time + Miss rate × Miss Penalty

Hence, we organize six cache optimizations into three categories: Reducing the miss rate: larger block size, larger cache size, and higher associativity Reducing the miss penalty: multilevel caches and giving reads priority over writes Reducing the time to hit in the cache: avoiding address translation when indexing the cache.

SPLIT CACHES:

The classical approach to improving cache behavior is to reduce miss rates, and we resent three techniques to do so. To gain better insights into the causes of misses, we first start with a model that sorts all misses into three simple categories:

Compulsory: The very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called cold-start misses or first-reference misses.

Capacity: If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.

Conflict: If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses. The idea is that hits in a fully associative cache that become misses in an n-way set-associative cache are due to more than n requests on some popular sets.

5. VIRTUAL MEMORY

- **What is virtual memory? Explain the steps involved in virtual memory address translation. (April/May 2015)**
- **What is virtual memory? Explain in detail about how virtual memory is implemented with neat diagram.(Nov/Dec 2015)**

The main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks. This technique is called **virtual memory**. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing of memory among multiple programs, and to remove the programming burdens of a small, limited amount of main memory.

A virtual memory block is called a page, and a virtual memory miss is called a page fault. With virtual memory, the processor produces a virtual address, which is translated by a combination of hardware and software to a physical address, which in turn can be used to access main memory. Figure shows the virtually addressed memory with pages mapped to main memory. This process is called address mapping or address translation.

Virtual memory also simplifies loading the program for execution by providing relocation. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory

Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system need only find a sufficient number of pages in main memory.

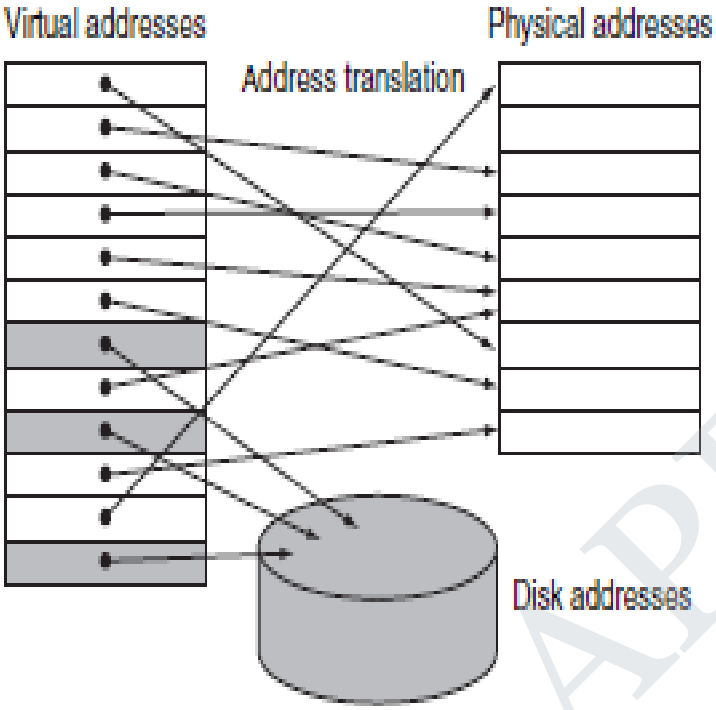


Figure 5.10 Blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).

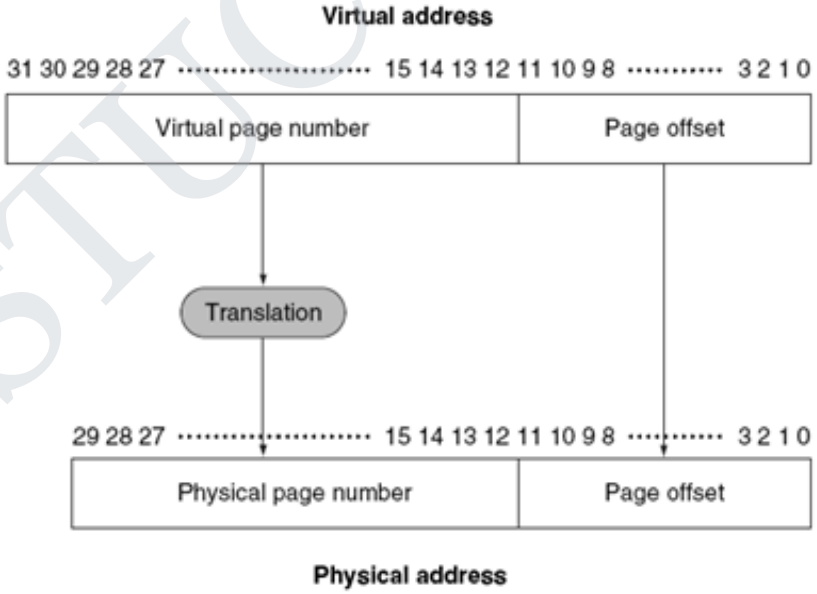


Figure 5.11 Mapping from a virtual to a physical address.

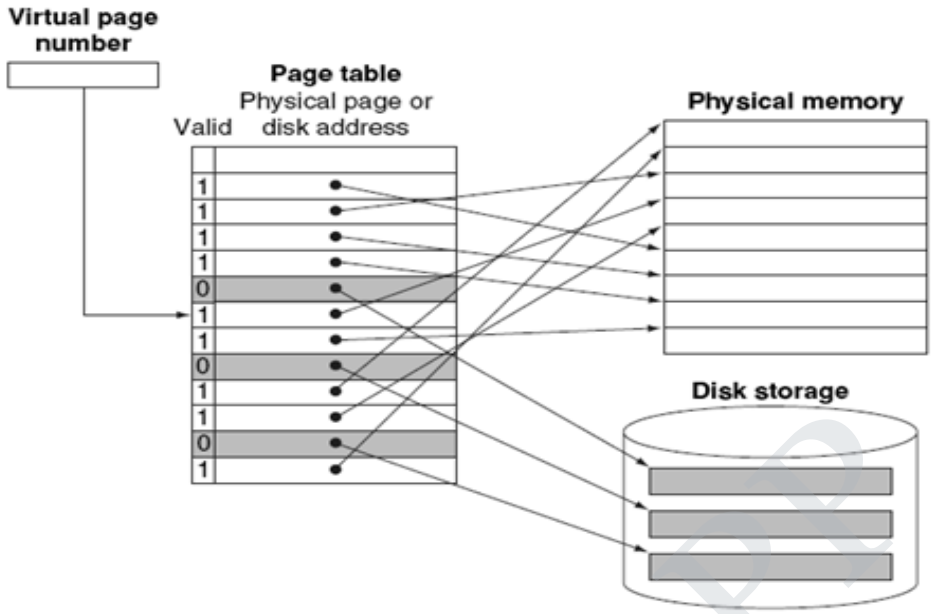


Figure 5.12 The page table maps each page in virtual memory.

The physical main memory is not as large as the address space spanned by an address issued by the processor. When a program does not completely fit into the main memory, the parts of it not currently being executed are stored on secondary storage devices, such as magnetic disks.

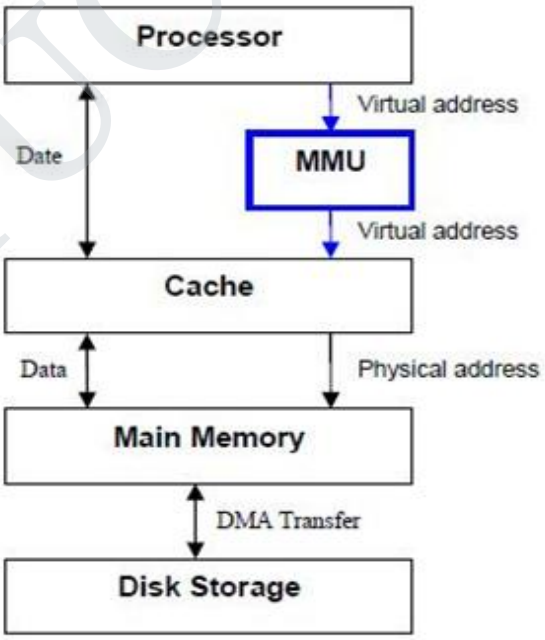


Figure 5.13 Virtual Address

When a new segment of a program is to be moved into a full memory, it must replace another segment already in the memory. The operating system moves programs and data automatically between the main memory and secondary storage. This process is known as swapping. Thus, the application programmer does not need to be aware of limitations imposed by the available main memory.

A special hardware unit, called the Memory Management Unit (MMU), translates virtual addresses into physical addresses. When the desired data (or instructions) are in the main memory, these data are fetched as described in our presentation of the ache mechanism. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. The DMA scheme is used to perform the data Transfer between the disk and the main memory.

❖ **Explain the steps involved in virtual address translation. (April/May 2015).**

The MMU must use the page table information for every read and write access; so ideally, the page table should be situated within the MMU.

The page table may be rather large, and since the MMU is normally implemented as part of the processor chip (along with the primary cache), it is impossible to include a complete page table on this chip. Therefore, the page table is kept in the main memory. However, a copy of a small portion of the page table can be accommodated within the MMU.

The process of translating a virtual address into physical address is known as address translation. It can be done with the help of MMU. A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of

information that is moved between the main memory and the disk whenever the translation mechanism determines that a move is required.

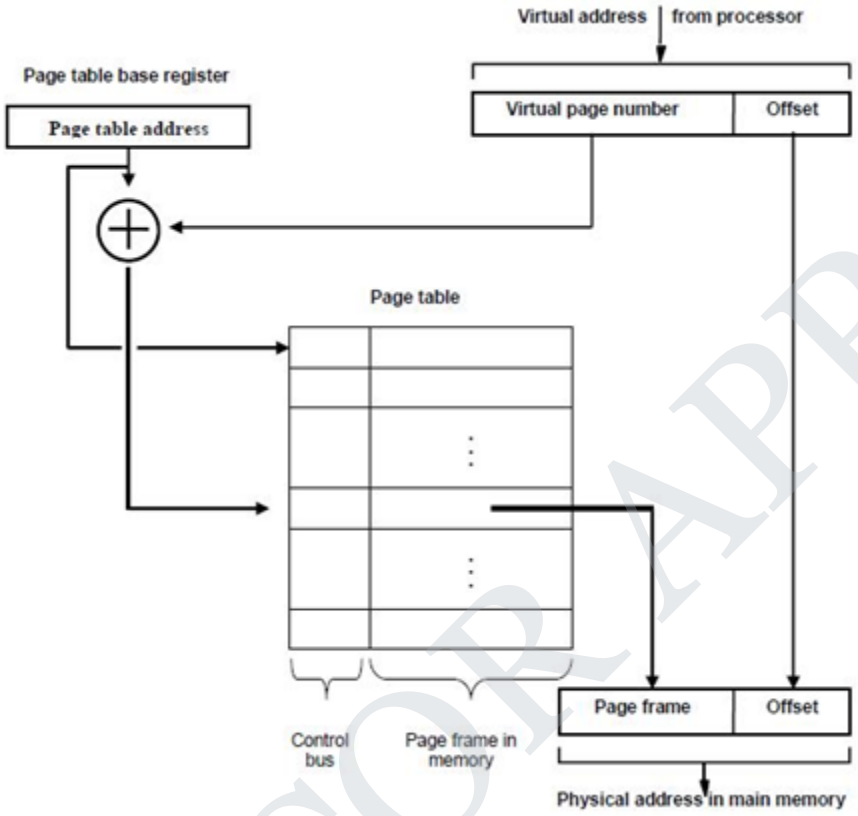


Figure 5.14 Virtual Address Translation

The cache bridges the speed gap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques.

A virtual-memory address translation method based on the concept of fixed-length pages. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand fetch/store operation, is interpreted as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a page table. This information includes the main memory address where the page is stored and the current status of the page.

This bit allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the 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.

TLBS- INPUT/OUTPUT SYSTEM:

This portion consists of the page table entries that correspond to the most recently accessed pages. A small cache, usually called the Translation Lookaside Buffer (TLB) is incorporated into the MMU for this purpose. The operation of the TLB with respect to the page table in the main memory is essentially the same as the operation of cache memory; the TLB must also include the virtual address of the entry. Figure shows a possible organization of a TLB where the associative-mapping technique is used. Set associative mapped TLBs are also found in commercial products.

An essential requirement is that the contents of the TLB be coherent with the contents of page tables in the memory. When the operating system changes the contents of page tables, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB will acquire the new information as part of the MMU's normal response to access misses.

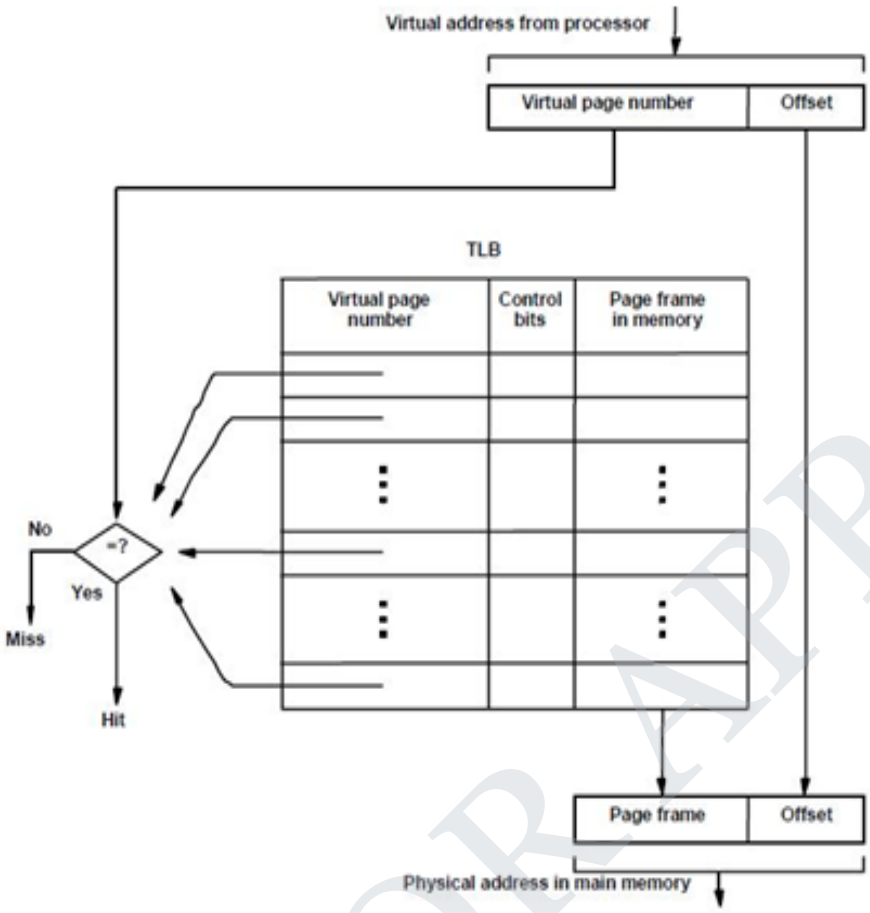


Figure 5.15 Translation Lookaside Buffer

Given a virtual address, the MMU looks in the TLB for the referenced page. Page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated. 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 whole page must be brought from the disk into the memory before access can proceed.

A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory.

PART C

- **Discuss Direct Memory Access in detail. (May/June 2014).(16)**
- **Draw the typical block diagram of a DMA controller and explain how it is used for direct data transfer between memory and peripherals. (Nov/Dec 2015).**
- **Explain about DMA controller, with the help of block diagram(May/June 2016)**

A special control unit is provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access, or DMA.

DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a DMA controller. 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 all the bus signals that control data transfer.

Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor. To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal.

While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer. I/O operations are always performed by the operating system of the computer in response to a request from an application program.

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, the controller performs a read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a write operation.

A DMA controller connects a high-speed network to the computer 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 device.

To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller. It also provides the disk controller with information to identify the data for future retrieval. The DMA controller proceeds independently to implement the specified operation.

When the DMA transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. 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 can also be used to record other information, such as whether the transfer took place correctly or errors occurred.

Memory accesses by the processor and the DMA controllers are interwoven. Requests by DMA devices for using the bus are always given higher priority than processor requests. Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device.

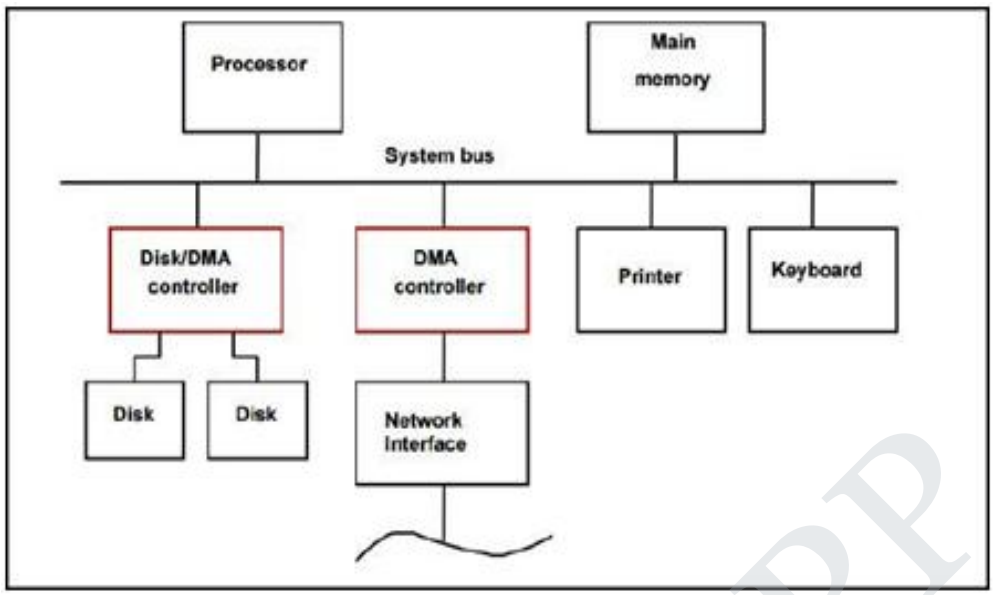


Figure 5.16 Direct Memory Access Data Transfer

Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as block or burst mode. Most DMA controllers incorporate a data storage buffer. In the case of the network interface for example, the DMA controller reads a block of data from the main memory and stores it into its input buffer. This transfer takes place using burst mode at a speed appropriate to the memory and the computer bus. Then, the data in the buffer are transmitted over the network at the speed of the network.

INDUSTRY CONNECTIVITY AND LATEST DEVELOPMENTS

Industry Connectivity:

- The following companies (Industries) are connectivity to computer architecture design: Intel, National Instruments, IBM, Free scale semiconductor

Latest Developments:

- Design for the evolution of the next generation of POWER and Z series Processors

Industrial Visit (Planned if any)

Industry: Arasan Chip Design Pvt. Ltd, Tuticorin

UNIVERSITY QUESTIONS

Question Paper Code: 11257

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011
Fourth Semester
Computer Science and Engineering
CS 2253 - COMPUTER ARCHITECTURE
(Common to Information Technology)
(Regulation 2008)

Time: 3 hours **Maximum: 100 marks**

Answer ALL questions

PART -A (10 x 2 = 20 marks)

1. What is an opcode? How many bits are needed to specify 32 distinct operations?
2. Write the logic equations of a binary half adder.
3. Write the difference between Horizontal and vertical Microinstructions.
4. In what ways the width and height of the control memory can be reduced?
5. What hazard does the above two instructions create when executed concurrently?
6. What are the disadvantages of increasing the number of stages in pipelined processing?
7. What is the use of EEPROM?
8. State the hardware needed to implement the LRU in replacement algorithm.
9. What is distributed arbitration?
10. How interrupt requests from multiple devices can be handled?

PART B - (5 x 16 = 80 marks)

11. (a) With examples explain the Data transfer, Logic and Program Control Instructions? (16)

Or

(b) Explain the Working of a Carry-Look Ahead adder. (16)

12. (a)(i) Describe the control unit organization with a separate Encoder and Decoder functions in a hardwired control. (8)

(ii) Generate the logic circuit for the following functions and explain. (8)

$Z_{in} = T_1 + T_6.ADD + T_4.BR+\ldots..$

$END = T_2.ADD + T_5.BR+(T_5.N + T_4.N).BRN+\ldots..$

Or

(b)Write a brief note on nano programming. (16)

13. (a) What are the hazards of conditional branches in pipelines? how it can be resolved? (16)

Or

(b)Explain the super scalar operations with a neat diagram. (16)

14. (a) What is mapping function? What are the ways cache can mapped? (16)

Or

(b)Explain the organization and accessing of data on a Disk. (16)

15. (a) (i)How data transfers can be controlled using handshaking technique ? (8)

(ii)Explain the protocols of USB. (8)

Or

(b)How the parallel port output interface circuit works? (16)

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2012
Fourth Semester
Computer Science and Engineering
CS 2253/141403 / CS43/CS1252 A/10144/ CS404/080250011 -
COMPUTER ARCHITECTURE
(Common to Information Technology)
(Regulation 2008)

Time: 3 hours **Maximum: 100 marks**

Answer ALL questions

PART A -A (10 x 2 = 20 marks)

1. What is SPEC? Specify the formula for SPEC rating.
2. What is relative addressing mode? When is it used?
3. Write the register transfer sequence for storing a word in memory.
4. What is hard-wired control? How is it different from micro-programmed control?
5. What is meant by data and control hazards in pipelining?
6. What is meant by speculative execution?
7. What is meant by an interleaved memory?
8. An address space is specified by 24 bits and the corresponding memory space by 16 bits. How many words are in the (a) virtual memory (b) main memory?
9. Specify the different I/O transfer mechanisms available.
10. What does isochronous data stream means?

PART B - (5 x 16 = 80 marks)

11.(a)(i) What are addressing modes? Explain the various addressing modes with Examples. (8)

(ii) Derive and explain an algorithm for adding and subtracting 2 floating point binary Numbers. (8)

Or

(b)(i) Explain instruction sequencing in detail. (10)

(ii) Differentiate RISC and CISC architectures. (6)

12. (a) (i) With a neat diagram explain the internal organization of a processor. (6)

(ii) Explain how control signals are generated using micro programmed control.

(10)

Or

(b) (i) Explain the use of multiple-bus organization for executing a three-operand Instruction. (8)

(ii) Explain the design of hardwired control unit. (8)

13. (a) (i) Discus the basic concepts of pipelining. (8)

(ii) Describe the data path and control considerations for pipelining. (8)

Or

(b) Describe the techniques for handling data and instruction hazards in pipelining. (16)

14. (a) (i) Explain synchronous DRAM technology in detail. (8)

(ii) In a cache-based memory system using FIFO for cache page replacement, it is found that the cache hit ratio H is low. The following proposals are made for increasing.

(1) Increase the cache page size.

(2) Increase the cache storage capacity.

(3) Increase the main memory capacity.

(4) Replace the FIFO replacement policy by LRU.

Analyse each proposal to determine its probable impact (8)

Or

(b) (i) Explain the various mapping techniques associated with cache memories.(10)

(ii) Explain a method of translating virtual address to physical address. (6)

15. (a) Explain the following:

(i) Interrupt priority schemes. (8)

(ii) DMA. (8)

Or

(b) Write an elaborated note on PCI, SCSI and USB bus standards (16)

B.E / B.Tech DEGREE EXAMINATION, NOV / DEC 2014

Third Semester

Computer Science and Engineering

CS6303- COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1. State Amdahl's Law.[Page No:4]
2. Brief about Relative addressing mode with an example.[Page No:6]
3. Define Little Endian arrangement.[Page No:52]
4. What is DMA?[Page No:222]
5. What is the need for Speculation?[Page No:154]
6. What is Exception?[Page No:148]
7. What is Flynn's Classification?[Page No:154]
8. Brief about Multithreading.[Page No:155]
9. Differentiate Programmed I/O and Interrupt I/O.[Page No:189]
10. What is the purpose of Dirty / Modified bit in cache memory?[Page No:187]

PART B - (5 x 16 = 80 marks)

11. a)(i) Assume a two address format specified as source, destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction. [Page No:50]

(10)

(6) Move (R5) + RO
(7) Add (R5) + ,RO
(8) Move RO , (R5)
(9) Move 16(R5),R3
(10) Add #40,R5.

(ii) Consider the computer with three instruction classes and CPI measurements as given below and Instruction counts for each instruction class for the same program from two different compilers are given. Assume that the computer's clock rate is 4GHZ. Which code sequence will execute faster according to execution time? [Page No:35] (6)

Code from	CPI for this instruction class		
	A	B	C
CPI	1	2	3
Code from	Instruction Count for each class		
	A	B	C
Compiler 1	2	1	2
Compiler 2	4	1	1

Or

b. (i) Explain the components of a computer System. [Page No:11] (8)

(ii) State the CPU performance equations and discuss that factors that affect performance. [Page No:20] (8)

12. (a) i) Multiply the following pair of signed nos. Using Booth's bit-pair recording of the multiplier. A=+13(Multiplicand) and B=-6 (Multiplier). [Page No:66] (10)

ii) Briefly about Carry looks ahead adder.[60] (6)

Or

b)Divide $(12)_{10}$ by $(3)_{10}$ using the Restoring and Non restoring division algorithm with step by step intermediate results and explain.[Page No:74] (16)

13. a) Explain Data path and its control in detail.[Page No:90] (16)

Or

b) What is Hazard? Explain its types with suitable examples. [Page No:126] (16)

14.a) Explain Instruction level Parallel Processing. State the challenges of parallel Processing.[Page No:157] (16)

Or

b) i) Multicore Processors.[Page No:177]

ii) Hardware Multithreading [Page No:172] (16)

15. a (i) Explain mapping functions in cache memory to determine the memory block are placed in Cache. [Page No:199] (8)

ii) Explain in detail about the Bus Arbitration techniques in DMA.[Page No:206] (8)

Or

b)i)Draw different memory address layouts and brief about the technique used to increase the average rate of fetching words from the main memory. (8)

ii) Explain in detail about any two Standard Input and Output Interfaces required to connect the I/O device to the bus. [Page No:206] (8)

B.E / B.Tech DEGREE EXAMINATION, APRIL / MAY 2015

Third Semester

Computer Science and Engineering

CS6303- COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1. List the eight great ideas invented by computer architects.[Page No:8]
2. Distinguish Pipelining from Parallelism.[Page No:6]
3. How overflow occur in subtraction?[Page No:52]
4. What do you mean by sub word parallelism?[Page No:53]
5. What are R-Type instructions?[Page No:86]
6. What is a branch prediction buffer?[Page No:76]
7. Differentiate between Strong scaling and Weak Scaling.[Page No:154]
8. Compare UMA and NUMA multiprocessors.[Page No:156]
9. What is the need to implement memory as a hierarchy?[Page No:186]
10. Point out how DMA can improve I/O speed.[Page No:188]

PART -B (5 X 16 = 80 marks)

11. a) Discuss about the various techniques to represent instructions in a computer system. [Page No:28] (16)

Or

b) What is the need for addressing in a computer system? Explain the different addressing modes with suitable examples. [Page No:42] (16)

12. a) Explain the sequential version of multiplication algorithm and its hardware [Page No:61] (16)

Or

b) Explain how floating point addition is carried out in a Computer system. Give an example for a binary floating point addition. [Page No: 78] (16)

13. a) Explain the different types of pipeline hazards with suitable examples. [Page No:108] (16)

Or

b) Explain in detail how exceptions are handled in MIPS architecture?

[Page No:148](16)

14.a) Discuss about SISD , MIMD, SIMD, SPMD and Vector System

. [Page No:167] (16)

Or

b) What is hardware multithreading? Compare and contrast Fine grained Multi-Threading and Coarse grained Multithreading. [Page No:172] (16)

15. a) Elaborate on the various memory technologies and its relevance.

[Page No:192] (16)

Or

b)What is virtual memory? Explain the steps involved in virtual memory address translation. [Page No:215] (16)

B.E / B.Tech DEGREE EXAMINATION, NOVEMBER/DECEMBER 2015

Third Semester

Computer Science and Engineering

CS6303-COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1. What is Instruction set architecture? [Page No:7]
2. How CPU execution time for a program is calculated?[Page No:5]
3. What are the overflow/underflow conditions for addition and subtraction?[Page No: 53,54]
4. State the representation of double precision floating point number.[Page No:54]
5. What is hazard? What are its types?[Page No:87]
6. What is meant by branch prediction?[Page No:86]
7. What is ILP?[Page No:155]
8. Define a super scalar processor.[Page No:156]
9. What are the various memory technologies?[Page No:192]
10. Define a Hit ratio.[Page No:189]

PART- B (5 X 16 = 80 Marks)

11. (a) Explain in detail the various components of computer system with neat diagram. [Page No:11] (16)

Or

(b) What is an addressing mode? Explain the various addressing modes with suitable examples.[Page No:42] (16)

12. (a) Explain in detail about the multiplication algorithm with suitable example and Diagram.[Page No:61] (16)

Or

(b) Discuss in detail about division algorithm in detail with diagram and examples. [Page No:68] (16)

13. (a) Explain the basic MIPS implementation with necessary multiplexers and control line. [Page No:97] (16)

Or

(b) Explain how the instruction pipeline works? What are the various situations where an instruction pipeline can stall? Illustrate with an example.[Page No:108]

(16)

14. (a) Explain in detail Flynn's classification of parallel hardware.[Page No:167]

(16)

Or

(b) Explain in detail about hardware Multithreading.[Page No:172] (16)

15. (a) What is virtual memory? Explain in detail about how virtual memory is implemented with neat diagram? [Page No: 219] (16)

Or

(b) Draw the typical block diagram of a DMA controller and explain how its is used for direct data transfer between memory and peripherals? [Page No:222]

(16)

B.E / B.Tech DEGREE EXAMINATION, MAY/JUNE 2016

Sixth Semester

Electronics and Communication Engineering

CS6303- COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1. How to represent Instruction in a Computer System?[Page No:28]

2. Distinguish between auto increment and auto decrement addressing mode.

[Page No:7]

3. Define ALU.[Page No:58]

4. What is Subword Parallelism?[Page No: 53]

5. What are the advantages of pipelining?[Page No:89]

6. What is Exception?[Page No:84]

7. State the need for Instruction Level Parallelism.[Page No:155]

8. What is Fine grained Multithreading?[Page No:1172]

9. Define Memory hierarchy.[Page No:189]

10. State the advantages of virtual memory.[Page No:215]

PART – B (5 X 16 = 80 Marks)

11. a) Discuss about the various components of a computer system.[Page No:11] (16)

Or

b) Elaborate the different types of addressing modes with a suitable example.

[Page No:42] (16)

12.a) Explain briefly about floating point addition and Subtraction algorithms.

[Page No:78](16)

Or

b) Define Booth Multiplication algorithm with suitable example.

[Page No:61] (16)

13. a) What is pipelining? Discuss about pipelined data path control. [Page No:108] (16)

Or

b) Briefly explain about various categories of hazards with examples.

[Page No:126] (16)

14. a) Explain in detail about Flynn's Classification. [Page No:167] (16)

Or

b) Write short notes on: (16)

i) Hardware multithreading [Page No:172]

ii) Multicore processors.[Page No:177]

15. a) Define Cache Memory? Explain the various Mapping Techniques associated with cache memories. [Page No:194] (16)

Or

b) Explain about DMA controller, with help of a block diagram.

[PageNo:222] (16)

B.E / B.Tech DEGREE EXAMINATION, NOV/DEC 2016

Sixth Semester

Electronics and Communication Engineering

CS6303- COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1.What is an instruction register?

2.Give the formula for CPU execution time for a program.

3.What is a guard bit and what are the ways to truncate the guard bits?

4.What is arithmetic overflow?

5.What is meant by pipeline bubble?

6.What is a data path?

7.What is instruction level parallelism?

8.What is multithreading?

9.What is meant by address mapping?

10.What is cache memory?

PART – B (5 X 16 = 80 Marks)

11. a) Explain in detail the various components of computer system with neat diagram.

Or

b) Explain the different types of Addressing modes with suitable examples.

12. a)Explain Booth's Algorithm for the multiplication of signed two's complement numbers.

Or

b)Discuss in detail about division algorithm in detail with diagram and examples.

13.a) Why is branch prediction algorithm needed? Differentiate between the static and dynamic techniques.

Or

b) Explain how the instruction pipeline works. What are the various situations where an instruction pipeline can stall?

14 a) Explain in detail about Flynn's classification of parallel hardware.

Or

b) Discuss Shared memory multiprocessor with a neat diagram.

15 a) Discuss DMA controller with block diagram.

Or

b) Discuss the steps involved in the address translation of virtual memory with necessary block diagram.

PART – C (1 X 15 = 15 Marks)

16. a) What is the disadvantage of Ripple carry addition and how it is overcome in carry look ahead adder and draw the logic circuit CLA.

Or

b) Design and explain a parallel priority interrupt hardware for a system with eight interrupt sources.

B.E / B.Tech DEGREE EXAMINATION, APR/MAY 2017

Sixth Semester

Electronics and Communication Engineering

CS6303- COMPUTER ARCHITECTURE

(Regulation 2013)

Time : 3 Hours **Maximum : 100 Marks**

Answer ALL Questions

PART- A (10 X 2 = 20 Marks)

1.List the major components of a computer system.

2.State the need for indirect addressing mode. Give an example.

3.Subtract(11010)2-(10000)2 using 1's complement and 2's complement method.

4.Write the rules to perform addition on floating point numbers.

5.Name the control signals required to perform arithmetic operations.

6.Define Hazard. Give an example for data hazard.

7.What is instruction level parallelism?

8.Distinguish implicit multithreading and explicit multithreading.

9.Define memory interleaving.

10.Summarize the sequence of events involved in handling an interrupt request from a single device.

PART – B (5 X 16 = 80 Marks)

11. a) Explain the important measures of the performance of a computer and derive the basic performance equation.

Or

b) Explain direct, immediate, relative and indexed addressing modes with examples.

12. a) (i) Demonstrate multiplication of two binary numbers with an example.Design an arithmetic element to perform this multiplication. (7)

(ii) Describe non restoring division with an example. (6)

Or

b) (i) Desing an arithmetic element to perform the basic floating point operations. (7)

(ii) What is meant by sub word parallelism? Explain. (6)

13. a) Discuss the modified data path to accommodate pipelined executions with a diagram. (13)

Or

b) (i) Explain the hazards caused by unconditional branching statements. (7)

(ii) Describe operand forwarding in a pipeline processor with a diagram. (6)

14. a) (i) Discuss the challenges in parallel processing with necessary diagrams. (6)

(ii) Explain Flynn's classification of parallel processing with necessary diagrams. (7)

Or

b) Explain the four principal approaches to multithreading with necessary diagrams. (13)

15. a) Explain the different mapping functions that can be applied on cache memories in detail. (13)

Or

b) (i) Explain virtual memory address translation in detail with necessary diagrams. (7)

(ii) What is meant by Direct Memory Access? Explain the use of DMA controllers in a computer system. (6)

PART – C (1 X 15 = 15 Marks)

16. a) (i) Explain mapping functions in cache memory to determine how memory blocks are placed in cache. (8)

(ii) Explain in Detail about the Bus Arbitration techniques in DMA. (7)

Or

b) A pipelined processor uses delayed branch technique. Recommend any one of the following possibility for the design of the processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot. In the second possibility, it has a 6-stage pipeline and two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20% ot the instructions are branch instructions and that an optimizing compiler has an 80% success rate in filling in the single delay slot. For the second alternative, the complier is able to fill the second slot 25% of the time.