COMPUTER ORGANIZATION AND ACHTECTURE

GATE 2023

	Sub Topics	Total	2020	2019	2018	2017- Set-1	2017 Set-2	2016 Set-1	2016 Set-2	2015 Set-1	2015 Set-2	2015 Set-3	2014 Set-1	2014 Set-2	2014 Set-3	Total
Machine instructions and addressing modes	Memory Interfacing	4	1	1	1			1								4
	Machine Instructions(Including Instruction Format) [easy]	12	2		2				4	1	2		1			12
	CPU Organization															
	Addressing Modes	3				3										3
ALU, data-path and control unit	Control Unit and design		1						2							
	Hardwired Control unit															
	Micro programmed control unit															
Instruction pipelining	Pipelining [must for rankers]				2				2		2	1	2		3	
	Performance of a Computer (Speed Up)		2			2		2		2			1			
	RISC and CISC				1											
	Instruction Cycles													2		
	Pipeline dependencies											2				
Memory hierarchy: cache, main memory and secondary storagev [Very easy to understand]	Memory Hierarchy(AMAT)	9	1			1	4				1				2	9
	Cache Memory	7		2					2			1		2		7
	Direct Mapping	4				2	2									4
	Associative Mapping															
	Set-associative Mapping	10	2	1	2				2					3		10
	Cache Replacement policies	4				2							2			4
	Secondary Storage	2								2						2
I/O interface (Interrupt and DMA mode)	I/O Interface	2	1		1											2
	DMA	2						2								2
Total			10	4	9	10	6	5	12	5	5	4	6	7	5	88

Key Points

Focus on memory hierarchy, as every year we are gettting good number of questions on an avargae 3-4 marks are coming from Memory Hierarchy including cache replacement policies

Pipelinig. we are getting 2-3 marks on average in each GATE exam including speedup. dependencies

Instruction format and memory interfacing(Number of memory chips required)

JANUARY 30, 2022

QUANTUM CITY

Nowhere

1. BASIC STRUCTURE OF COMPUTERS

1.1) Computer Types :

Modern computers can be divided roughly into four general categories : 1) Embedded Computer, 2) Server and enterprise Computer, 3) Personal Computer, 4) Supercomputer. Von-Neumann computer belong to SISD class (Single instruction single data stream)

1.2) Functional Unit : (IMACO)

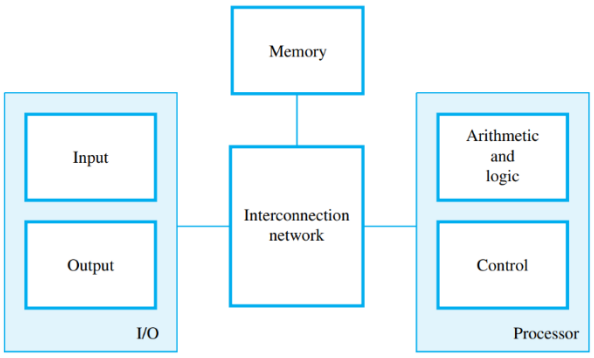


Figure 1.1 Basic functional units of a computer.

1.2.1) Input Unit : Computers accept coded information through input units. The most common input device is the keyboard, touchpad, mouse, joystick, and trackball.

1.2.2) Memory Unit : Two classes primary and secondary

Primary Memory : Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called **words.** To provide easy access to any word in the memory, a distinct address is associated with each word location. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a **random-access memory (RAM).** The time required to access one word is called the **memory access time**.

Cache Memory : As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Secondary Storage : Magnetic disks, optical disks (DVD and CD), and flash memory devices.

1.2.3) Arithmetic and Logic Unit : For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. When operands are brought into the processor, they are stored in high-speed storage elements called **registers**. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

1.2.4) Output Unit : The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a **printer**.

1.2.5) Control Unit : Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer.

1.3) Basic Operational Concepts : (Important as hell)

To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.

A typical instruction might be i) *Load R2, LOC*

- The original contents of location LOC are preserved, whereas those of register R2 are overwritten.
- First, the instruction is fetched from the memory into the processor.
- Next, the operation to be performed is determined by the control unit.
- The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

ii) *Add R4, R2, R3* : Adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

iii) *Store R4, LOC :* This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

- For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

The **instruction register (IR)** holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The

program counter (PC) is another specialized register it contains the memory address of the next instruction to be fetched and executed. **Memory address register (MAR)** holds the address of the location in memory, which contains data, that is required by the current instruction being executed. simply MAR points to the memory location that contains data required. (Not Instruction)

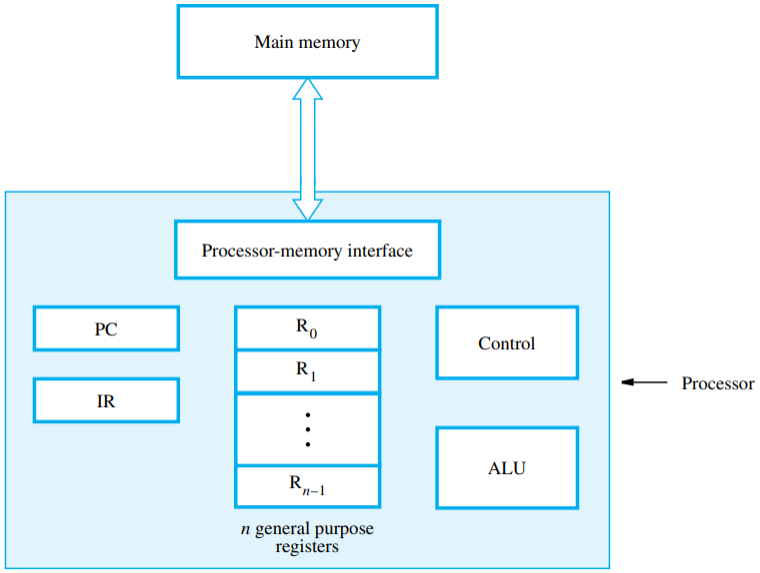


Figure 1.2 Connection between the processor and the main memory.

It is customary to say that the PC points to the next instruction that is to be fetched from the memory. In addition to the IR and PC, Figure 1.2 shows **general-purpose registers** R0 through Rn−1, often called **processor registers**.

Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register. If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

Interrupt : In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an **interrupt-service routine**.

1.4) Floating-Point Numbers : If we use a full word in a 32-bit word length computer to represent a signed integer in 2's-complement representation, the range of values that can be represented is -2^{31} to $+2^{31}-1$. In decimal terms, this range is somewhat smaller than -10^{10} to $+10^{10}$. The same 32-bit patterns can also be interpreted as fractions in the range -1 to $+1-2^{-31}$ if we assume that the implied binary point is just to the right of the sign bit; that is, between bit b_{31} and bit b^{30} at the left end of the 32-bit representation. In this case, the magnitude of the smallest fraction representable is approximately 10^{-10}.

1.5) Performance : The most important measure of the performance of a computer is how quickly it can execute programs.

Parallelism :

Instruction-level Parallelism : The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. This is also known as **Pipelining**.

Multicore Processors : Multiple processing units can be fabricated on a single chip. In technical literature, the term core is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology dual-core, quad-core, and octo-core processors for chips that have two, four, and eight cores, respectively.

Multiprocessors : All processors usually have access to all of the memory in such systems, and the term shared-memory multiprocessor is often used to make this clear.

1.6) History : The first generation, 1945 to 1955; the second generation, 1955 to 1965; the third generation, 1965 to 1975; and the fourth generation, 1975 to the present.

Questions from Carl Hamachar :

1) Explain instruction : *Load R2, LOC*

Answer : • Send the address of the instruction word from register PC to the memory and issue a Read control command.
• Wait until the requested word has been retrieved from the memory, then load it into register IR, where it is interpreted (decoded) by the control circuitry to determine the operation to be performed.
• Increment the contents of register PC to point to the next instruction in memory.
• Send the address value LOC from the instruction in register IR to the memory and issue a Read control command.
• Wait until the requested word has been retrieved from the memory, then load it into register R2.

2) Quantify the effect on performance that results from the use of a cache in the case of a program that has a total of 500 instructions, including a 100-instruction loop that is executed 25 times. Determine the ratio of execution time without the cache to execution time with the cache. This ratio is called the **speedup**.

Assume that main memory accesses require 10 units of time and cache accesses require 1 unit of time. We also make the following further assumptions so that we can simplify calculations in order to easily illustrate the advantage of using a cache:

• Program execution time is proportional to the total amount of time needed to fetch instructions from either the main memory or the cache, with operand data accesses being ignored.

• Initially, all instructions are stored in the main memory, and the cache is empty.

• The cache is large enough to contain all of the loop instructions.

Solution: Execution time without the cache is $T = 400 \times 10 + 100 \times 10 \times 25 = 29{,}000$

Execution time with the cache is $T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7{,}400$

Therefore, the speedup is $T/T_{cache} = 3.92$ ← *Speedup* ☺

Execution time = (Ideal CPI + no of stalls per instruction)*one cycle time

Here no of stalls is nothing but the cycle stalls per instruction. And one cycle time is nothing but time in sec taken by one cycle to complete. Execution time is inversely proportional to performance of program.

2. INSTRUCTION SET ARCHITECTURE (ISA)

2.1) Memory Locations and Addresses :

The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n **bits** is referred to as a **word** of information, and n is called the **word length**.

A unit of 8 bits is called a **byte**. Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct **names or addresses** for each location. It is customary to use numbers from 0 to $2^k - 1$, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2^k addressable locations. The 2^k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 2^{20} (1,048,576). A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations, where 1G is 2^{30}.

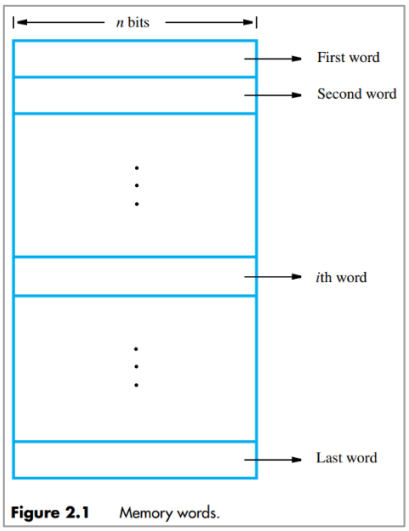


Figure 2.1 Memory words.

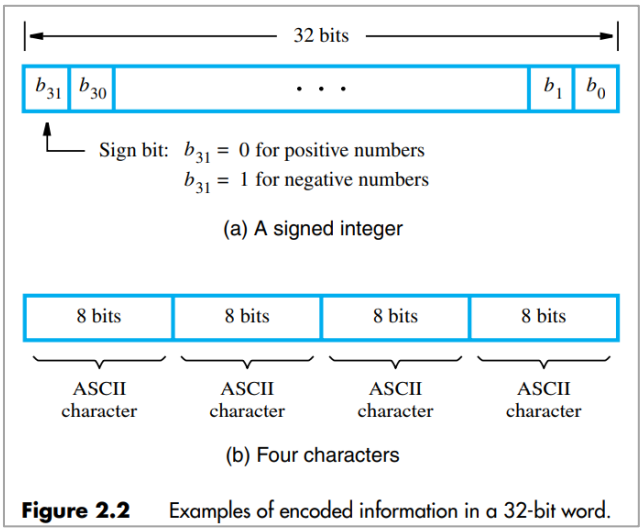


Figure 2.2 Examples of encoded information in a 32-bit word.

2.1.1) Byte Addressability : The term **byte-addressable memory** is used for this assignment. Byte locations have addresses 0, 1, 2,.... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes.

2.1.2) Big-Endian and Little-Endian Assignments : The name **big-endian** is used when lower byte addresses are used for the *more significant bytes* (the leftmost bytes) of the word. Example, 53 in 16bit binary is 0000 0000 0011 0101 so at address 365 : **0000 0000**, at address 366 : **0011 0101**. The name **little-endian** is used for the opposite ordering, where the lower byte addresses are used for the *less significant bytes* (the rightmost bytes) of the word. The same ordering is also used for labeling bits within a byte, that is, b7, b6,..., b0, from left to right.

2.1.3) Word Alignment : We say that the word locations have **aligned** addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4,..., and for a word length of 64 (2^3 bytes), aligned words begin at byte addresses 0, 8, 16,....

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have **unaligned** addresses.

2.2) Memory Operations :

The **Read** operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The **Write** operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

2.3) Instructions and Instruction Sequencing :

A computer must have instructions capable of performing four types of operations:

• Data transfers between the memory and the processor registers

• Arithmetic and logic operations on data
• Program sequencing and control
• I/O transfers, we begin by discussing instructions for the first two types of operations.

2.3.1) Register Transfer Notation :

The expression R2 ← [LOC] means that the contents of memory location LOC are transferred into processor register R2. As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as R4 ← [R2]+[R3] This type of notation is known as **Register Transfer Notation (RTN).**

- ✓ In computer jargon, the words "transfer" and "move" are commonly used to mean "copy". Note that the righthand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

2.3.2) Assembly-Language Notation :

- Load R2, LOC

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten.

- The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3

- Such operations are defined by using **mnemonics**, which are typically abbreviations of the words describing the operations. For example LD for Load, STR or ST for Store.

2.3.3) RISC and CISC Instruction Sets :

There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called **Reduced Instruction Set Computers (RISC).**

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called **Complex Instruction Set Computers (CISC).**

2.3.4) Introduction to RISC Instruction Sets :

Two key characteristics of RISC instruction sets are:
• Each instruction fits in a single word.
• A load/store architecture is used, and Register to register arithmetic operation only

Load instructions are of the form

Load destination, source

or more specifically

Load processor_register, memory_location

The memory location can be specified in several ways. The term addressing modes is used to refer to the different ways in which this may be accomplished. Lets take one example,

The statement C = A + B in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. Hence, the above high-level language statement requires the action C ← [A]+[B] to take place in the computer.

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

```
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
```

We say that Add is a **three-operand**, or a **three-address**, instruction of the form

Add destination, source1, source2

The Store instruction is of the form Store source, destination where the source is a processor register and the destination is a memory location.

2.3.5) Instruction Execution and Straight-Line Sequencing : (Terms of this session are imp)

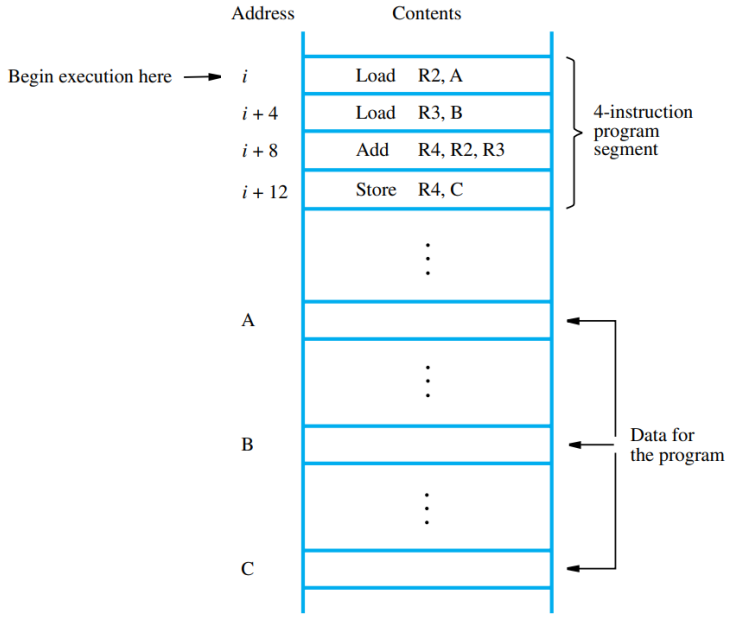


Figure 2.4 A program for C ← [A] + [B].

Let us consider how this program is executed. The processor contains a register called the **program counter (PC)**, which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called **straight-line sequencing**. During the execution of each instruction, the PC is incremented by 4(Byte) to point to the next instruction. Thus, after the Store instruction at location i + 12 is executed, the PC contains the value i + 16, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called **instruction fetch**, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the instruction register (IR) in the processor. At the start of the second phase, called **instruction execute**, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor.

2.3.6) Branching : Consider the task of adding a list of n numbers. The program outlined in Figure 2.5 is a generalization of the program in Figure 2.4. The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2,..., NUMn, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

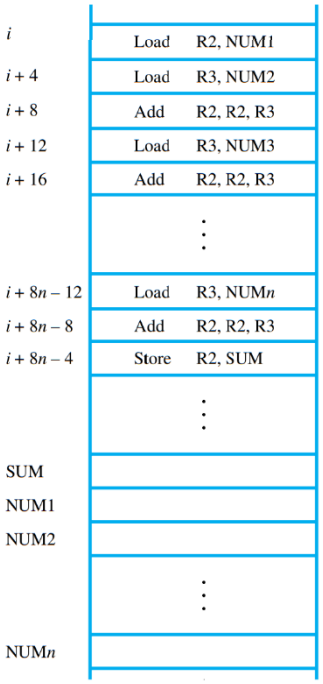


Figure 2.5 A program for adding *n* numbers.

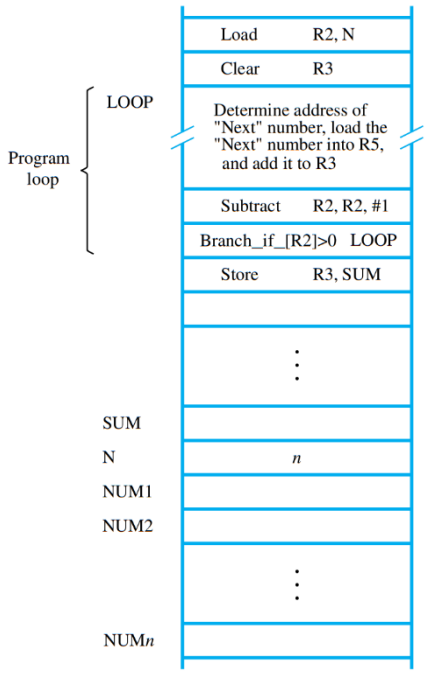


Figure 2.6 Using a loop to add *n* numbers.

We now introduce **branch instructions**. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the **branch target**, instead of the instruction at the location that follows the branch instruction in sequential address order. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed. In the program in Figure 2.6, the instruction Branch_if_[R2]>0 LOOP is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified requirement. For example, the instruction that implements the action Branch_if_[R4]>[R5] LOOP may be written in generic assembly language as Branch_greater_than R4, R5, LOOP or using an actual mnemonic as BGT R4, R5, LOOP.

2.3.7) Generating Memory Addresses : ***How are the addresses specified in LOOP?***

As one possibility, suppose that a processor register, Ri, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability. This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called **addressing modes**. This is the reason we need to study addressing modes.

2.4) Addressing Modes :

The different ways for specifying the locations of instruction operands are known as **addressing modes**.

2.4.1) Implementation of Variables and Constants :

Register mode—The operand is the contents of a processor register; the name of the register is given in the instruction.

Absolute mode—The operand is in a memory location; the address of this location is given explicitly in the instruction.

The instruction Add R4, R2, R3 uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination. The Absolute mode can represent global variables in a program.

A declaration such as Integer NUM1, NUM2, SUM; in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. The Absolute mode is used in the instruction Load R2, NUM1 which loads the value in the memory location NUM1 into register R2.

Table 2.1 RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

- Constants can be represented in assembly language using the Immediate addressing mode. **Immediate mode**—The operand is given explicitly in the instruction. For example, the instruction Add R4, R6, $200_{immediate}$ adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form Add R4, R6, #200.

NOTE : In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an **effective address** (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

2.4.2) Indirection and Pointers : The register acts as a **pointer** to the list, and we say that an item in the list is accessed **indirectly** by using the address in the register. The desired capability is provided by the indirect addressing mode.

Indirect mode—The effective address of the operand is the contents of a register that is specified in the instruction.

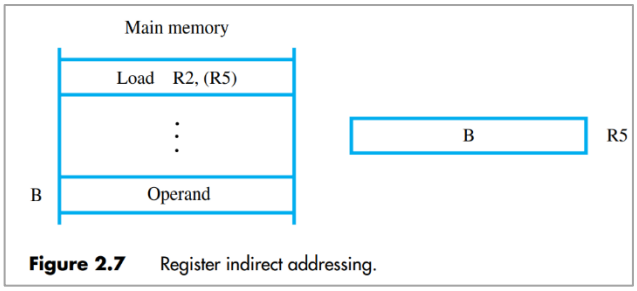


Figure 2.7 Register indirect addressing.

```
        Load    R2, N           Load the size of the list.
        Clear   R3              Initialize sum to 0.
        Move    R4, #NUM1       Get address of the first number.
LOOP:   Load    R5, (R4)        Get the next number.
        Add     R3, R3, R5      Add this number to sum.
        Add     R4, R4, #4      Increment the pointer to the list.
        Subtract R2, R2, #1     Decrement the counter.
        Branch_if_[R2]>0 LOOP   Branch back if not finished.
        Store   R3, SUM         Store the final sum.
```

Figure 2.8 Use of indirect addressing in the program of Figure 2.6.

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as Add R4, R0, #NUM1

It is often the case that Move is provided as a pseudo instruction for the convenience of programmers, but it is actually implemented using the Add instruction.

Consider this example : consider the C-language statement A = *B;
where B is a pointer variable and the '*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

```
Load R2, B
Load R3, (R2)
Store R3, A
```

2.4.3) Indexing and Arrays : The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode—The effective address of the operand is generated by adding a constant value to the contents of a register.

For convenience, we will refer to the register used in this mode as the index register. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as X(Ri) where X denotes a constant signed integer value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by EA = X + [Ri].

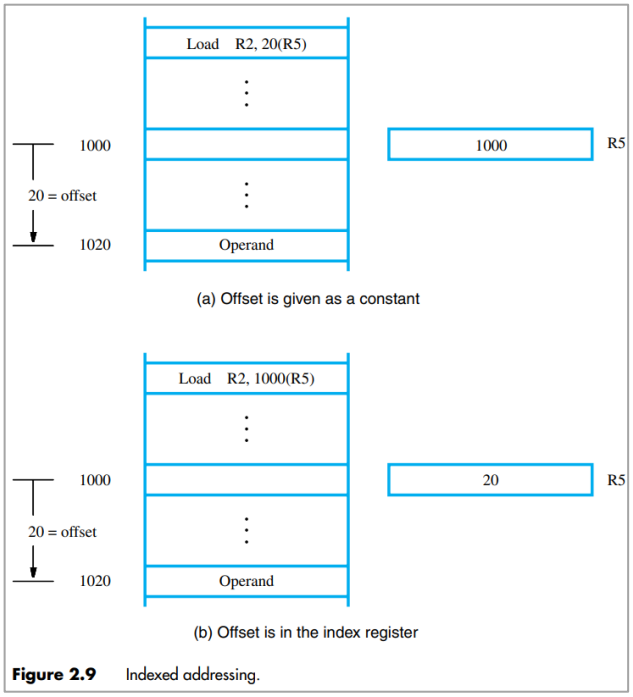


Figure 2.9 Indexed addressing.

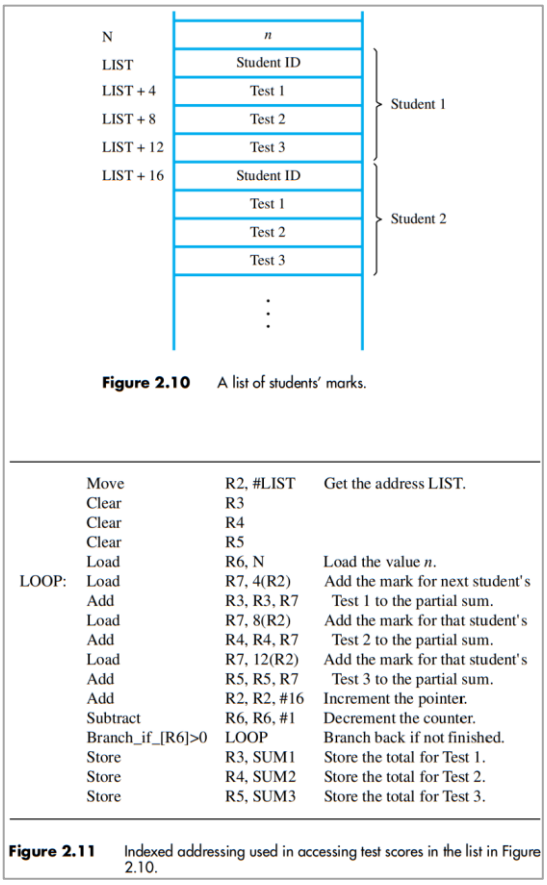


Figure 2.10 A list of students' marks.

```
        Move            R2, #LIST       Get the address LIST.
        Clear           R3
        Clear           R4
        Clear           R5
        Load            R6, N           Load the value n.
LOOP:   Load            R7, 4(R2)       Add the mark for next student's
        Add             R3, R3, R7       Test 1 to the partial sum.
        Load            R7, 8(R2)       Add the mark for that student's
        Add             R4, R4, R7       Test 2 to the partial sum.
        Load            R7, 12(R2)      Add the mark for that student's
        Add             R5, R5, R7       Test 3 to the partial sum.
        Add             R2, R2, #16     Increment the pointer.
        Subtract        R6, R6, #1      Decrement the counter.
        Branch_if_[R6]>0 LOOP           Branch back if not finished.
        Store           R3, SUM1        Store the total for Test 1.
        Store           R4, SUM2        Store the total for Test 2.
        Store           R5, SUM3        Store the total for Test 3.
```

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

Different forms of Index mode :

1) A second register Rj may be used to contain the offset X, in which case we can write the Index mode as (Ri,Rj) The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the **base register**.
2) X(Ri,Rj) In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj.
3) If the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as X(R0).

2.5) Subroutines : In a given program, it is often necessary to perform a particular task many times on different data values. Such a block of instructions is usually called a **subroutine**. For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order.

When a program branches to a subroutine we say that it is **calling** the subroutine. The instruction that performs this branch operation is named a **Call instruction**. The subroutine is said to **return** to the program that called it, and it does so by executing a Return instruction.

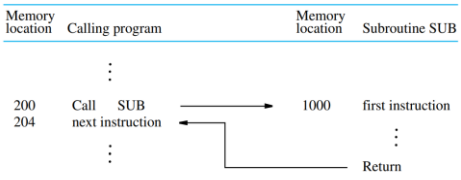
The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed. The way in which a computer makes it possible to call and return from subroutines is referred to as its **subroutine linkage method**. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the **link register**. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register.



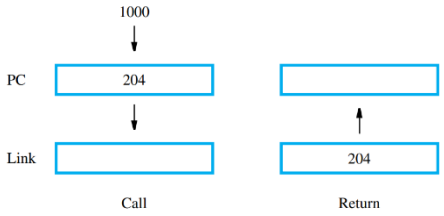


Figure 2.16 Subroutine linkage using a link register.

2.5.1) Subroutine Nesting and the Processor Stack : A common programming practice, called **subroutine nesting**, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, overwriting its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost. **Processor stack** is used for this purpose.

2.5.2) Parameter Passing : When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as **parameter passing.**

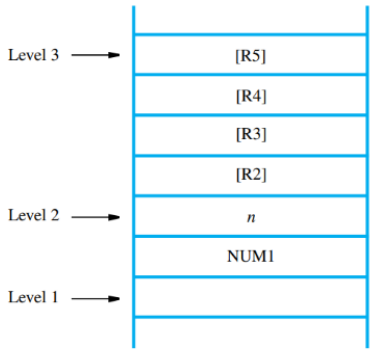


Figure 2.19 Stack contents for the program in Figure 2.18.

Calling program

```
        Load      R2, N          Parameter 1 is list size.
        Move      R4, #NUM1      Parameter 2 is list location.
        Call      LISTADD        Call subroutine.
        Store     R3, SUM        Save result.
        ⋮
```

Subroutine

```
LISTADD: Subtract  SP, SP, #4     Save the contents of
         Store     R5, (SP)       R5 on the stack.
         Clear     R3             Initialize sum to 0.
LOOP:    Load      R5, (R4)       Get the next number.
         Add       R3, R3, R5     Add this number to sum.
         Add       R4, R4, #4     Increment the pointer by 4.
         Subtract  R2, R2, #1     Decrement the counter.
         Branch_if_[R2]>0 LOOP
         Load      R5, (SP)       Restore the contents of R5.
         Add       SP, SP, #4
         Return                   Return to calling program.
```

Figure 2.17 Program of Figure 2.8 written as a subroutine; parameters passed through registers.

Assume top of stack is at level 1 in Figure 2.19.

```
         Move      R2, #NUM1      Push parameters onto stack.
         Subtract  SP, SP, #4
         Store     R2, (SP)
         Load      R2, N
         Subtract  SP, SP, #4
         Store     R2, (SP)
         Call      LISTADD        Call subroutine
                                  (top of stack is at level 2).
         Load      R2, 4(SP)      Get the result from the stack
         Store     R2, SUM          and save it in SUM.
         Add       SP, SP, #8     Restore top of stack
                                  (top of stack is at level 1).
         ⋮

LISTADD: Subtract  SP, SP, #16    Save registers
         Store     R2, 12(SP)
         Store     R3, 8(SP)
         Store     R4, 4(SP)
         Store     R5, (SP)       (top of stack is at level 3).
         Load      R2, 16(SP)     Initialize counter to n.
         Load      R4, 20(SP)     Initialize pointer to the list.
         Clear     R3             Initialize sum to 0.
LOOP:    Load      R5, (R4)       Get the next number.
         Add       R3, R3, R5     Add this number to sum.
         Add       R4, R4, #4     Increment the pointer by 4.
         Subtract  R2, R2, #1     Decrement the counter.
         Branch_if_[R2]>0 LOOP
         Store     R3, 20(SP)     Put result in the stack.
         Load      R5, (SP)       Restore registers.
         Load      R4, 4(SP)
         Load      R3, 8(SP)
         Load      R2, 12(SP)
         Add       SP, SP, #16    (top of stack is at level 2).
         Return                   Return to calling program.
```

Figure 2.18 Program of Figure 2.8 written as a subroutine; parameters passed on the stack.

2.5.3) The Stack Frame : Locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program. Such space is called a **stack frame**. In addition to the stack pointer SP, it is useful to have another pointer register, called the **frame pointer (FP)**, for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine.

The parameters can be accessed by using addresses 4(FP), 8(FP),.... The local variables can be accessed by using addresses −4(FP), −8(FP),.... The contents of FP remain fixed throughout the execution of the subroutine, unlike the stack pointer SP, which must always point to the current top element in the stack.

Thus, the first three instructions executed in the subroutine are

```
Subtract SP, SP, #4
Store FP, (SP)
Move FP, SP
```

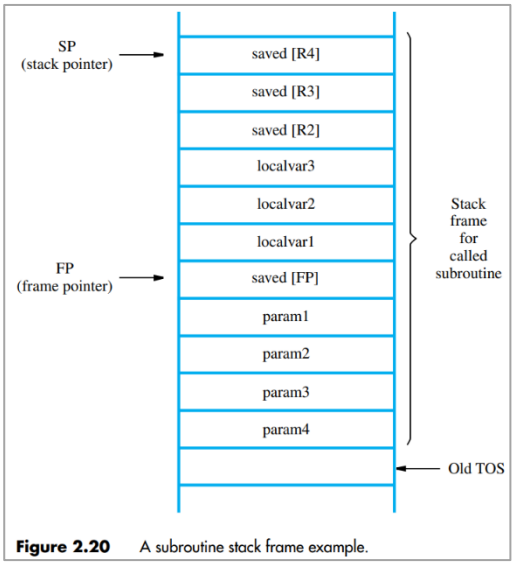


Figure 2.20 A subroutine stack frame example.

Stack Frames for Nested Subroutines : When nested subroutines are used, it is necessary to ensure that the return addresses are properly saved. The appropriate place for saving this return address is within the stack frame for Subroutines called by main program. Nested means Main program calls SUB1 then SUB1 calls SUB2 and so on.

2.6) Additional Instructions :

So far, we have introduced the following instructions: Load, Store, Move, Clear, Add, Subtract, Branch, Call, and Return. We will see more just revise previous instruction plz.

2.6.1) Logic Instructions :

1) AND operation : And R4, R2, R3

Computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4. An immediate form of this instruction may be

And R4, R2, #Value

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

Example : Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. If it is, then a conditional branch to FOUNDZ is to be made. FOUNDZ is call leads to any operation.

Answer : And R2, R2, #0xFF

Move R3, #0x5A

Branch_if_[R2]=[R3] FOUNDZ

And instruction clears all bits in the leftmost three-character positions of R2 to zero, leaving the rightmost character unchanged. This is the result of using an immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the left. The Move instruction loads the hex value 5A into R3. Since both R2 and R3 have 0s in the leftmost 24 bits, the Branch instruction compares the remaining character at the right end of R2 with the binary representation for the character Z, and causes a branch to FOUNDZ if there is a match.

2.8.2) Shift and Rotate Instruction :

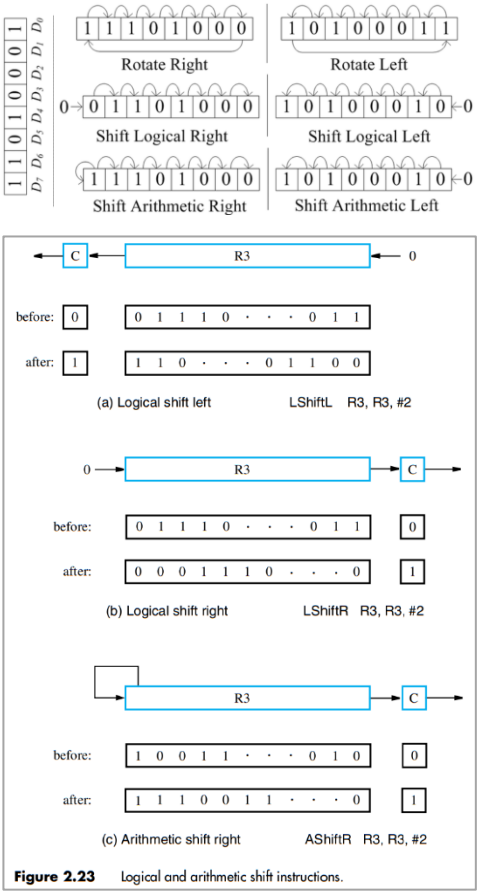


Figure 2.23 Logical and arithmetic shift instructions.

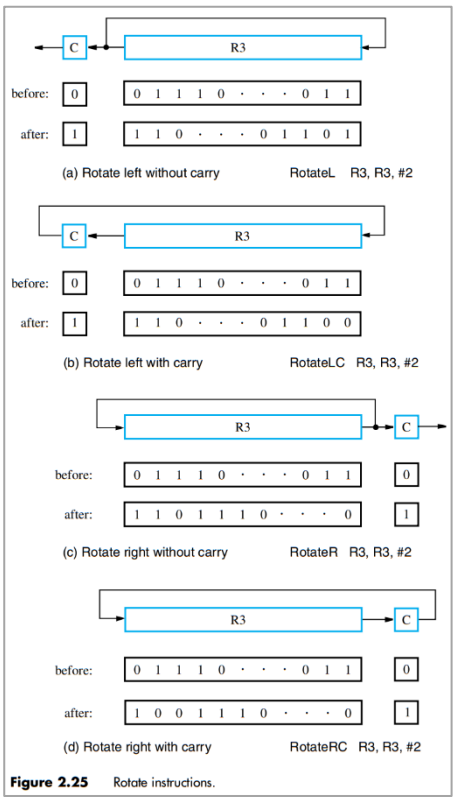


Figure 2.25 Rotate instructions.

Digit-Packing Example : Suppose that two decimal digits represented in ASCII code are located in the memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in **packed-BCD format.**

```
Move        R2, #LOC      R2 points to data.
LoadByte    R3, (R2)      Load first byte into R3.
LShiftL     R3, R3, #4    Shift left by 4 bit positions.
Add         R2, R2, #1    Increment the pointer.
LoadByte    R4, (R2)      Load second byte into R4.
And         R4, R4, #0xF  Clear high-order bits to zero.
Or          R3, R3, R4    Concatenate the BCD digits.
StoreByte   R3, PACKED    Store the result.
```

Figure 2.24 A routine that packs two BCD digits into a byte.

Note : Arithmetic left shift and logical shift is exactly same.

Instruction set simply means instruction only. Like we have op code, address, register length, etc. So if one these changes instruction set will also get affected.

In Relative addressing Mode is same as basic index because PC value is address and An is also address and both are getting added into some constant.

Table C.1 ColdFire addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Absolute Short	Value	EA = Sign Extended WValue
Absolute Long	Value	EA = Value
Register	Rn	EA = R_n that is, Operand = [R_n]
Register Indirect	(An)	EA = [A_n]
Autoincrement	(An)+	EA = [A_n]; Increment A_n
Autodecrement	−(An)	Decrement A_n; EA = [A_n]
Basic index	WValue(An)	EA = WValue + [A_n]
Full index	BValue(An, Rk)	EA = BValue + [A_n] +[R_k]
Basic relative	WValue(PC)	EA = WValue + [PC]
Full relative	BValue(PC, Rk)	EA = BValue + [PC] + [R_k]

EA = effective address
Value = a number given either explicitly or represented by a label
BValue = an 8-bit Value
WValue = a 16-bit Value
A_n = an address register
R_n = an address register or a data register

2.6.3) Multiplication and Division :

Multiply Rk, Ri, Rj

Divide Rk, Ri, Rj <- Special One (This instruction means Rk ← [Rj]/[Ri])

Questions on addressing Mode :

1) Base addressing - Position independent (By changing the value in base register, location of address can be changed), position independent means body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address.
Indexed addressing - Array
Stack addressing - Reentranecy (Whenever code happens to be used again, address need not be the same)
Implied addressing - Accumulator (If an address is not specified, it is assumed/implied to be the Accumulator)

2) A certain processor supports only the immediate and the direct addressing modes. Which of the following programming language features cannot be implemented on this processor?

A. Pointers
B. Arrays
C. Records
D. Recursive procedures with local variable

Answer : Pointer access requires indirect addressing which can be simulated with indexed addressing or register indirect addressing but not with direct and immediate addressing. An array and record access needs a pointer access. So, options (A), (B) and (C) cannot be implemented on such a processor. Now, to handle recursive procedures we need to use stack. A local variable inside the stack will be accessed as * which is nothing but a pointer access and requires indirect addressing.

3) In autoincrement mode (An)+ does not mean after each time the address incremented by 1 but incremented by the size of data item, here data item can be int, char so according to that increment has been done. Same as for autodecrement.

4) Displacement Mode :-

Similar to index mode, except instead of a index register a base register will be used. Base register contains a pointer to a memory location. An integer (constant) is also referred to as a displacement. The address of the operand is obtained by adding the contents of the base register plus the constant. The difference between index mode and displacement mode is in the number of bits used to represent the constant. When the constant is represented a number of bits to access the memory, then we have index mode. Index mode is more appropriate for array accessing; displacement mode is more appropriate for structure (records) accessing.

3. BASIC INPUT/OUTPUT

3.1) Accessing I/O Devices : (Programmed I/O)

Some addresses in the address space of the processor are assigned to I/O locations, rather than to the main memory. These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers. It is customary to refer to them as **I/O registers**. Since the I/O devices and the memory share the same address space, this arrangement is called **memory-mapped I/O**. The clock cycle is the time between two adjacent pulses of the oscillator that sets the temp of the computer processor.

With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of a register in an input device, the instruction Load R2, DATAIN reads the data from the DATAIN register and loads them into processor register R2. Similarly, the instruction Store R2, DATAOUT sends the contents of register R2 to location DATAOUT, which is a register in an output device.

3.1.1) I/O Device Interface : An I/O device is connected to the interconnection network by using a circuit, called the **device interface**, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behavior of the device. These **data, status**, and **control registers** are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor.

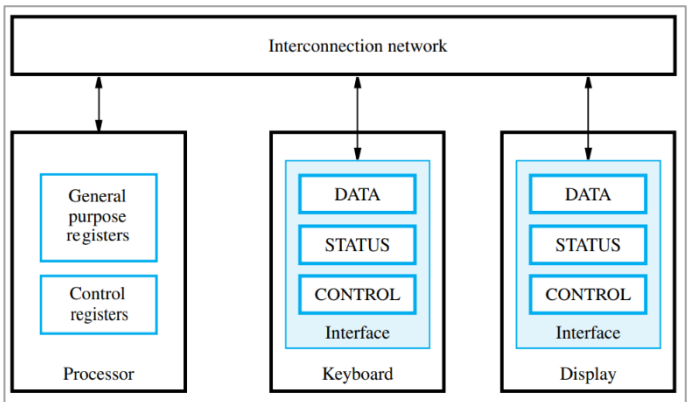


Figure 3.2 The connection for processor, keyboard, and display.

3.1.2) Program-Controlled I/O :

Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as **program-controlled I/O**. In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them. Programmed I/O does not make use of interrupt.

One solution to this problem involves a **signaling protocol**. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way. The processor waits for a signal indicating that a key has been pressed and that a binary code that represents the corresponding character is available in an I/O register associated with the keyboard. Then the processor proceeds to read that code. The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer. We will assume that ASCII code (presented in Table 1.1) is used, in which each character code occupies one byte.

Some terminologies which you have to remember :

KBD_DATA	The address label of an 8-bit register that holds the generated character.
KIN	A signal indicating that a key has been pressed provided by setting to 1 a flip-flop
KBD_STATUS	KIN is the part of an eight-bit status register KBD_STATUS

DISP_DATA	Used to receive characters from the processor
DOUT	A signal indicate that it is ready to receive the next character; this can be done by using a status flag
DISP_STATUS	DOUT is a one bit in status register.

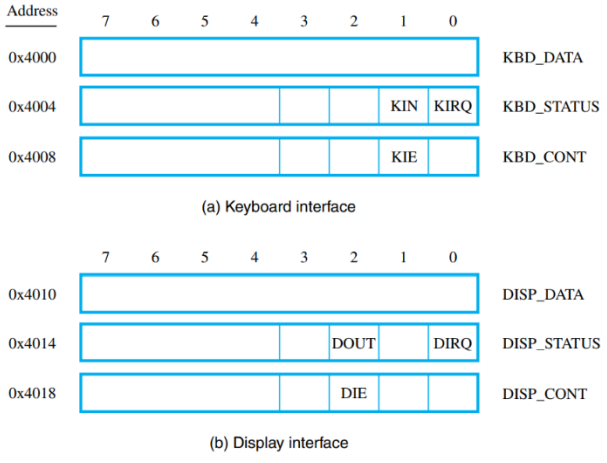


Figure 3.3 Registers in the keyboard and display interfaces.

Detail Explanation :

Let us consider the details of the input process. When a key is pressed, the keyboard circuit places the ASCII-encoded character into the KBD_DATA register. At the same time, the circuit sets the KIN flag to 1. Meanwhile, the processor is executing the I/O program which continuously checks the state of the KIN flag. When it detects that KIN is set to 1, it transfers the contents of KBD_DATA into a processor register. Once the contents of KBD_DATA are read, KIN must be cleared to

0, which is usually done automatically by the interface circuit. If a second character is entered at the keyboard, KIN is again set to 1 and the process repeats. When the processor reads the status flag to determine its state, we say that the processor **polls the I/O device**. This is known as **POLLING.** The desired action can be achieved by performing the operations:

```
READWAIT    Read the KIN flag
            Branch to READWAIT if KIN = 0
            Transfer data from KBD_DATA to R5
```

An analogous process takes place when characters are transferred from the processor to the display. When DOUT is equal to 1, the display is ready to receive a character. Under program control, the processor monitors DOUT, and when DOUT is equal to 1, the processor transfers an ASCII-encoded character to DISP_DATA. The transfer of a character to DISP_DATA clears DOUT to 0. When the display device is ready to receive a second character, DOUT is again set to 1. This can be achieved by performing the operations:

```
WRITEWAIT   Read the DOUT flag
            Branch to WRITEWAIT if DOUT = 0
            Transfer data from R5 to DISP_DATA
```

```
        Move          R2, #LOC        Initialize pointer register R2 to point to the
                                        address of the first location in main memory
                                        where the characters are to be stored.
        MoveByte      R3, #CR         Load ASCII code for Carriage Return into R3.
READ:   LoadByte      R4, KBD_STATUS  Wait for a character to be entered.
        And           R4, R4, #2      Check the KIN flag.
        Branch_if_[R4]=0  READ
        LoadByte      R5, KBD_DATA    Read the character from KBD_DATA
                                        (this clears KIN to 0).
        StoreByte     R5, (R2)        Write the character into the main memory and
        Add           R2, R2, #1       increment the pointer to main memory.
ECHO:   LoadByte      R4, DISP_STATUS Wait for the display to become ready.
        And           R4, R4, #4      Check the DOUT flag.
        Branch_if_[R4]=0  ECHO
        StoreByte     R5, DISP_DATA   Move the character just read to the display
                                        buffer register (this clears DOUT to 0).
        Branch_if_[R5]≠[R3] READ      Check if the character just read is the
                                        Carriage Return. If it is not, then
                                        branch back and read another character.
```

Figure 3.4 A RISC-style program that reads a line of characters and displays it.

```
        Move          R2, #LOC        Initialize pointer register R2 to point to the
                                        address of the first location in main memory
                                        where the characters are to be stored.
READ:   TestBit       KBD_STATUS, #1  Wait for a character to be entered
        Branch=0      READ            in the keyboard buffer KBD_DATA.
        MoveByte      (R2), KBD_DATA  Transfer the character from KBD_DATA into
                                        the main memory (this clears KIN to 0).
ECHO:   TestBit       DISP_STATUS, #2 Wait for the display to become ready.
        Branch=0      ECHO
        MoveByte      DISP_DATA, (R2) Move the character just read to the display
                                        buffer register (this clears DOUT to 0).
        CompareByte   (R2)+, #CR      Check if the character just read is CR
                                        (carriage return). If it is not CR, then
        Branch≠0      READ            branch back and read another character.
                                        Also, increment the pointer to store the
                                        next character.
```

Figure 3.5 A CISC-style program that reads a line of characters and displays it.

3.2) Interrupts :

In the examples in Figures 3.4 and 3.5, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an **interrupt request** to the processor.

If you have time consider this example : Consider a task that requires continuous extensive computations to be performed and the results to be displayed on a display device. The displayed results must be updated every ten seconds. The ten-second intervals can be determined by a simple timer circuit, which generates an appropriate signal. The processor treats the timer circuit as an input device that produces a signal that can be interrogated. If this is done by means of polling, the processor will waste considerable time checking the state of the signal. A better solution is to have the timer circuit raise an interrupt request once every ten seconds. In response, the processor displays the latest results.

The task can be implemented with a program that consists of two routines, COMPUTE and DISPLAY. The processor continuously executes the COMPUTE routine. When it receives an interrupt request from the timer, it suspends the execution of the COMPUTE routine and executes the DISPLAY routine which sends the latest results to the display device. Upon completion of the DISPLAY routine, the processor resumes the execution of the COMPUTE routine. Since the time needed to send the results to the display device is very small compared to the ten-second interval, the processor in effect spends almost all of its time executing the COMPUTE routine.

The routine executed in response to an interrupt request is called the **interrupt-service routine**, which is the DISPLAY routine in our example. After execution of the interrupt-service routine, the processor returns to instruction i + 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i + 1, must be put in temporary storage in a known location. The **return address** must be saved either in a designated general-purpose register or on the processor stack.

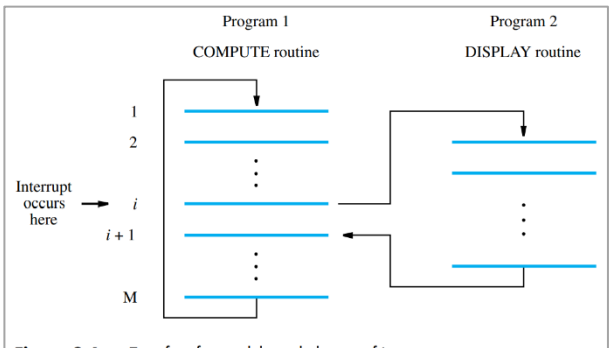


Figure 3.6 Transfer of control through the use of interrupts.

The processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This can be accomplished by means of a special control signal, called **interrupt acknowledge**, which is sent to the device through the interconnection network.

Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called **interrupt latency**. In some applications, a long interrupt latency is unacceptable. For these reasons, the amount of information saved automatically by the processor when an interrupt request is accepted should be kept to a minimum.

Another interesting approach is to provide duplicate sets of processor registers. In this case, a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers. The duplicate registers are sometimes called the **shadow registers**. The concept of interrupts is used in operating systems and in many control applications where processing of certain routines must be accurately timed relative to external events. The latter type of application is referred to as **real-time processing**. Interrupts increases CPU utilization because it is usually done to execute multitasking.

3.2.1) Enabling and Disabling Interrupts :

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.
3. Interrupts are disabled by clearing the IE bit in the PS to 0.
4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

3.2.2) Handling Multiple Devices : Device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor determine which device is requesting an interrupt?

2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?

3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?

4. How should two or more simultaneous interrupt requests be handled?

The information needed to determine whether a device is requesting an interrupt is available in its status register. When the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the IRQ bit. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system. The first device encountered with its IRQ bit set to 1 is the device that should be serviced. An appropriate subroutine is then called to provide the requested service.

The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we describe next.

- *Vectored Interrupts* : To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. **Vectored Interrupt are those interrupt whose interrupt service routine (ISR) addresses are stored in predefined location.**

  The term vectored interrupts refer to interrupt-handling schemes based on this approach.

  A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as **interrupt vectors**, and they are said to constitute the **interrupt-vector table**. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors.

- *Interrupt Nesting* : Interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. An interrupt request from a high-priority device should be accepted while the processor is servicing a request from a lower-priority device. This is also called **Daisy Chaining**, here we create chain in which highest priority device will be place first in chain and lowest in last.

  A **multiple-level priority** organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. If nested interrupts are allowed, then each interrupt service routine must save on the stack the saved contents of the program counter and the status register. This has to be done before the interrupt-service routine enables nesting by setting the IE bit in the status register to 1. INTR is non vectored interrupt.

Types of Interrupts:

1. *External interrupts*
2. *Internal Interrupts*
3. *Software Interrupts*

External interrupts (Asynchronous) come from input-output (1/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are 1/0 device requesting transfer of data, 1/0 device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation.

The 8085 has five hardware interrupts (1) TRAP (2) RST 7.5 (3) RST 6.5 (4) RST 5.5 (5) INTR

Internal interrupts (Synchronous) arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called ***traps***. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event.

Note:-

1. If the program is rerun, the internal interrupts will occur in the same place each time.
2. External/internal interrupts are initiated from signals that occur in the H/W of the CPU
3. A **software interrupt** is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.
4. Processor checks for interrupts not before executing new instruction it checks for interrupts before fetching instruction.
5. Loop instruction **can be** interrupted till they complete.
6. If bus (DMA) request and interrupts arrive simultaneously then bus request will be served first.

See the whole working mechanism of Status Register :

The KIRQ bit is set to 1 if an interrupt request has been raised, but not yet serviced. The keyboard may raise interrupt requests only when the interrupt-enable bit, KIE, in its control register is set to 1. Thus, when both KIE and KIN bits are equal to 1, an interrupt request is raised and the KIRQ bit is set to 1. Similarly, the DIRQ bit in the status register of the display interface indicates whether an interrupt request has been raised. Bit DIE in the control register of this interface is used to enable interrupts.

3.2.4) Processor Control Registers : To deal with interrupts it is useful to have some other control registers. The status register, PS, includes the interrupt-enable bit, IE, in addition to other status information. Recall that the processor will accept interrupts only when this bit is set to 1. The IPS register is used to automatically save the contents of PS when an interrupt request is received and accepted. At the end of the interrupt-service routine, the previous state of the processor is automatically restored by transferring the contents of IPS into PS. Since there is only one register available for storing the previous status information, it becomes necessary to save the contents of IPS on the stack if nested interrupts are allowed. The IENABLE register allows the processor to selectively respond to individual I/O devices. The IPENDING register indicates the active interrupt requests.

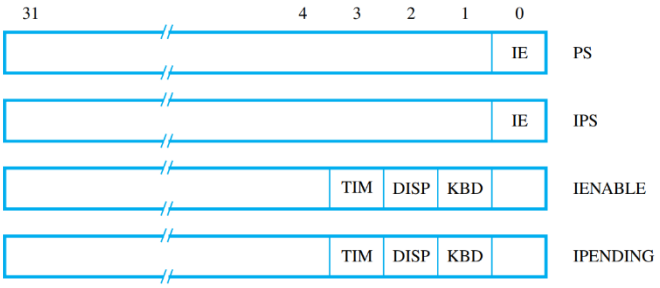


Figure 3.7 Control registers in the processor.

In a RISC-style processor, the special instructions may be of the type MoveControl R2, PS which loads the contents of the program status register into register R2, and MoveControl IENABLE, R3 which places the contents of R3 into the IENABLE register. These instructions perform transfers between control and general-purpose registers.

3.2.6) Exceptions : The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception.

Difference Interrupts and Exceptions

▪ Exceptions are caused by software executing instructions. Exceptions are unplanned software interrupts.
Eg : a page fault, or an attempted write to read only page. An expected exception is 'trap', unexpected is a "fault".
▪ Interrupts are caused by hardware devices
Eg : device finishes I/O, timer fires.

Debugging : System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities: trace mode and breakpoints.

Summery of imp :

In this module you have studied 3 flavors of dealing with IO:

Programmed I/O (PIO) : For every 'data item', you need to execute an instruction on the CPU and the CPU will wait for the instruction to complete. So, if you have a lot of data (so many 'data items') that needs to be send/received, the CPU will be blocked for a long time. Programmed I/O is not suitable for high speed data transfer because

i) Programmed I/O mode does not support synchronous mode of data transmission that is a requirement for many high speed peripherals like disk.
ii) For transfer of every word between I/O device and memory, a set of machine instruction has to be executed by CPU.

Interrupt driven I/O : So the transfer starts on the CPU, and on completion of the transfer, an interrupt is send. This means that the CPU is free for other tasks. But for every data item you still need to interact with the CPU. So better than #1, but still not great.

DMA : a DMA controller (either a central one or one on the storage device) will take care for sending multiple data items without any involvement of the CPU. So, prevent the CPU from being a bottleneck.

Questions on Interrupts :

1) A device with data transfer rate 10 KB/sec is connected to a CPU. Data is transferred byte-wise. Let the interrupt overhead be 4 microsec. The byte transfer time between the device interface register and CPU or memory is negligible. What is the minimum performance gain of operating the device under interrupt mode over operating it under program-controlled mode?

Answer *:* In Programmed I/O, the CPU issues a command and waits for I/O operations to complete. So here, CPU will wait for 1 sec to transfer 10KB of data. overhead in programmed I/O = 1 sec. In Interrupt mode, data is transferred word by word (here word size is 1 byte as mentioned in question "Data is transferred byte-wise"). So, to transfer 1 byte of data overhead is 4 microsec. Thus, to transfer 10KB of data overhead is = 40000 microsec.
Performance gain = 1sec / 40000microsec = 25.

2. BASIC PROCESSING UNIT

In this chapter we focus on the processing unit, which executes machine-language instructions and coordinates the activities of other units in a computer. We examine its internal structure and show how it performs the tasks of fetching, decoding, and executing such instructions. The processing unit is often called the **central processing unit (CPU)**. A processor can have a **pipelined** organization where the execution of an instruction is started before the execution of the preceding instruction is completed. Another approach, known as **superscalar** operation, is to fetch and start the execution of several instructions at the same time.

2.1) Some Fundamental Concepts :

To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is IR ← [[PC]]
2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is PC ← [PC] + 4
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the **instruction fetch phase**. Performing the operation specified in the instruction constitutes the **instruction execution phase**.

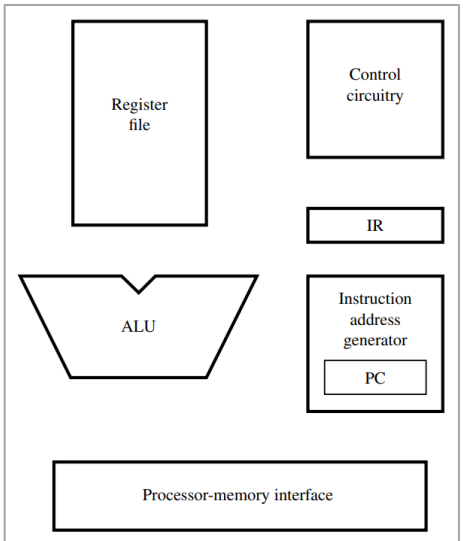


Figure 5.1 Main hardware components of a processor.

Data Processing Hardware : A typical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. Figure 5.2 illustrates this structure.

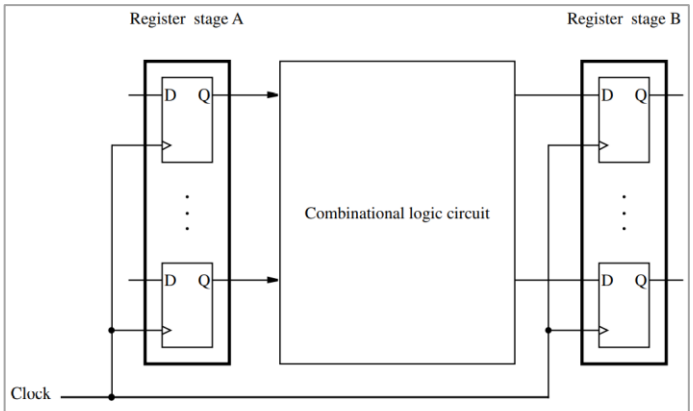


Figure 5.2 Basic structure for data processing.

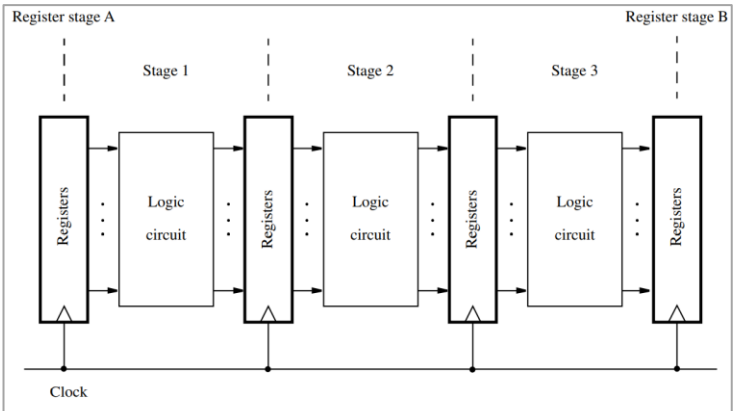


Figure 5.3 A hardware structure with multiple stages.

2.2) Instruction Execution : Let us now examine the actions involved in fetching and executing instructions.

4.2.1) Load Instructions : Consider the instruction Load R5, X(R7) which uses the Index addressing mode to load a word of data from memory location X + [R7] into register R5.

In the discussion that follows, we will assume that the processor has five hardware stages, which is a commonly used arrangement in RISC-style processors. Execution of each instruction is divided into five steps, such that each step is carried out by one hardware stage.

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of register R7 in the register file.
3. Compute the effective address.
4. Read the memory source operand.
5. Load the operand into the destination register, R5.

4.2.2) Arithmetic and Logic Instructions : A typical instruction of this type is Add R3, R4, R5 It requires the following steps:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of source registers R4 and R5.
3. Compute the sum [R4] + [R5].
4. Load the result into the destination register, R3.

To this end, the Add instruction should be extended to five steps, patterned along the steps of the Load instruction.

1. Fetch the instruction and increment the program counter.

2. Decode the instruction and read registers R4 and R5.
3. Compute the sum [R4] + [R5].
4. No action.
5. Load the result into the destination register, R3.

The five-step sequence is suitable for all Load and Store instructions, because the addressing modes that can be used in these instructions are special cases of the Index mode. Most RISC-style processors provide one general-purpose register, usually register R0, that always contains the value zero.

Step	Action
1	Fetch an instruction and increment the program counter.
2	Decode the instruction and read registers from the register file.
3	Perform an ALU operation.
4	Read or write memory data if the instruction involves a memory operand.
5	Write the result into the destination register, if needed.

Figure 5.4 A five-step sequence of actions to fetch and execute an instruction.

2.3) Hardware Components :

We now examine the components in Figure 5.1 to see how they may be organized in the multi-stage structure of Figure 5.3.

4.3.1) Register File : General-purpose registers are usually implemented in the form of a register file, which is a small and fast memory block. It consists of an array of storage elements, with access circuitry that enables data to be read from or written into any register. The register file has two address inputs that select the two registers to be read. These inputs are connected to the fields in the IR that specify the source registers, so that the required registers can be read. The inputs and outputs of any memory unit are often called input and output ports. A memory unit that has two output ports is said to be **dual-ported**.

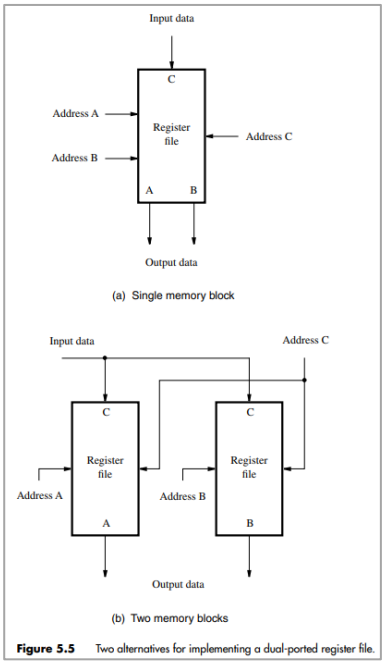


Figure 5.5 Two alternatives for implementing a dual-ported register file.

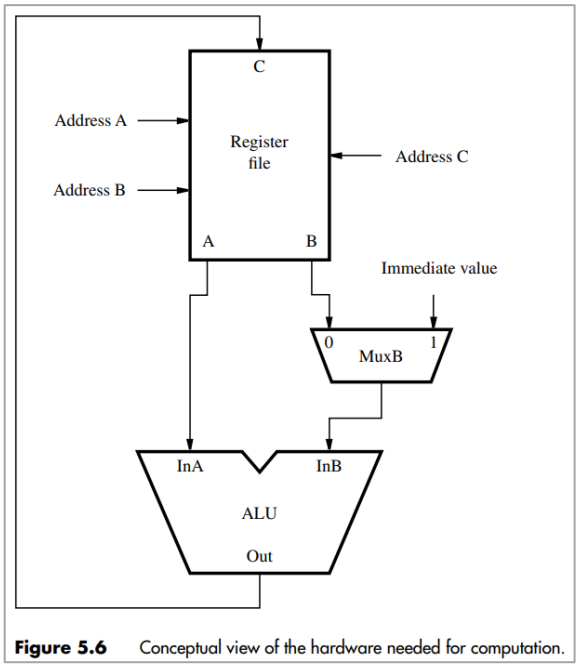


Figure 5.6 Conceptual view of the hardware needed for computation.

4.3.2) ALU : The arithmetic and logic unit is used to manipulate data. It performs arithmetic operations such as addition and subtraction, and logic operations such as AND, OR, and XOR. Output A is connected directly to the first input of the ALU, InA, and output B is connected to a multiplexer, MuxB. The multiplexer selects either output B of the register file or the immediate value in the IR to be connected to the second ALU input, InB. The output of the ALU is connected to the data input, C, of the register file so that the results of a computation can be loaded into the destination register.

4.3.3) Datapath : An instruction is fetched in step 1 by hardware stage 1 and placed into the IR. It is decoded, and its source registers are read in step 2. The information in the IR is used to generate the control signals for all subsequent steps. Therefore, the IR must continue to hold the instruction until its execution is completed. It is necessary to insert registers between stages. The hardware in the figure is often referred to as the **datapath.**

Working **:**

Data read from the register file are placed in registers RA and RB. Register RA provides the data to input InA of the ALU. Multiplexer MuxB forwards either the contents of RB or the immediate value in the IR to the ALU's second input, InB. The ALU constitutes stage 3, and the result of the computation it performs is placed in register RZ.

Multiplexer MuxY in Figure 5.8 selects register RZ to transfer the result of the computation to RY. The

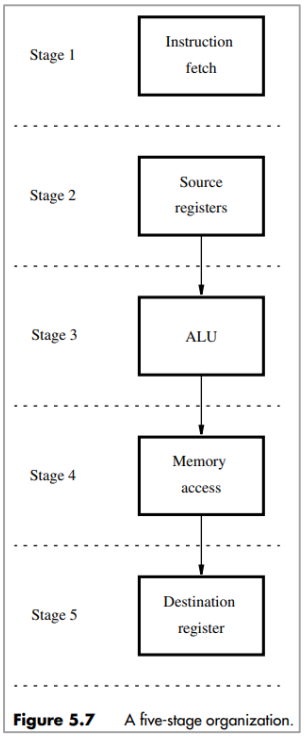


Figure 5.7 A five-stage organization.

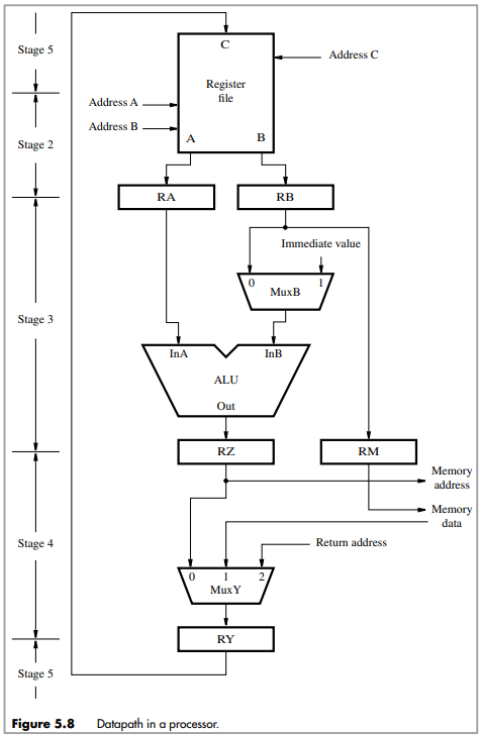


Figure 5.8 Datapath in a processor.

contents of RY are transferred to the register file in step 5 and loaded into the destination register. For this reason, the register file is in both stages 2 and 5. It is a part of stage 2 because it contains the source registers and a part of stage 5 because it contains the destination register.

For Load and Store instructions, the effective address of the memory operand is computed by the ALU in step 3 and loaded into register RZ. From there, it is sent to the memory, which is stage 4. In the case of a Load instruction, the data read from the memory are selected by multiplexer MuxY and placed in register RY, to be transferred to the register file in the next clock cycle. For a Store instruction, data are read from the register file, which is part of stage 2, and placed in register RB. Since memory access is done in stage 4, another inter-stage register is needed to maintain correct data flow in the multi-stage structure. Register RM is introduced for this purpose. The data to be stored are moved from RB to RM in step 3, and from there to the memory in step 4. No action is taken in step 5 in this case.

The subroutine call instructions introduced in Section 2.7 save the return address in a general-purpose register, which we call LINK for ease of reference. Similarly, interrupt processing requires a return address to be saved, as described in Section 3.2. Assume that another general-purpose register, IRA, is used for this purpose. Both of these actions require the contents of the program counter to be sent to the register file. For this reason, multiplexer MuxY has a third input through which the return address can be routed to register RY, from where it can be sent to the register file.

4.3.4) Instruction Fetch Section :

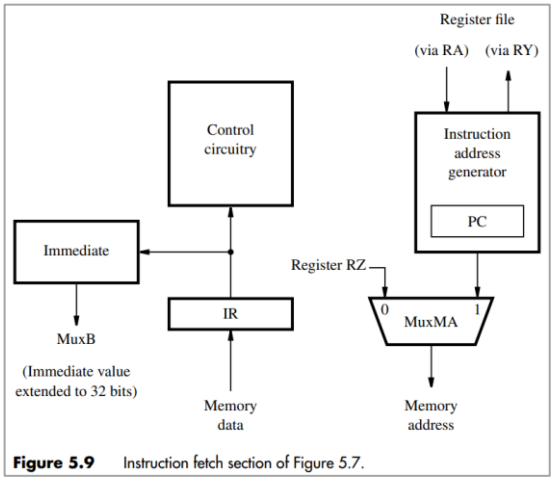


Figure 5.9 Instruction fetch section of Figure 5.7.

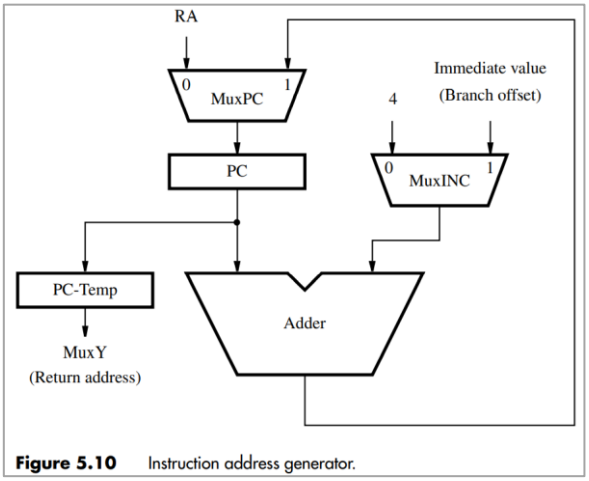


Figure 5.10 Instruction address generator.

2.4) Instruction Fetch and Execution Steps :

We now examine the process of fetching and executing instructions in more detail, using the datapath in Figure 5.8. Consider again the instruction Add R3, R4, R5. After the instruction has been fetched from the memory and placed in the IR, the source register addresses are available in fields IR_{31-27} and IR_{26-22}. These two fields are connected to the address inputs for ports A and B of the register file. then it perform operation according to table given below :

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R4], RB ← [R5]
3	RZ ← [RA] + [RB]
4	RY ← [RZ]
5	R3 ← [RY]

Figure 5.11 Sequence of actions needed to fetch and execute the instruction: Add R3, R4, R5.

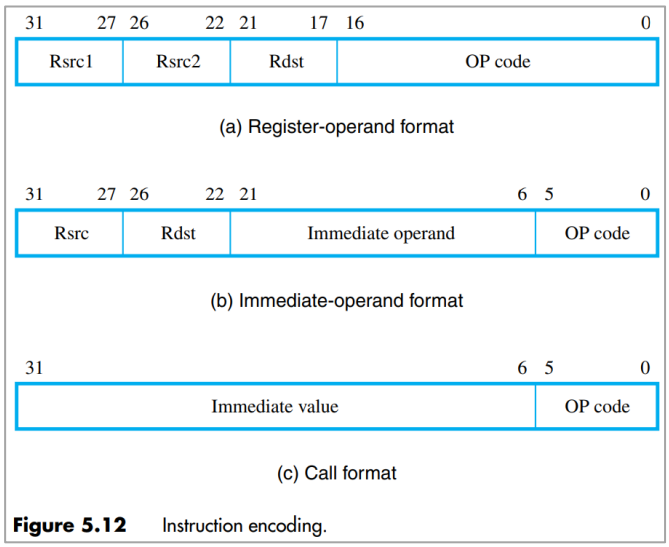


Figure 5.12 Instruction encoding.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R7]
3	RZ ← [RA] + Immediate value X
4	Memory address ← [RZ], Read memory, RY ← Memory data
5	R5 ← [RY]

Figure 5.13 Sequence of actions needed to fetch and execute the instruction: Load R5, X(R7).

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R8], RB ← [R6]
3	RZ ← [RA] + Immediate value X, RM ← [RB]
4	Memory address ← [RZ], Memory data ← [RM], Write memory
5	No action

Figure 5.14 Sequence of actions needed to fetch and execute the instruction: Store R6, X(R8).

4.4.1) Branching : Whenever an instruction is fetched, the processor increments the PC by 4 to point to the next word. This execution pattern continues until a branch or subroutine call instruction loads a new address into the PC. Subroutine call instructions also save the return address, to be used when returning to the calling program. One way to maximize the use of the

pipeline, is to find an instruction that can be safely executed whether the branch is taken or not, and execute that instruction. This is also known as **delayed branching**.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R5], RB ← [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.16 Sequence of actions needed to fetch and execute the instruction: Branch_if_[R5]=[R6] LOOP.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.15 Sequence of actions needed to fetch and execute an unconditional branch instruction.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R9]
3	PC-Temp ← [PC], PC ← [RA]
4	RY ← [PC-Temp]
5	Register LINK ← [RY]

Figure 5.17 Sequence of actions needed to fetch and execute the instruction: Call_Register R9.

4.4.2) Waiting for Memory : Assume that the processor-memory interface circuit generates a signal called **Memory Function Completed (MFC)**. It asserts this signal when a requested memory Read or Write operation has been completed. Most of the time, the requested information is found in the cache, so the MFC signal is generated quickly, and the step is completed in one clock cycle. When an access involves the main memory, the MFC response is delayed, and the step is extended to several clock cycles.

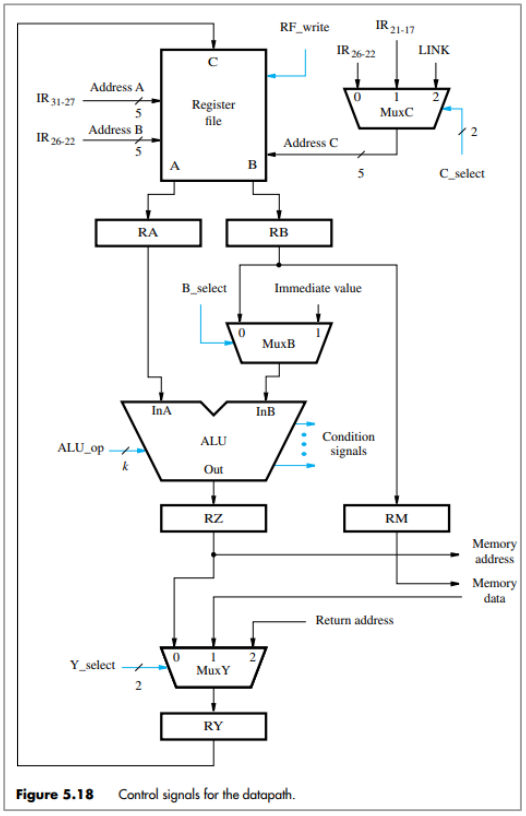


Figure 5.18 Control signals for the datapath.

2.5) Control Signals : The operation of the processor's hardware components is governed by **control signals**.

In each clock cycle, the results of the actions that take place in one stage are stored in inter-stage registers, to be available for use by the next stage in the next clock cycle. Since data are transferred from one stage to the next in every clock cycle, inter-stage registers are always enabled.

The role of the multiplexers is to select the data to be operated on in any given stage.

Figure No 5.18 Details :

The register file has three 5-bit address inputs, allowing access to 32 general-purpose registers. Two of these inputs, Address A and Address B, determine which registers are to be read. They are connected to fields IR_{31-27} and IR_{26-22} in the instruction register. The third address input, Address C, selects the destination register, into which the input data at port C are to be written. Multiplexer MuxC selects the source of that address. We have assumed that three-register instructions use bits IR_{21-17} and other instructions use IR_{26-22} to specify the destination register, as in Figure 5.12. The third input of the multiplexer is the address of the link register used in subroutine linkage instructions. New data are loaded into the selected register only when the control signal RF_write is asserted.

ALU : The operation performed by the ALU is determined by a k-bit control code, ALU_op, which can specify up to 2k distinct operations, such as Add, Subtract, AND, OR, and XOR.

Figure No 5.19 Details : Two signals, MEM_read and MEM_write are used to initiate a memory Read or a memory Write operation. When the requested operation has been completed, the interface asserts the MFC signal. The instruction register has a control signal, IR_enable, which enables a new instruction to be loaded into the register. During a fetch step, it must be activated only after the MFC signal is asserted.

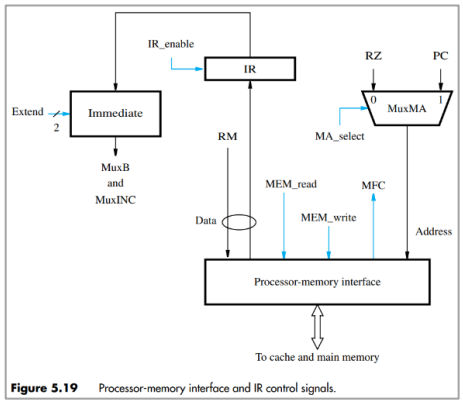


Figure 5.19 Processor-memory interface and IR control signals.

Figure No 5.20 Details : The INC_select signal selects the value to be added to the PC, either the constant 4 or the branch offset specified in the instruction. The PC_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC_enable control signal is activated.

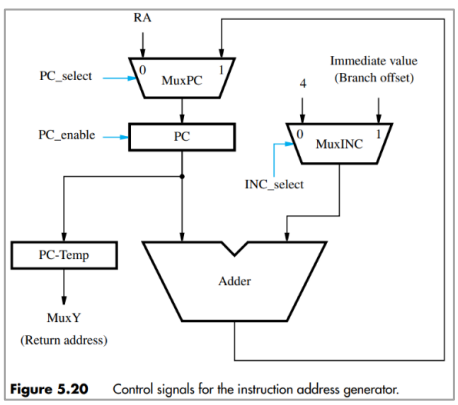


Figure 5.20 Control signals for the instruction address generator.

2.6) Hardwired Control :

We now examine how the processor generates the control signals that cause these actions to take place in the correct sequence and at the right time. There are two basic approaches : hardwired control and microprogrammed control. Hardwired control is discussed in this section.
The setting of the control signals depends on:

- Contents of the step counter
- Contents of the instruction register
- The result of a computation or a comparison operation
- External input signals, such as interrupt requests

The instruction decoder interprets the OP-code and addressing mode information in the IR and sets to 1 the corresponding INSi output. During each clock cycle, one of the outputs T1 to T5 of the step counter is set to 1 to indicate which of the five steps involved in fetching and executing instructions is being carried out.

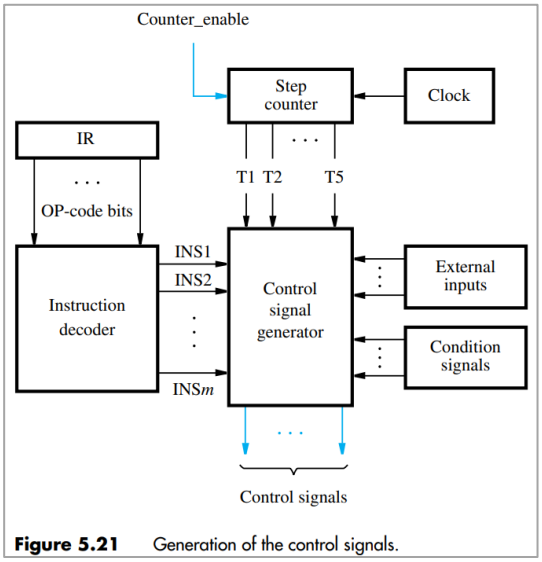


Figure 5.21 Generation of the control signals.

2.7) CISC-Style Processors :

A traditional approach to the implementation of the Interconnect is to use **buses**. A bus consists of a set of lines to which several devices may be connected, enabling data to be transferred from any one device to any other. A logic gate that sends a signal over a bus line is called a **bus driver**. The bus driver is a special type of logic gate called a **tri-state gate**. It has a control input that turns it on or off. When turned on, the gate places a logic signal of 0 or 1 on the bus, according to the value of its input. When turned off, the gate is electrically disconnected from the bus.

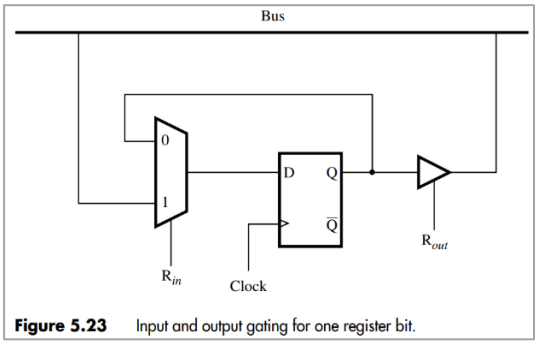


Figure 5.23 Input and output gating for one register bit.

Microprogrammed Control :

Instead of employing circuits as in RISC, it is possible to use a "software" approach, in which the desired setting of the control signals in each step is determined by a program stored in a special memory. The logic of control unit is specified in memory.

- **Control Memory** :

The function of the **control unit** in a digital computer is to initiate sequences of micro-operations. During any given time, certain micro-operations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a **control word**. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variable are stored in memory is called a **microprogrammed control unit**. Control memory can be a **read-only memory (ROM)**.

- **Microinstruction (Control word) :**

Flag + control signal(microoperation) + address (# of microinstruction)

Each word in control memory contains within it a **microinstruction**. The microinstruction specifies one or more **microoperations** for the system.

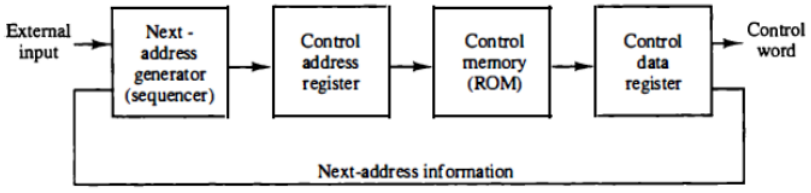
- **Microprogram :**

A sequence of microinstructions constitutes a microprogram. That means microoperation makes microinstruction which in turn makes microprogram.

Note :-

A memory that is part of a control unit is referred to as a control memory. Control signal means microoperation.
Width of control memory = size of control word

Figure 7-1 Microprogrammed control organization.



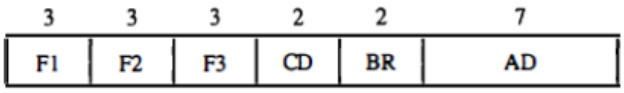
A computer that employs a microprogrammed control unit will have two separate memories: **a main memory** and a **control memory**. The main memory is available to the user for storing the programs. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered

by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations.

- **Control address register :**

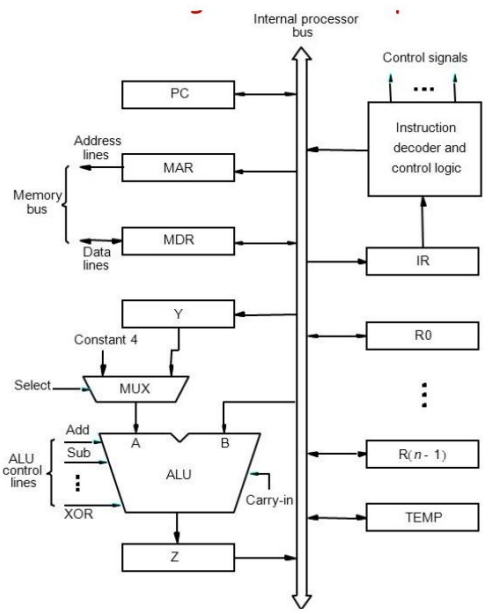
The control memory is a ROM, within which all control information is permanently stored. The **control memory address register (CAR)** specifies the address of the microinstruction, and the control data register (CDR) holds the **microinstruction** read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.

- **Microinstruction code format :**



Microinstruction code format

F1, F2, F3 micro-operations fields, CD: Condition for branching, BR: Branch Field, AD: Address field.



Datapath in which the arithmetic and logic unit (ALU) and all the registers are interconnected through a single common bus, which is internal to the processor and should not be confused with the **external bus** that connects the processor to the memory and I/O devices. The **data** and **address lines** of the external memory bus are shown above connected to the internal processor bus via the **memory data register**, MDR, and the **memory address register**, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. **The control lines of the memory bus are connected to the instruction decoder and control logic block**. Registers from R0 to Rn-1 may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as **index registers** or **stack pointers**. The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. The instruction **decoder** and **control logic unit** is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:
1. Enable the output of register R1 out by setting R1out to 1. This place the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting R4in to 1. This loads data from the processor bus into register R4.

Horizontal Microprogramming (decoded binary formate)	Vertical Microprogramming
Each microinstruction specifies many different microoperations to be performed in parallel	Each microinstruction specifies a single micro-operation to be performed.
Wide control memory word (more bits)	Control word width is narrow
Optimizing performance as result of fast execution	Slower execution
Efficient hardware utilization	Limited ability to express parallelism
Little encoding of control information	Control signals encoded into function codes – needs to be decoded using decider circuits (more hardware)
More difficult to program, but more flexible	Optimize programming

Note :-

1) Hardwired is faster than microprogram control unit.
2) For smaller program hardwired is faster.
3) MAR holds the memory location of data that needs to be accessed. When reading from memory, data addressed by MAR is fed into the MDR (memory data register) and then used by the CPU. When writing to memory, the CPU writes data from MDR to the memory location whose address is stored in MAR. PC also holds address of instruction but it then gives that address to MAR. Then PC increment and points to next instruction or store address of next instruction meanwhile address stored in MAR is being proceeded. Then instruction is decoded in IR.

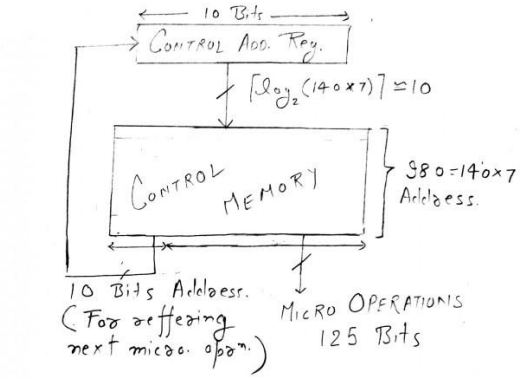
4) Data transfer between a microprocessor and an I/O device is usually faster in memory-mapped-I/O scheme than in I/O-mapped -I/O scheme. Because in I/O mapped – I/O control signals are there which will take more time.

Problems :

1) Consider a CPU where all the instructions require 7 clock cycles to complete execution. There are 140 instructions in the instruction set. It is found that 125 control signals are needed to be generated by the control unit. While designing the horizontal microprogrammed control unit, single address field format is used for branch control logic. What is the minimum size of the control word and control address register?

125, 7 *135, 7*
125, 10 *135, 10*

Answer : Each microinstruction in an instruction require one cycle, so that's why 140 is multiplied with 7 to get total no of microinstructions which are 980. these 980 micro instructions will be stored in control memory (rom or ram) so to uniquely identify each of them we require 10 bits,



<u>microinstruction is divided into 3 parts : **flag microoperation nextAddress**</u>

since nothing mentioned about flags take it 0 - these flags tell about branch and non-branch microinstruction

microoperation → 125 control signal mentioned are nothing but microoperation and since we are using horizontal microinstruction these 125 bits will not be encoded so take it 125

next address → 10 bits from those 140 instructions we calculated are next address bits

total size of control word = 0+125+10 = 135

now control address register size is equal to next address field bits = 10 bits

so answer is → 135, 10

NOTE : Control address register holds the address of micro instruction in control memory we want to fetch and to fetch this micro instruction we use next address field which will send its address to control address register and will get the micro instruction, So register size should be same as next address field.

2) Consider a CPU where all the instruction takes 10 cycles to complete execution. There are total 245 instruction in a instruction set. It is found that 2148 control signals are needed to be generated by control unit. While designing the vertical micro-programmed control, single address field format is used for control logic. The size of control memory in bytes is _____.

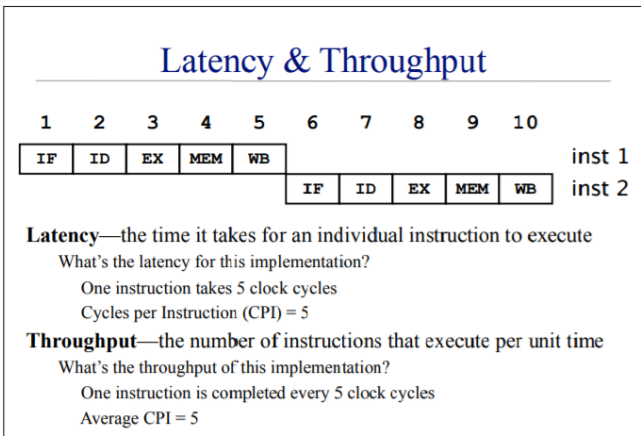
Answer : see above diagram and understand the logic behind the question. Question asks us about the size of control memory. First we know that 1microinstruction takes 1 cycle. Here 245 instruction are given so each will total cycle taken = 2450 which requires 12 bits. 2148 control signals are given means 2148 microoperation is given and vertical programming means we have to use decoder it will take 12 bits. Total 24 bit in one microinstruction. Microinstruction is control word so 1 control word = 24 bits = 3 bytes. And total of 2450 microinstruction is given means we have 2450*3 = 7350bytes.

3) Find out the width of the control memory of a horizontal microprogrammed control unit, given the following specifications:

16 control lines for the processor consisting of ALU and 7 registers.
Conditional branching facility by checking 4 status bits.
Provision to hold 128 words in the control memory.

Answer :

Width of the control memory = size of a control word
Control word = condition bits + control signal bits + next address
Condition bits = 4 ➔ ln4 = 2 bit
128 words = 7 bit
In the case of horizontal microprogramming, there is 1-bit for each control signal. It is given that there are 16 control signals.
Length of control word = 2 + 16 + 7 = 25 bits.



Consider a processor with a fixed 16-bit instruction length, 16 registers, and the following 3 instruction classes:

Class A: Two registers.

Class B: One register.

Class C: One register, one 8-bit immediate value.

In all cases we assume that the encoding space for opcodes is fully utilized and there exist at least one instruction of each type. The minimum number of opcodes that can be encoded on this machine is _____

Class C	Opcode (4-bits) 0000 to 1110	Register (4-bits)	Immediate Value(8-bits)	15 opcodes
Class A	Opcode (8-bits) 11110000 to 11111110	Register (4-bits)	Register (4-bits)	15 opcodes
Class B	Opcode (8-bits) 11111111	0000 to 1111	Register (4-bits)	16 opcodes

4)

Answer : 46

3. PIPELINING

3.1) Basic Concept — The Ideal Case :

One way to improve performance is to arrange the hardware so that more than one operation can be performed at the same time. Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as *an assembly-line operation*. While one group of workers is installing the engine on one automobile, another group is fitting a body on the chassis of a second automobile, and yet another group is preparing a new chassis for a third automobile. In write stage we write values into register not in memory or not in cache.

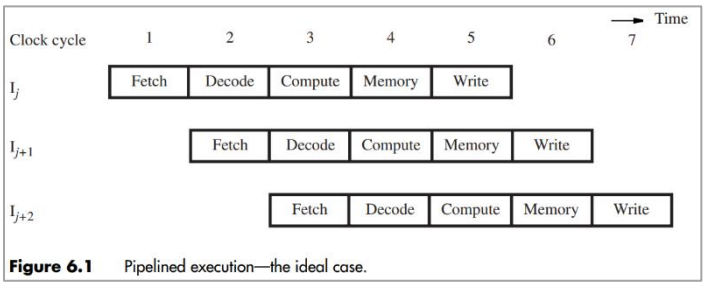


Figure 6.1 Pipelined execution—the ideal case.

3.2) Pipeline Organization :

Information such as register addresses, immediate data, and the operations to be performed must be carried through the pipeline as each instruction proceeds from one stage to the next. This information is held in **interstage buffers**. These include registers RA, RB, RM, RY, and RZ in Figure 5.8, the IR and PC-Temp registers in Figures 5.9 and 5.10, and additional storage. The interstage buffers are used as follows:

- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
- Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder.
- Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage. In the case of a write access to memory, buffer B3 holds the data to be written.
- Interstage buffer B4 feeds the Write stage with a value to be written into the register file.

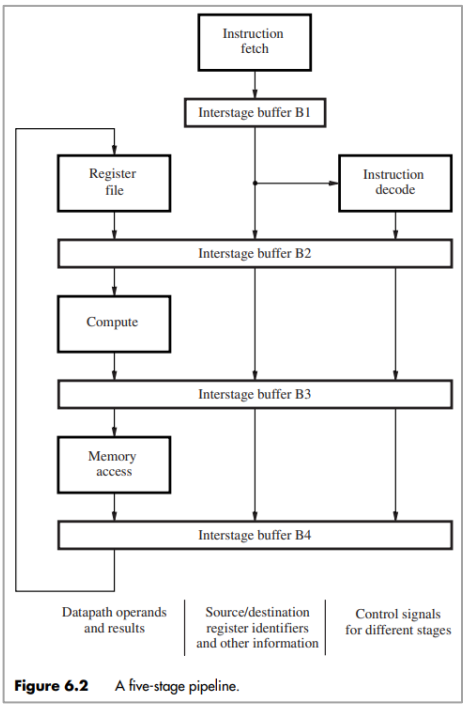


Figure 6.2 A five-stage pipeline.

3.3) Pipelining Issues :

Consider the case of two instructions, Ij and Ij+1, where the destination register for instruction Ij is a source register for instruction Ij+1. The result of instruction Ij is not written into the register file until cycle 5, but it is needed earlier in cycle 3 when the source operand is read for instruction Ij+1. If execution proceeds as shown in Figure 6.1, the result of instruction Ij+1 would be incorrect because the arithmetic operation would be performed using the old value of the register in question. To obtain the correct result, it is necessary to wait until the new value is written into the register by instruction Ij. Hence, instruction Ij+1 cannot read its operand until cycle 6, which means it must be **stalled** in the Decode stage for three cycles. While instruction Ij+1 is stalled, instruction Ij+2 and all subsequent instructions are similarly delayed. New instructions cannot enter the pipeline, and the total execution time is increased. Any condition that causes the pipeline to stall is called a **hazard**. We have just described an example of a **data hazard**, where the value of a source operand of an instruction is not available when needed. The performance of a pipeline processor suffers if the pipeline stages have different delays.

3.4) Data Dependencies :

Consider the two instructions in Figure 6.3:

Add R2, R3, #100

Subtract R9, R2, #30

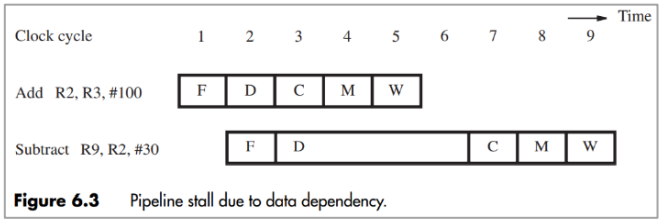


Figure 6.3 Pipeline stall due to data dependency.

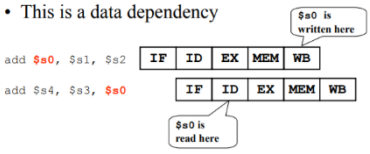
The destination registers R2 for the Add instruction is a source register for the Subtract instruction. There is a **data dependency** between these two instructions, because register R2 carries data from the first instruction to the second. The Subtract instruction is stalled for three cycles to delay reading register R2 until cycle 6 when the new value becomes available. This is called **DATA hazards**.

Pipeline Hazards

- Structural hazards
  - » Instructions in different stages need the same resource, eg, memory
- Data hazards
  - » data not available to perform next operation
- Control hazards
  - » data not available to make branch decision

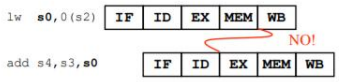
Data Hazards

- When an instruction depends on the results of a previous instruction still in the pipeline
- This is a data dependency



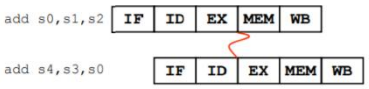
Another data hazard

- What if the first instruction is lw?
- s0 isn't known until after the MEM stage
  - » We can't forward back into the past
- Either **stall** or **reorder** instructions



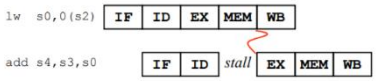
Solution: Forwarding

- The value of $s0 is known internally after cycle 3 (after the first instruction's EX stage)
- The value of $s0 isn't needed until cycle 4 (before the second instruction's EX stage)
- If we **forward** the result there isn't a stall



Stall for `lw` hazard

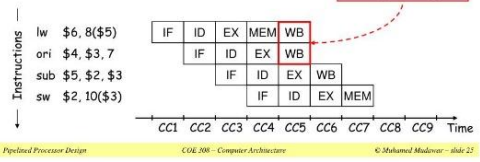
- We can stall for one cycle, but we hate to stall

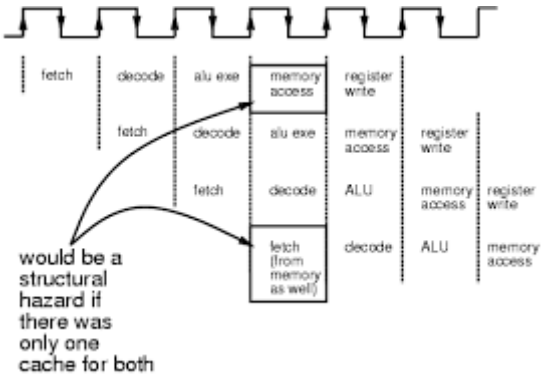


Structural Hazards

- Problem
  - Attempt to use the same hardware resource by two different instructions during the same cycle
- Example
  - Writing back ALU result in stage 4
  - Conflict with writing load data in stage 5

Structural Hazard
Two instructions are attempting to write the register file during same cycle





The structural hazards are minimized using a hardware technique is used called **renaming**. The renaming mechanism states that it splits the memory into two independent sub-modules to store instruction and data separately.

We now explain the stall in more detail.

The control circuit must first recognize the data dependency when it decodes the Subtract instruction in cycle 3 by comparing its source register identifier from interstage buffer B1 with the destination register identifier of the Add instruction that is held in interstage buffer B2. Then, the Subtract instruction must be held in interstage buffer B1 during cycles 3 to 5. Meanwhile, the Add instruction proceeds through the remaining pipeline stages. In cycles 3 to 5, as the Add instruction moves ahead, control signals can be set in interstage buffer B2 for an implicit NOP (No-operation) instruction that does not modify the memory or the register file. Each NOP creates one clock cycle of idle time, called a **bubble**, as it passes through the Compute, Memory, and Write stages to the end of the pipeline.

5.4.1) Operand Forwarding : Pipeline stalls due to data dependencies can be alleviated through the use of **operand forwarding**. Consider previous add subtract example. When the ALU completes the operation for the Add instruction. This value is loaded into register RZ in Figure 5.8, which is a part of interstage buffer B3. Rather than stall the Subtract instruction, the hardware can forward the value from register RZ to where it is needed in cycle 4, which is the ALU input. The arrow shows that the ALU result from cycle 3 is used as an input to the ALU in cycle 4. But it fails when there is load instruction. See above 5th diagram.

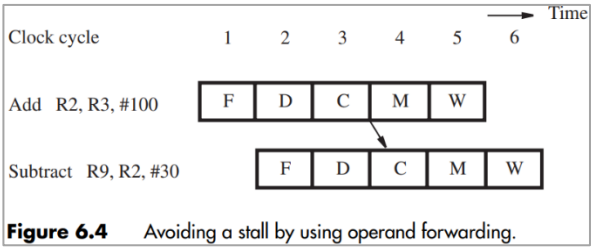


Figure 6.4 Avoiding a stall by using operand forwarding.

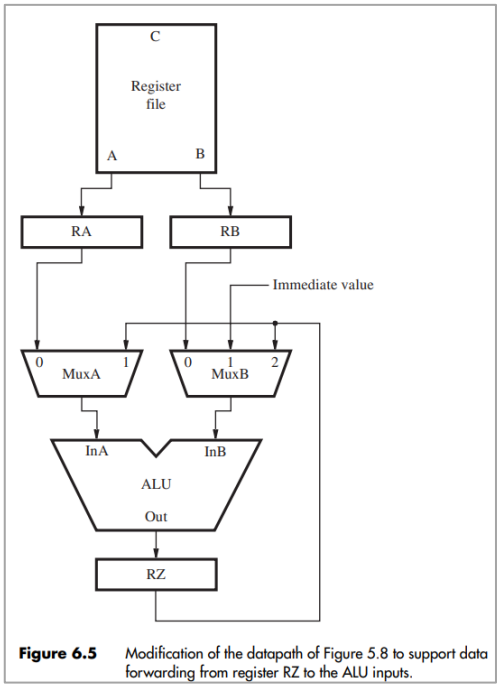


Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

5.4.2) Handling Data Dependencies in Software :

An alternative approach is to leave the task of detecting data dependencies and dealing with them to the compiler. When the compiler identifies a data dependency between two successive instructions Ij and Ij+1, it can insert three explicit NOP (No-operation) instructions between them. The NOPs introduce the necessary delay to enable instruction Ij+1 to read the new value from the register file after it is written. But forwarding is better than this as you can see.

5.4.3) Register Renaming :

Register renaming is done to eliminate WAR (Write after Read) and WAW (Write after Write) dependency between instructions which could have caused pipeline stalls.

WAR (Anti dependency) : If Read then write write on same variable then consider first pair only.

RAW (True data dependency)

WAW (Output dependency) : if I1, I3, I5 instruction having write on r3 then only two d exists. i.e. I1-I3, I3-I5

Example: I1: Read A to B

I2: Write C to A

Here, there is a WAR dependency and pipeline would need stalls. In order to avoid it register renaming is done and

Write C to A will be

Write C to A'

WAR dependency is actually called **anti-dependency** and there is no real dependency except the fact that both uses same memory location. Register renaming can avoid this. Similarly, WAW also.

3.5) Memory Delays : Delays arising from memory accesses are another cause of pipeline stall. For example, a Load instruction may require more than one clock cycle to obtain its operand from memory. This may occur because the requested instruction or data are not found in the cache, resulting in a **cache miss.**

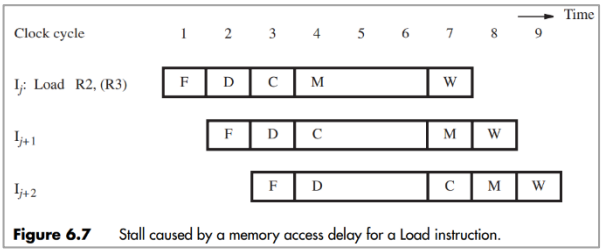


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

Consider the instructions:

Load R2, (R3)

Subtract R9, R2, #30

Assume that the data for the Load instruction is found in the cache, requiring only one cycle to access the operand. The destination register R2 for the Load instruction is a source register for the Subtract instruction. Operand forwarding cannot be done in the same manner as Figure 6.4, because the data read from memory (the cache, in this case) are not available until they are loaded into register RY at the beginning of cycle 5. Therefore, the Subtract instruction must be stalled for one cycle, as shown in Figure 6.8, to delay the ALU operation. The memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.

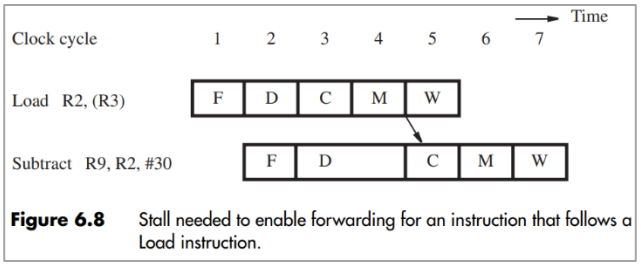


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

3.6) Branch Delays :

We now examine the effect of branch instructions and the techniques that can be used for mitigating their impact on pipelined execution.

5.6.1) Unconditional Branches : In pipelined execution, instructions Ij+1 and Ij+2 are fetched in cycles 2 and 3, respectively, before the branch instruction is decoded and its target address is known. They must be discarded. The resulting two-cycle delay constitutes a **branch penalty**.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some instructions in a program are executed many times, because of loops.)

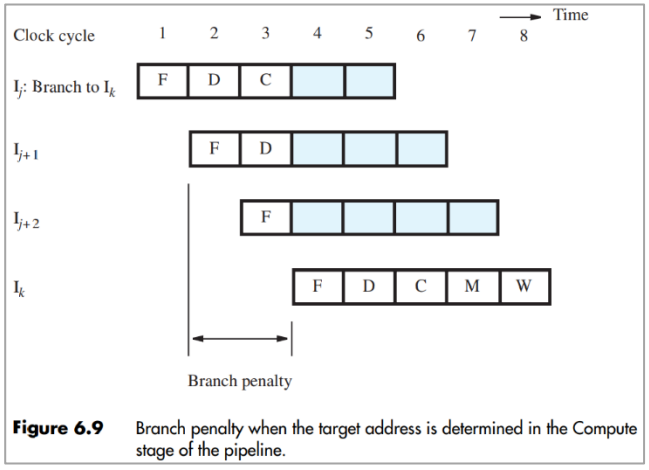


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

The hardware in Figure 5.10 must be modified to implement this change. The adder in the figure is needed to increment the PC in every cycle. A second adder is needed in the Decode stage to compute a branch target address for every instruction.

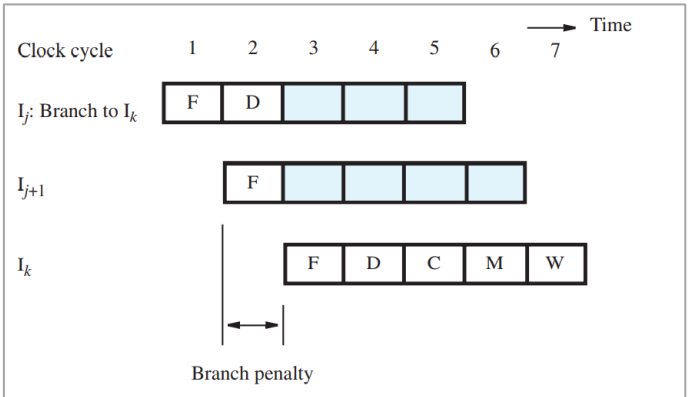


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

5.6.2) Conditional Branches : Consider a conditional branch instruction such as Branch_if_[R5]=[R6] LOOP The execution steps for this instruction are shown in Figure 5.16. The result of the comparison in the third step determines whether the branch is taken. For pipelining, the branch condition must be tested as early as possible to limit the branch penalty. We have just described how the target address for an unconditional branch instruction can be determined in the Decode stage. Similarly, the comparator that tests the branch condition can also be moved to the Decode stage, enabling the conditional branch decision to be made at the same time that the target address is determined.

5.6.3) The Branch Delay Slot : The location that follows a branch instruction is called the **branch delay slot**. Rather than conditionally discard the instruction in the delay slot, we can arrange to have the pipeline always execute this instruction, whether or not the branch is taken. The Add instruction is always fetched and executed, even if the branch is taken. Instruction Ij+1 is fetched only if the branch is not taken. Logically, execution proceeds as though the branch instruction were placed after the Add instruction. That is, branching takes place **one instruction later** than where the branch instruction appears in the instruction sequence. This technique is called **delayed branching**.

5.6.4) Branch Prediction : To reduce the branch penalty further, the processor needs to anticipate that an instruction being fetched is a branch instruction and *predict* its outcome to determine which instruction should be fetched in cycle 2. In this section, we first describe different methods for branch prediction. Then, we discuss how the prediction is made in cycle 1 while a branch instruction is being fetched.

1) Static Branch Prediction : The simplest form of branch prediction is to assume that the branch will not be taken and to fetch the next instruction in sequential address order. If the prediction is correct, the fetched instruction is allowed to complete and there is no penalty. This simple approach is a form of **static branch prediction**.

2) Dynamic Branch Prediction : The two states are:

LT - Branch is likely to be taken

LNT - Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is executed and the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the state machine is in state LT. Otherwise it is predicted as not taken.

Better prediction accuracy can be achieved by keeping more information about execution history. An algorithm that uses four states is shown in Figure 6.12b.

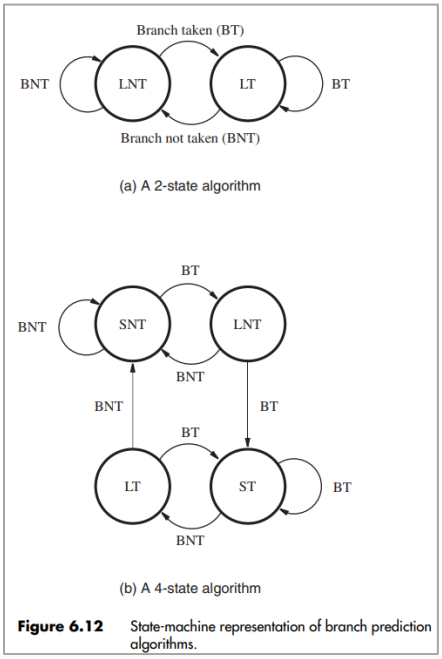


Figure 6.12 State-machine representation of branch prediction algorithms.

3.7) Resource Limitations : If two instructions need to access the same resource in the same clock cycle, one instruction must be stalled to allow the other instruction to use the resource. This can be prevented by providing additional hardware. If 25 percent of all instructions executed are Load or Store instructions, these stalls increase the execution time by 25 percent. Using separate caches for instructions and data allows the Fetch and Memory stages to proceed simultaneously without stalling.

3.8) Performance Evaluation :

For a non-pipelined processor, the execution time, T, of a program that has a dynamic instruction count of N is given by T = N × S/R where S(CPI) is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate in cycles per second. This is often referred to as the *basic performance equation*. A useful performance indicator is the instruction throughput, which is the number of instructions executed per second. For non-pipelined execution, the throughput, P_{np}, is given by P_{np} = R/S. The processor presented in Chapter 5 uses five cycles to execute all instructions. Thus, if there are no cache misses, S is equal to 5.

For the five-stage pipeline described in this chapter, each instruction is executed in five cycles, but a new instruction can ideally enter the pipeline every cycle. Thus, in the absence of stalls, S is equal to 1, and the ideal throughput with pipelining is P_p = R

A five-stage pipeline can potentially increase the throughput by a factor of five. In general, an n-stage pipeline has the potential

to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

• How much of this potential increase in instruction throughput can actually be realized in practice?

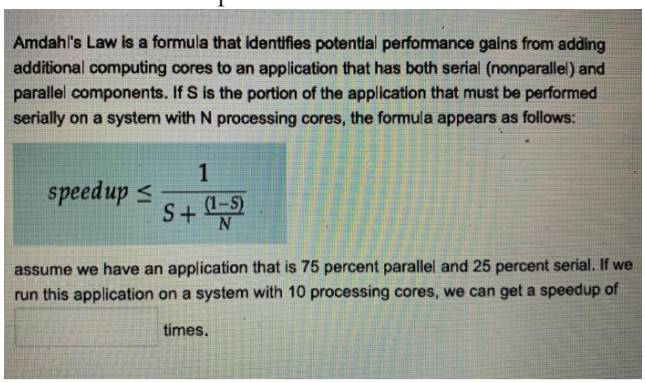
• What is a good value for n?

5.8.1) Number of Pipeline Stage : The fact that an n-stage pipeline may increase instruction throughput by a factor of n suggests that we should use a large number of stages. However, as the number of pipeline stages increases, there are more instructions being executed concurrently. Consequently, there are more potential dependencies between instructions that may lead to pipeline stalls. Furthermore, the branch penalty may be larger than one cycle if a longer pipeline moves the branch decision to a later stage. For these reasons, the gain in throughput from increasing the value of n begins to diminish, and the cost of a deeper pipeline may not be justified.

Read Amdahl's Law from 12.7 Performance Modeling…

Consider a program whose execution time on some computer is T_{orig}. Our objective is to assess the extent to which the execution time can be reduced when a performance enhancement, such as parallel processing, is introduced. Assume that a fraction f_{enh} of the execution time is affected by the enhancement. The remaining fraction, $f_{unenh} = 1 - f_{enh}$, is unchanged. Let p represent the factor by which the portion of time $f_{enh} \times T_{orig}$ is reduced due to the performance enhancement means number of processor. The new execution time is $T_{new} = T_{orig}\ (f_{unenh} + f_{enh}/p)$. The speedup is the ratio T_{orig}/T_{new} or $1/\ (f_{unenh} + f_{enh}/p)$ The above expression for speedup is known as **Amdahl's Law**.

Another Version : Amdahl's law is not restricted to parallel and serial operation it is applicable to many other cases where there is enhancement in portion of execution.



The important conclusion from this discussion is that the unenhanced portion of the original execution time can significantly limit the achievable speedup, even if the enhanced portion is improved by an arbitrarily large factor.

Example :

Consider a task that makes extensive use of floating-point operation with 60% of time consumed by floating point operations with a new hardware design. The floating-point module is speed up by factor of 4. The overall speedup is _____. (Upto 2 decimal places)

Answer :

Given, Speedup = 4

Frequency of floating point = 60%

Using the formula:

$$\text{Speedup} = \left[(1-F)+\frac{F}{S}\right]^{-1}$$

$$= \left[(1-0.6)+\frac{0.6}{4}\right]^{-1}$$

$$= [0.4+0.15]^{-1} = \frac{1}{0.55} = 1.81$$

Problems on Pipelining :

1) Consider the following program segment for a hypothetical CPU having three user registers R1, R2 and R3.

Instruction	Operation	Instruction size (in Words)
MOV $R_1, 5000$	$R1 \leftarrow$ Memory$[5000]$	2
MOV $R_2(R_1)$	$R2 \leftarrow$ Memory$[(R_1)]$	1
ADD R_2, R_3	$R_2 \leftarrow R_2 + R_3$	1
MOV $6000, R_2$	Memory$[6000] \leftarrow R_2$	2
Halt	Machine Halts	1

Let the clock cycles required for various operations be as follows:

Register to/from memory transfer	3 clock cycles
ADD with both operands in register	1 clock cycles
Instruction fetch and decode	2 clock cycles

The total number of clock cycles required to execute the program is

Answer :

Instruction	Size	Fetch and Decode + Execute
MOV	2	$2 \times 2 + 3 = 7$
MOV	1	$2 \times 1 + 3 = 5$
ADD	1	$2 \times 1 + 1 = 3$
MOV	2	$2 \times 2 + 3 = 7$
HALT	1	$2 \times 1 + 0 = 2$
	Total	24 Cycles

2) Following table indicates the latencies of operations between the instruction producing the result and instruction using the result

Instruction producing the result	**Instruction using the result**	**Latency**
ALU Operation	ALU Operation	2
ALU Operation	Store	2
Load	ALU Operation	1
Load	Store	0

Consider the following code segment :

Load R1, Loc 1; Load R1 from memory location Loc1
Load R2, Loc 2; Load R2 from memory location Loc 2
Add R1, R2, R1; Add R1 and R2 and save result in R1
Dec R2; Decrement R2
Dec R1; Decrement R1
Mpy R1, R2, R3; Multiply R1 and R2 and save result in R3
Store R3, Loc 3; Store R3 in memory location Loc 3

Answer : First meaning of table : see third instruction it says that R2 and R1 is being produced by first and second instruction which is load instruction and the result of this load is begin used by our third instruction which is ALU operation. So from table there is delay of 1. And that is why T3 is empty in Gantt chart. first step is to note that latencies is between two instruction not per instruction. Now there are latencies between two instruction whenever there exist some dependency between two instruction. For example, value used by 3rd instruction is being produced by instruction two and one. You can make Gantt chart based on this as given below :

Clock cycle	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
	I1	I2		I3	I4		I5			I6			I7

Remember that if one instruction depends upon two instruction then we have consider all latency of instruction by subtracting addition middle latencies. 13 is answer.

3) Consider 6 stage pipeline with a cycle of 3 ns used to execute the program which contains combination of 3 categories of instruction 40, 50 and 60 respectively. All the 3 categories of instructions are taking 1 cycle on all the stages but Category-1 instruction takes 6 cycles, Category-2 instruction takes 4 cycles and Category-3 instruction takes 2 cycles on 5th stage of a pipeline. How much time is required to complete the program.

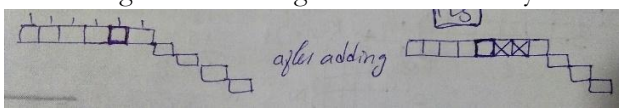
Answer : formula is important will save your lot of time. K = maximum phase time

Number of cycles required to execute the instruction

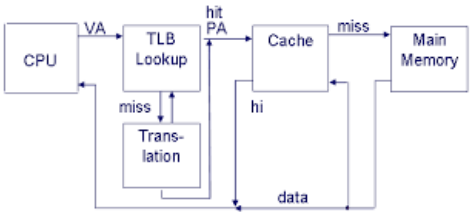
$$= (K + n - 1) + 40 \times (6 - 1) + 50 \times (4 - 1) + 60 \times (2 - 1)$$
$$= (6 + 150 - 1) + 40 \times 5 + 50 \times 3 + 60 \times 1 = 565$$

Total time to execute the instruction = 565 × 3 ns = 1695 ns

use this logic while solving. First we take on cycle and after calculating first image time we add remaining cross boxes.



NOTE :



1)
2) If CPU encounters any interrupt then it will complete the current address instruction execution and the store the address of instruction after current instruction on stack and then process interrupt.
3) Execution time = CPI * No. of instruction * Time taken per stage (You can verify)
4) On a perfect pipeline (i.e., one which has no stalls) CPI=1 as during it an instruction takes just one cycle time to get completed.
5) Speed Up =Old Execution Time of an Instruction/New Execution Time of an Instruction.

Latency	Bandwidth
The amount of time needed for an operation to complete.	The rate at which operations are performed.
A memory load that misses the cache has a latency of 200 cycles	Memory can provide data to the processor at 25 GB/sec.
A packet takes 20 ms to be sent from my computer to Google	A communication link can send 10 million messages per second
Asking a question on Piazza gets response in 10 minutes	The TAs answer 50 questions per day on Piazza

6)

So, if Operation are fast then latency will be low.

7) OPcode in processor defines number of operations like ADD, SUB, MUL… It does not mean number of instruction but it represents number of instructions as one instruction contains one operation.
8) If question includes "All register reads take place in the second phase of a clock cycle and all register writes occur in the first phase." Means read happens at the second phase of clock cycle of second stage (i.e. decode stage) and write occurs at the first phase of clock cycle of last stage (i.e. write stage).
9) There is difference between dependency and hazards. For hazards you have to draw table or diagram. For dependency you can refer instruction sequence.
10) In without operand forwarding **or *bypassing*** case you can still pipeline till instruction decode stage but you can only execute after write. Like these two examples : **Diagram 1 is same as all register being written in the first half of WB and read in the last half of ID.**

	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
MUL		IF	ID			EX	EX	MEM	WB

Instruction	Clock cycle number															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LW R1, 0 (R4)	IF	ID	Ex	M	W											
LW R2,400(R4)		IF	ID	Ex	M	W										
ADDI R3,R1,R2			IF	ID	*	*	Ex	M	W							
SW R3, 0 (R4)				IF	*	*	ID	Ex	*	M	W					
SUB R4, R4, #4							IF	ID	*	Ex	M	W				
BNEZ R4, L1								IF	*	ID	*	*	Ex	M	W	

4. INPUT/OUTPUT ORGANIZATION (Buses)

An interconnection network is used to transfer data among the processor, memory, and I/O devices. We describe below a commonly used interconnection network called a bus.

4.1) Bus Structure :

The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown in Figure 7.2 for an input device. Each I/O device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines. When I/O devices and the memory share the same address space, the arrangement is called **memory-mapped I/O**.

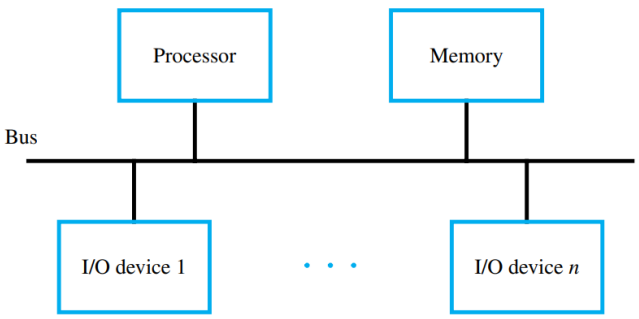


Figure 7.1 A single-bus structure.

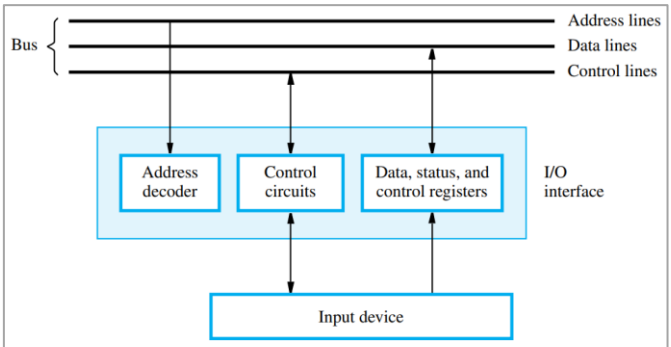


Figure 7.2 I/O interface for an input device.

4.2) Bus Operation :

A bus requires a set of rules, often called a **bus protocol**, that govern how the bus is used by various devices. The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers, and so on. These rules are implemented by control signals that indicate what and when actions are to be taken. One control line, usually labelled R/¬W, specifies whether a Read or a Write operation is to be performed. As the label suggests, it specifies Read when set to 1 and Write when set to 0.

In any data transfer operation, one device plays the role of a **master**. This is the device that initiates data transfers by issuing Read or Write commands on the bus. Normally, the processor acts as the master, but other devices may also become masters as we will see in Section 7.3. The device addressed by the master is referred to as a **slave**.

6.2.1) Synchronous Bus :

On a synchronous bus, all devices derive timing information from a control line called the **bus clock**, shown at the top of Figure 7.3. The signal on this line has two phases: a high level followed by a low level. The two phases constitute a **clock cycle**.

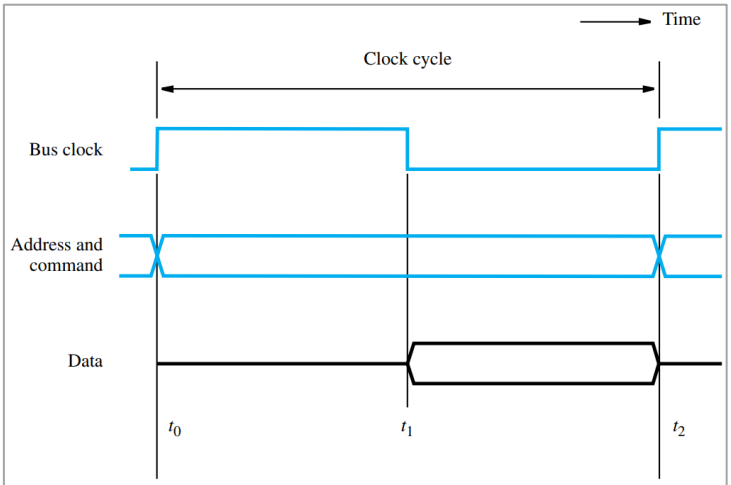


Figure 7.3 Timing of an input transfer on a synchronous bus.

Let us consider the sequence of signal events during an input (Read) operation. At time t0, the master places the device address on the address lines and sends a command on the control lines indicating a Read operation. The command may also specify the length of the operand to be read. Information travels over the bus at a speed determined by its physical and electrical characteristics. The clock pulse width, t1 − t0, must be longer than the maximum propagation delay over the bus. Also, it must be long enough to allow all devices to decode the address and control signals, so that the addressed device (the slave) can respond at time t1 by placing the requested input data on the data lines. At the end of the clock cycle, at time t2, the master loads the data on the data lines into one of its registers. To be loaded correctly into a register, data must be available for a period greater than the **setup time of the register** (see Appendix A). Hence, the period t2 − t1 must be greater than the maximum propagation time on the bus plus the setup time of the master's register.

Figure 7.4 gives a more realistic picture of what actually happens.

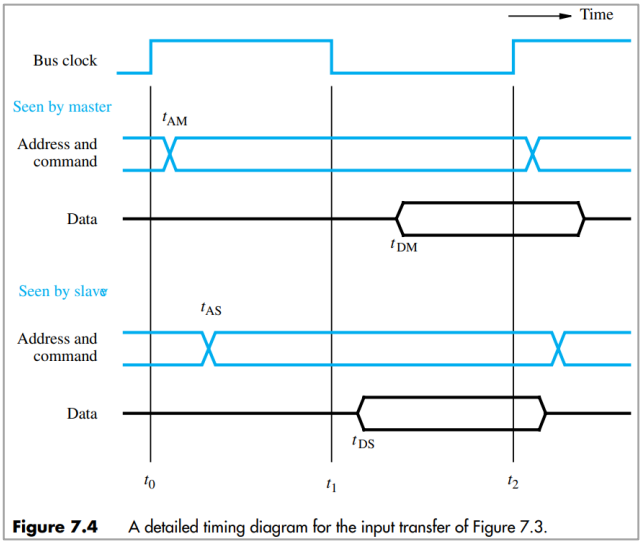


Figure 7.4 A detailed timing diagram for the input transfer of Figure 7.3.

The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t0). However, these signals do not actually appear on the bus until tAM, largely due to the delay in the electronic circuit output from the master to the bus lines. A short while later, at tAS, the signals reach the slave. The slave decodes the address, and at t1 sends the requested data. Here again, the data signals do not appear on the bus until tDS. They travel toward the master and arrive at tDM. At t2, the master loads the data into *its register.* Hence the period t2 − tDM must be greater than the setup time of that register. The data must continue to be valid after t2 for a period equal to the hold time requirement of the register (see Appendix A for hold time).

The number of clock cycles involved can vary from one device to another. An example of this approach is shown in Figure 7.5.

Multiple-Cycle Data Transfer :

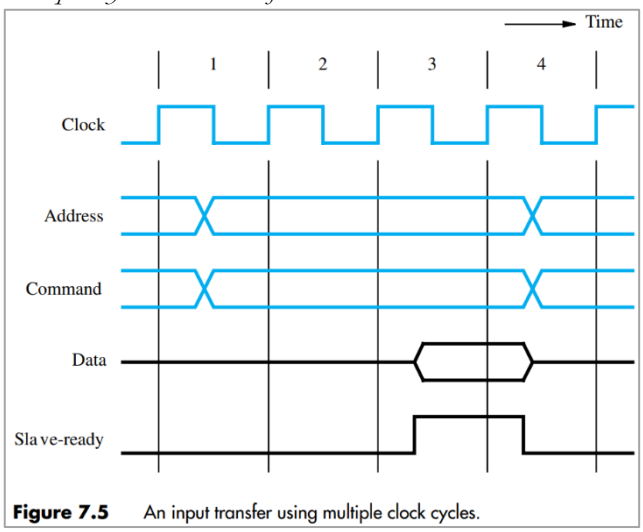


Figure 7.5 An input transfer using multiple clock cycles.

During clock cycle 1, the master sends address and command information on the bus, requesting a Read operation. The slave receives this information and decodes it. It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2. We have assumed that due to the delay involved in getting the data, the slave cannot respond immediately. The data become ready and are placed on the bus during clock cycle 3. The slave asserts a control signal called **Slave-ready** at the same time. The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle. The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3. The Slave-ready signal is an acknowledgment from the slave to the master, confirming that the requested data have been placed on the bus.

6.2.2) Asynchronous Bus : An alternative scheme for controlling data transfers on a bus is based on the use of a handshake protocol between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave.

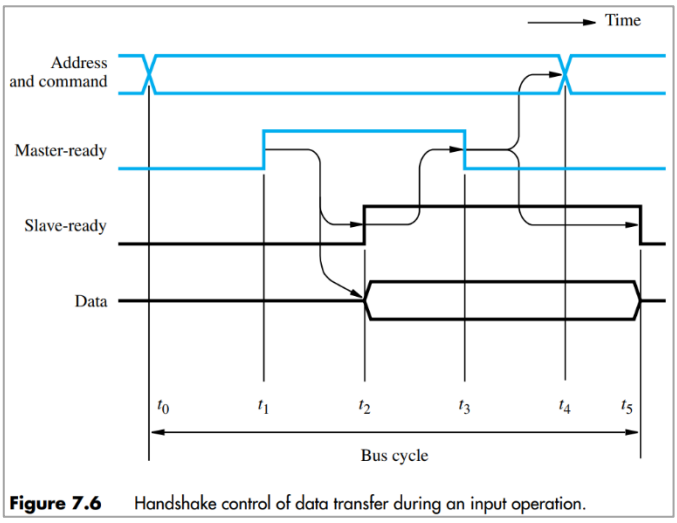


Figure 7.6 Handshake control of data transfer during an input operation.

t0—The master places the address and command information on the bus, and all devices on the bus decode this information.

t1—The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. The delay t1 − t0 is intended to allow for any skew that may occur on the bus. Skew occurs when two signals transmitted simultaneously from one source arrive at the destination at different times. The delay t1 − t0 should be longer than the maximum possible bus skew. (Note that bus skew is a part of the maximum propagation delay in the synchronous case.)

t2—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1.

t3—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. The master must allow for bus skew. It must also allow for the setup time needed by its register.

t4—The master removes the address and command information from the bus. The delay between t3 and t4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

t5—When the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer. The handshake signals in Figures 7.6 and 7.7 are said to be **fully**

interlocked, because a change in one signal is always in response to a change in the other. Hence, this scheme is known as a **full handshake**.

It is essential to ensure that only one device can place data on the bus at any given time. A logic gate that places data on the bus is called a **bus driver**. All devices connected to the bus, except the one that is currently sending data, must have their bus drivers turned off. A special type of logic gate, known as **a tri-state gate**, is used for this purpose. has a control input that is used to turn the gate on or off. When turned on, or enabled, it drives the bus with 1 or 0, corresponding to the value of its input signal. When turned off, or disabled, it is effectively disconnected from the bus.

4.3) Arbitration :

Two devices may need to access a given slave at the same time. In such cases, it is necessary to decide which device will access the slave first. The decision is usually made in an arbitration process performed by an **arbiter circuit**. The arbitration process starts by each device sending a *request* to use the shared resource. The arbiter associate's priorities with individual requests. If it receives two requests at the same time, it *grants* the use of the slave to the device having the higher priority first.
In Section 6.2, the discussion involved only one bus master—the processor.

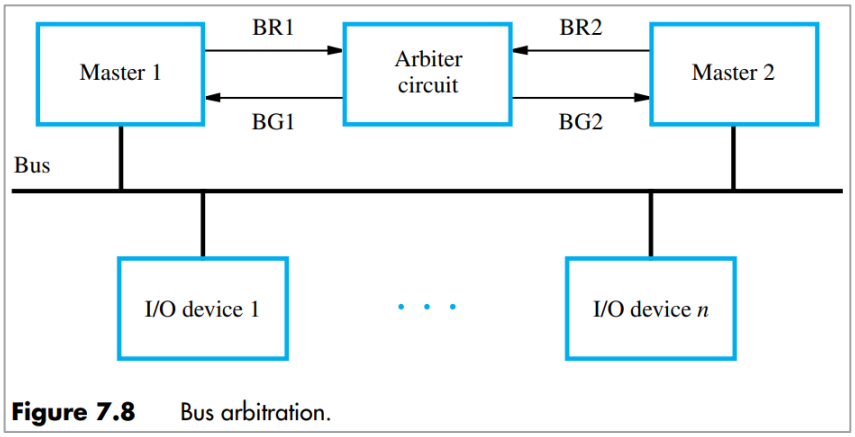


Figure 7.8 Bus arbitration.

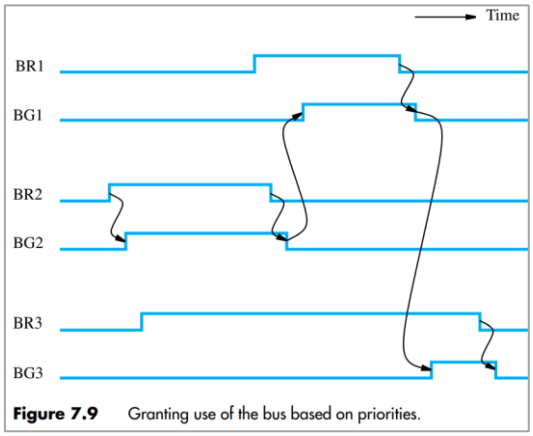


Figure 7.9 Granting use of the bus based on priorities.

For some devices, a delay in gaining access to the bus may lead to an error. Such devices must be given high priority. If there is no particular urgency among requests, the arbiter may grant the bus using a simple round-robin scheme.

Assume that master 1 has the highest priority, followed by the others in increasing numerical order. Master 2 sends a request to use the bus first. Since there are no other requests, the arbiter grants the bus to this master by asserting BG2. When master 2 completes its data transfer operation, it releases the bus by deactivating BR2. By that time, both masters 1 and 3 have activated their request lines. Since device 1 has a higher priority, the arbiter activates BG1 after it deactivates BG2, thus granting the bus to master 1. Later, when master 1 releases the bus by deactivating BR1, the arbiter deactivates BG1 and activates BG3 to grant the bus to master 3. Note that the bus is granted to master 1 before master 3 even though master 3 activated its request line before master 1.

4.4) Interface Circuits :

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a **port**, and it can be either a **parallel** or a **serial port**. A parallel port transfers multiple bit of data simultaneously to or from the device. A serial port sends and receives data one bit at a time.
Before we present specific circuit examples, let us recall the functions of an I/O interface. According to the discussion in Section 3.1, an I/O interface does the following:

1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Includes a control register that holds the information governing the behavior of the interface
4. Contains address-decoding circuitry to determine when it is being addressed by the processor
5. Generates the required timing signals
6. Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port

6.4.1) Parallel Interface :

Input Interface : Figure 7.10 shows a circuit that can be used to connect a keyboard to a processor. The registers in this circuit correspond to those given in Figure 3.3. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such mechanical pushbutton switches is that the contacts **bounce** when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position.

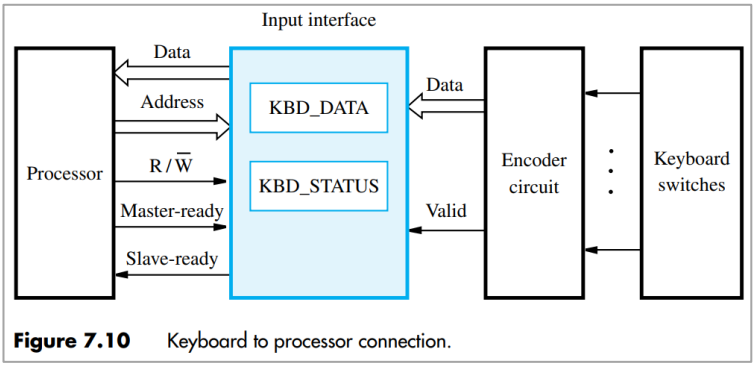
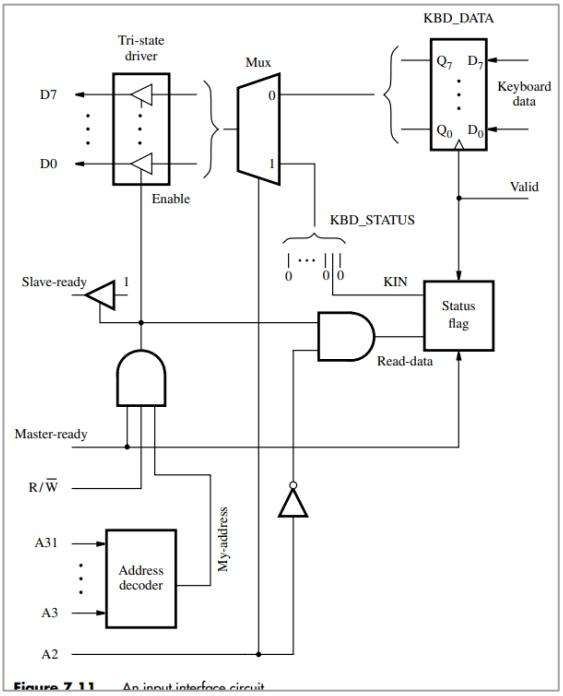


Figure 7.10 Keyboard to processor connection.

The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided. When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1.

When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status flag KIN to be set to 1. The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The bus has one other control line, R/W, which indicates a Read operation when equal to 1.



When the processor requests a Read operation, it places the address of the appropriate register on the address lines of the bus. The address decoder in the interface circuit examines bits A31−3, and asserts its output, My-address, when one of the two registers KBD_DATA or KBD_STATUS is being addressed.

The interface circuit turns the tri-state gates on only when the three signals Master-ready, My_address, and R/W are all equal to 1, indicating a Read operation.

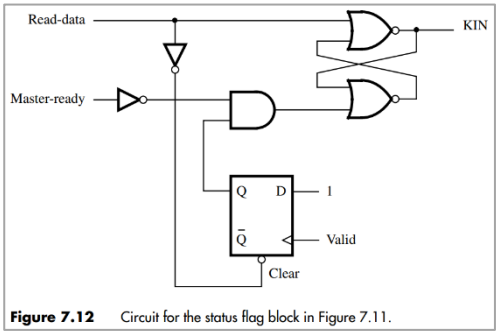


Figure 7.12 Circuit for the status flag block in Figure 7.11.

Output Interface : When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1. When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA. When it is ready to receive another character, it asserts Ready again, and the cycle repeats.

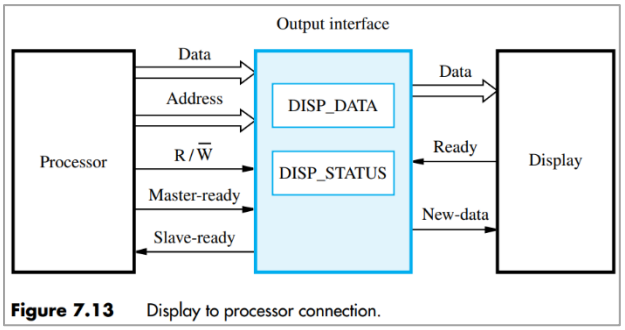


Figure 7.13 Display to processor connection.

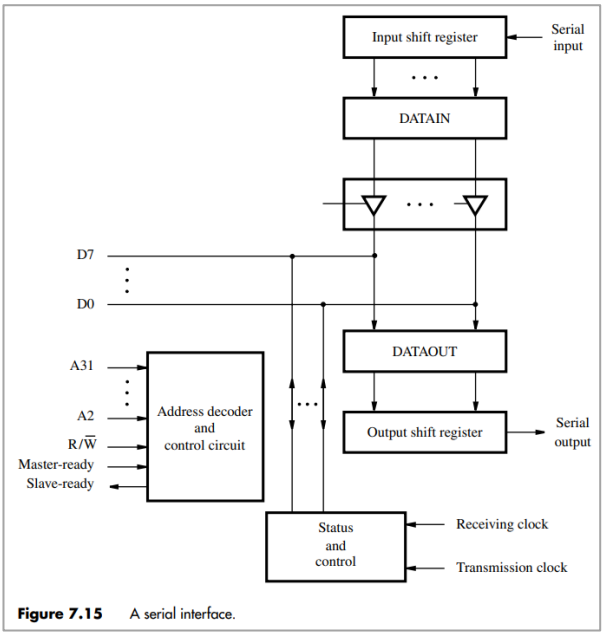


Figure 7.15 A serial interface.

6.4.2) Serial Interface :

A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor side. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are transferred to the output shift register, from which the bits are shifted out and sent to the I/O device.

During serial transmission, the receiver needs to know when to shift each bit into its input shift register. Since there is no separate line to carry a clock signal from the transmitter to the receiver, the timing information needed must be embedded into the transmitted data using an encoding scheme. There are two basic approaches.

Asynchronous Transmission :

This approach uses a technique called start-stop transmission. Data are organized in small groups of 6 to 8 bits, with a well-defined beginning and end. The line connecting the transmitter and the receiver is in the 1 state when idle. A character is transmitted as a 0 bit, referred to as the Start bit, followed by 8 data bits and 1 or 2 Stop bits. The Stop bits have a logic value of 1. The 1-to-0 transition at the beginning of the Start bit alerts the receiver that data transmission is about to begin. Using its own clock, the receiver determines the position of the next 8 bits, which it loads into its input register. The Stop bits following the transmitted character, which are equal to 1, ensure that the Start bit of the next character will be recognized. When transmission stops, the line remains in the 1 state until another character is transmitted.

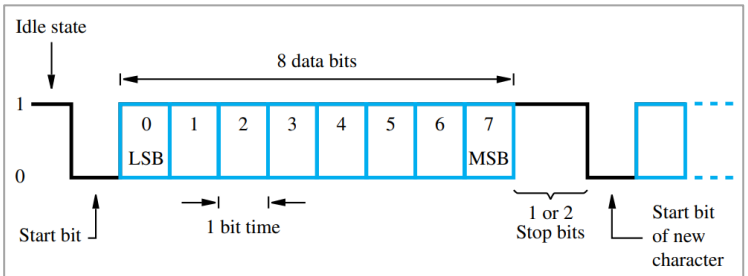


Figure 7.16 Asynchronous serial character transmission.

Synchronous Transmission :

In synchronous transmission, the receiver generates a clock that is synchronized to that of the transmitter by observing successive 1-to-0 and 0-to-1 transitions in the received signal. It adjusts the position of the active edge of the clock to be in the center of the bit position. Synchronous transmission enables very high data transfer rates.

Point of Comparison	Synchronous Transmission	Asynchronous Transmission
Definition	Transmits data in the form of chunks or frames	Transmits 1 byte or character at a time
Speed of Transmission	Quick	Slow
Cost	Expensive	Cost-effective
Time Interval	Constant	Random
Gaps between the data?	Does not exist	Exist
Examples	Chat Rooms, Telephonic Conversations, Video Conferencing	Email, Forums, Letters

An ISR (inter service routing) is invoked in synchronous transmission and schedule handler is invoked in asynchronous transmission.

5. THE MEMORY SYSTEM

5.1) Basic Concept :

The memory is usually designed to store and retrieve data in word-length quantities. Consider, for example, a byte-addressable computer whose instructions generate 32-bit addresses. When a 32-bit address is sent from the processor to the memory unit, the high order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved.

MFC signal : The processor's internal control signal that indicates that the requested memory operation has been completed. When asserted, the processor proceeds to the next step in its execution sequence.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation. This is referred to as the **memory access time**. Another important measure is the **memory cycle time**, which is the minimum time delay required between the initiation of two successive *memory operations*, for example, the time between two successive Read operations. The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit.

Cache and Virtual Memory : Cache memory is a small, fast memory inserted between the larger, slower main memory and the processor. It holds the currently active portions of a program and their data.

Virtual memory is another important concept related to memory organization. With this technique, only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device. Sections of the program are transferred back and forth between the main memory and the secondary storage device in a manner that is transparent to the application program. As a result, the application program sees a memory that is much larger than the computer's physical main memory.

5.2) Semiconductor RAM Memories :

we discuss the main characteristics of Semiconductor RAM memories. We start by introducing the way that memory cells are organized inside a chip.

7.2.1) Internal Organization of Memory Chips :

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. A possible organization is illustrated in Figure 8.2. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the **word line (W_0)**, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two-bit lines, and the Sense/Write circuits are connected to the data input/output lines of the chip. Two control lines, R/W and CS, are provided. The R/W (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

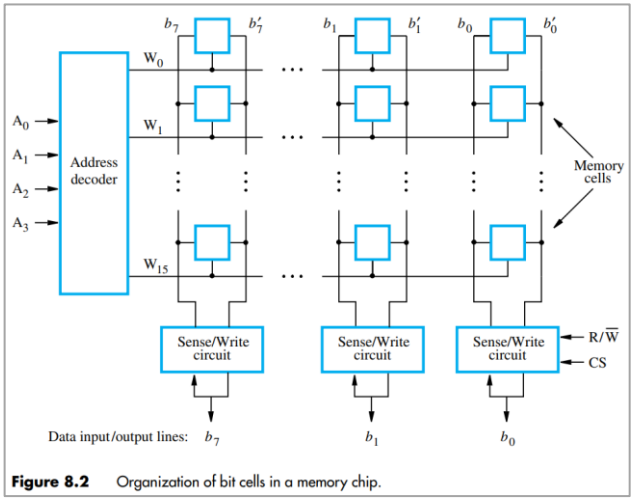


Figure 8.2 Organization of bit cells in a memory chip.

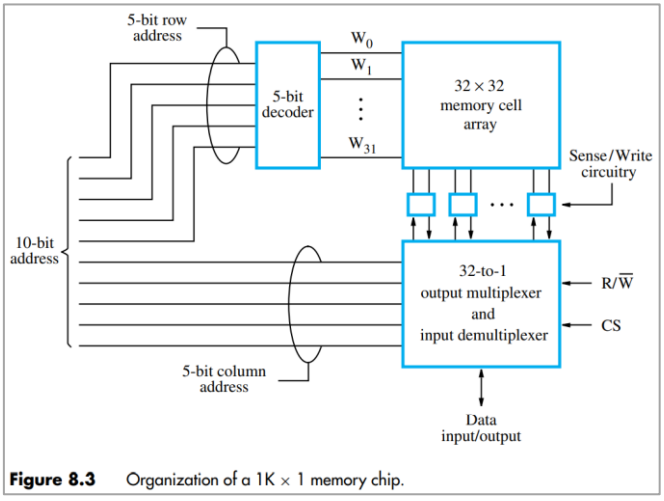


Figure 8.3 Organization of a 1K × 1 memory chip.

The memory circuit in Figure 8.2 stores 128 bits and requires 14 external connections for address, data, and control lines. It also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K × 1 format shown in figure 8.3.

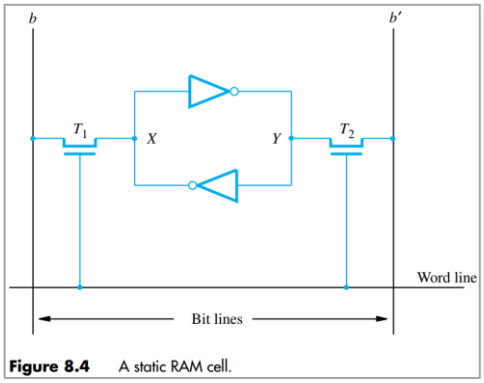


Figure 8.4 A static RAM cell.

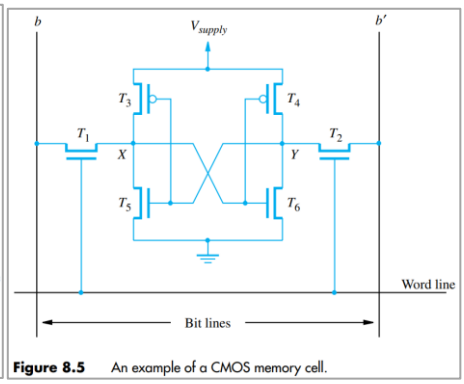


Figure 8.5 An example of a CMOS memory cell.

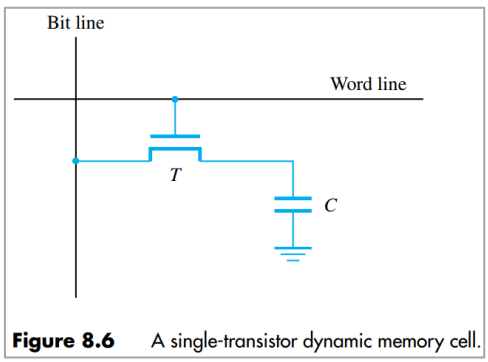


Figure 8.6 A single-transistor dynamic memory cell.

7.2.2) Static Memories : Memories that consist of circuits capable of retaining their state as long as power is applied are known as **static memories**. Figure 8.4 illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, if the logic value at point X is 1 and at point Y is 0, this state is maintained as long as the signal on the word line is at ground level. Assume that this state represents the value 1.

CMOS Cell : In state 1(X =1, Y=0), the voltage at point X is maintained high by having transistors T3 and T6 on, while T4 and T5 are off. If T1 and T2 are turned on, bit lines b and b' will have high and low signals, respectively. SRAMs are said to be **volatile memories** because their contents are lost when power is interrupted. SRAMs are used in applications where speed is of critical concern.

7.2.3) Dynamic RAMs : Static RAMs are fast, but their cells require several transistors. Less expensive and higher density RAMs can be implemented with simpler cells. But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations. Memories that use such cells are called **dynamic RAMs** (DRAMs).

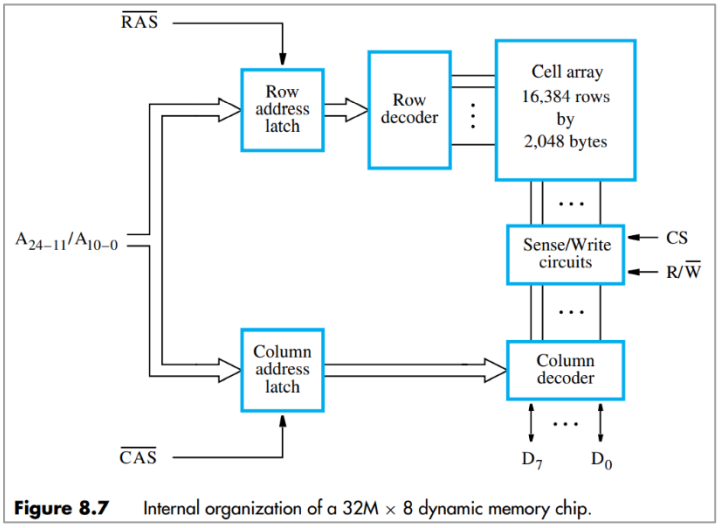


Figure 8.7 Internal organization of a 32M × 8 dynamic memory chip.

To store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor. After the transistor is turned off, the charge remains stored in the capacitor, but not for long. The capacitor begins to discharge. This is because the transistor continues to conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge in the capacitor drops below some threshold value.

During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on an input control line called the **Row Address Strobe** (RAS).

Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of a second control line called the **Column Address Strobe** (CAS).

During a Read operation, the output data are transferred to the processor after a delay equivalent to the memory's access time. Such memories are referred to as **asynchronous DRAMs**. DRAMs also supports **fast page mode**. The block transfer capability is referred to as the *fast page mode feature.*

Synchronous DRAMs : The distinguishing feature of an SDRAM is the use of a clock signal, the availability of which makes it possible to incorporate control circuitry on the chip that provides many useful features.

Latency and Bandwidth : **Memory latency** is the amount of time it takes to transfer the first word of a block. A useful performance measure is the number of bits or bytes that can be transferred in one second. This measure is often referred to as the **memory bandwidth**. It depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel.

7.2.4) Structure of Larger Memories :

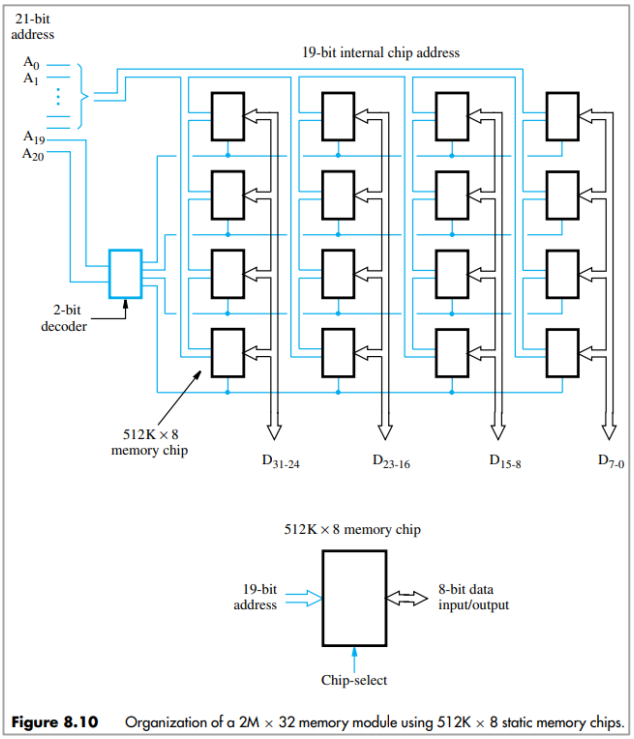


Figure 8.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

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

5.3) Direct Memory Access :

Data are transferred from an I/O device to the memory by first reading them from the I/O device using an instruction such as Load R2, DATAIN which loads the data into a processor register. Then, the data read are stored into a memory location. The reverse process takes place for transferring data from the memory to an I/O device. An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready, either by polling its status register or by waiting for an interrupt request. But consider if there is large amount of data to be transfer between memory and I/O devices.

An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks. A special control unit is provided to manage the transfer, without continuous intervention by the processor. This approach is called **direct memory access**, or **DMA**. The unit that controls DMA transfers is referred to as a **DMA controller**. DMA Controller is initialized by the CPU for the number of bytes to be transferred.

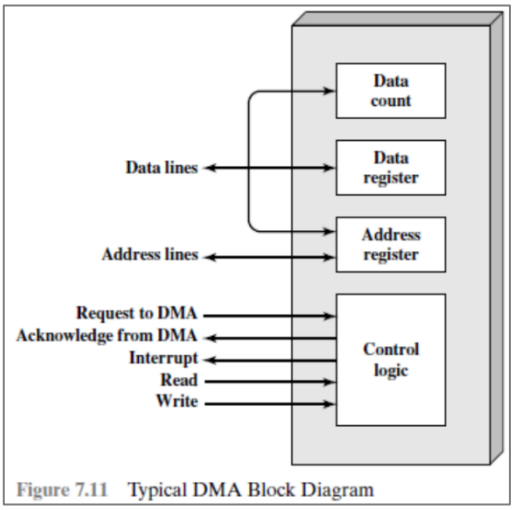


Figure 7.11 Typical DMA Block Diagram

CPU = Processor

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module
- The address of the I/O device involved, communicated on the data lines
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register.
- The number of words to be read or written, again communicated via the data lines and stored in the data count register.

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer. The DMA technique does not make use of interrupt mechanism.

If the DMA module is to transfer a block of data from memory to disk, it will do the following:

1. The peripheral device (such as the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
2. The DMA will put a high on its HRQ (hold request), signaling the CPU through its HOLD pin that it needs to use the buses.
3. The CPU will finish the present bus cycle (not necessarily the present instruction) and respond to the DMA request by putting high on its HDLA (hold acknowledge), thus telling the DMA that it can go ahead and use the buses to perform its task. HOLD must remain active high as long as DMA is performing its task.
4. DMA will activate DACK (DMA acknowledge), which tells the peripheral device that it will start to transfer the data.
5. DMA starts to transfer the data from memory to peripheral by putting the address of the first byte of the block on the address bus and activating MEMR.
6. After the DMA has finished its job it will deactivate HRQ, signaling the CPU that it can regain control over its buses.

Example :

Hard disk with transfer rate of 1 KBps is constantly transferring data to memory using DMA. The size of data transfer is 16 bytes. The processor runs at 400 kHz clock frequency. The DMA controller requires 10 cycles for initialization of operation and transfer takes 2 cycles to transfer one byte of data from device to the memory. Let M and N be the maximum percentage of time that the CPU is blocked in cycle stealing mode and Burst mode. Find the value of M-N ?

Answer :

for cycle stealing mode,

Transfer time=1KBps

For X,

$1\text{kBps} = 1 \text{ sec}, 1 \text{ Bps} = \frac{1}{10^3}$

For 16 bytes, $X = \frac{16}{10^3} = 16$ msec

For DMA transfer,

$f = 400 \text{ kHz}, t = \frac{1}{f} = \frac{1}{400 \times 10^3}$

for cycle stealing mode,

Transfer time(Y)= [initialization + transfer time] x clock frequency

$= [10 + 2] \times 16 \times \frac{1}{400 \times 10^3}$

$= \frac{192}{400 \times 10^3} = 0.48$ msec

$\%\text{ CPU blocked(M)} = \frac{Y}{X} \times 100$

$= \frac{0.48}{16} \times 100 = 3\ \%$

for burst mode,

Transfer time=1KBps

For X,

$1\text{kBps} = 1 \text{ sec}, 1 \text{ Bps} = \frac{1}{10^3}$

For 16 bytes, $X = \frac{16}{10^3} = 16$ msec

For DMA transfer,

$f = 400 \text{ kHz}, t = \frac{1}{f} = \frac{1}{400 \times 10^3}$

for burst mode,

Transfer time(Y)= {[initialization] + [transfer time]16} x clock frequency

$= [10 + 2 \times 16] \times \frac{1}{400 \times 10^3}$

$= \frac{42}{400 \times 10^3} = 0.105$ msec

$\%\text{ CPU blocked(N)} = \frac{Y}{X+Y} \times 100$

$= \frac{0.105}{0.105+16} \times 100 = 0.65\ \%$

Cycle Stealing mode:

In cycle stealing mode we always follow pipelining concept that when one byte is getting transferred then Device is parallel preparing the next byte. "The fraction of CPU time to the data transfer time" if asked then cycle stealing mode is used.

Where, X µsec =data transfer time or preparation time (words/block)

Y µsec =memory cycle time or cycle time or transfer time (words/block)

% CPU idle (Blocked) ={ Y/ X }* 100

% CPU busy = {X / Y } * 100

Burst mode:
X µsec =data transfer time or preparation time (words/block)
Y µsec =memory cycle time or cycle time or transfer time (words/block)
% CPU idle (Blocked) = {Y/ X+Y }x 100
% CPU busy = {X/ X+Y} x 100

5.4) Cache Memories :

The cache is a small and very fast memory, interposed between the processor and the main memory. The effectiveness of this approach is based on a property of computer programs called **locality of reference**. This behavior manifests itself in two ways: **temporal** and **spatial**. The first means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions close to a recently executed instruction are also likely to be executed soon.

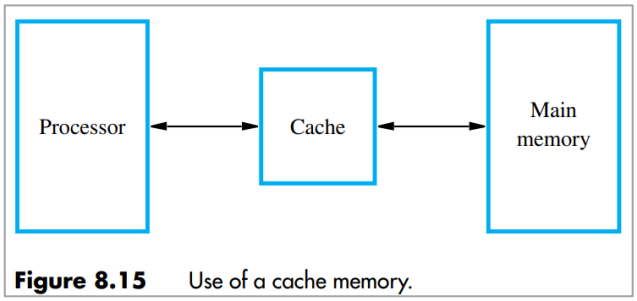


Figure 8.15 Use of a cache memory.

Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The term *cache block* refers to a set of contiguous address locations of some size. Another term that is often used to refer to a **cache block** is a **cache line**.

The correspondence between the main memory blocks and those in the cache is specified by a **mapping function**. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the **cache's replacement algorithm**.

7.4.1) The Basics of Caches :

Direct-mapped cache : A cache structure in which each memory location is mapped to exactly one location in the cache. For example, almost all direct-mapped caches use this mapping to find a block: (Block address) modulo (Number of blocks in the cache) If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order log2 (cache size in blocks) bits of the address. Thus, an 8-block cache uses the three lowest bits ($8 = 2^3$) of the block address. For example, Figure 5.8 shows how the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words. In a direct mapped cache with many words per cache line, the offset is given by the rightmost bits and the cache line's index is given by some of the middle bits.

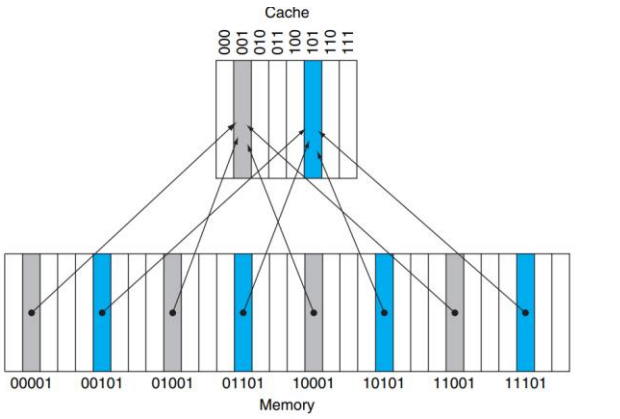


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

how do we know whether a requested word is in the cache or not? We answer this question by adding a set of tags to the cache. The **tags** contain the address information required to identify whether a word in the cache corresponds to the requested word.

When a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address.

7.4.2) Accessing a Cache :

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.6b)	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss (5.6c)	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss (5.6d)	(10000_{two} mod 8) = 000_{two}
3	00011_{two}	miss (5.6e)	(00011_{two} mod 8) = 011_{two}
16	10000_{two}	hit	(10000_{two} mod 8) = 000_{two}
18	10010_{two}	miss (5.6f)	(10010_{two} mod 8) = 010_{two}
16	10000_{two}	hit	(10000_{two} mod 8) = 000_{two}

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

■ A tag field, which is used to compare with the value of the tag field of the cache
■ A cache index, which is used to select the block.
■ 32-bit addresses
■ A direct-mapped cache
■ The cache size is 2^n blocks, so n bits are used for the index
■ The block size is 2^m words (2^m+2 bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address the size of the tag field is 32 - (n - m - 2).
The total number of bits in a direct-mapped cache is 2^n X (block size + tag size + valid field size)
Since the block size is 2^m words (2^m+5 bits), and we need 1 bit for the valid fi eld, the number of bits in such a cache is 2^n X (2^m X 32 + (32 - n - m - 2) + 1) = 2^n X (2^m X 32 + 31 - n - m).

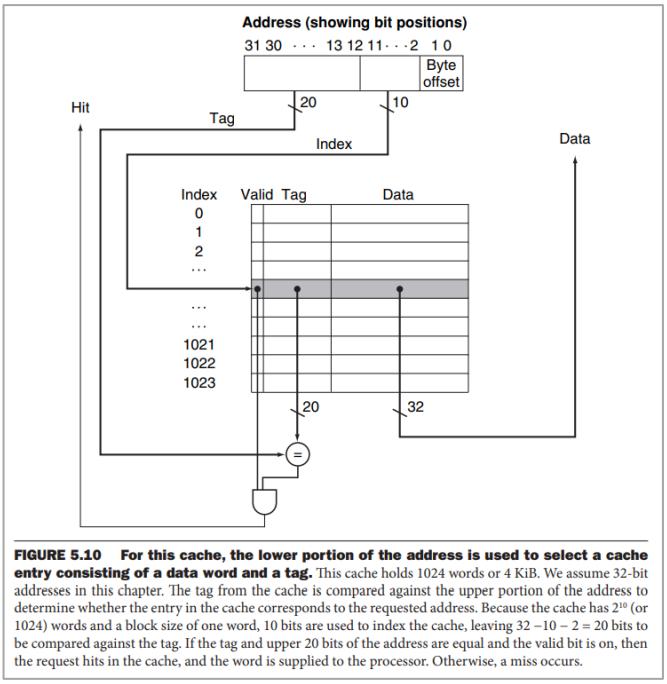


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving 32 –10 – 2 = 20 bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Bits in a Cache

How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address?

We know that 16 KiB is 4096 (2^{12}) words. With a block size of 4 words (2^2), there are 1024 (2^{10}) blocks. Each block has 4 × 32 or 128 bits of data plus a tag, which is 32 − 10 − 2 − 2 bits, plus a valid bit. Thus, the total cache size is

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kibibits}$$

or 18.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.

Mapping an Address to a Multiword Cache Block

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

We saw the formula on page 384. The block is given by

(Block address) modulo (Number of blocks in the cache)

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{6} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

This is address for cache. Address in main memory is **Tag + Index**.

Handling Cache Misses : Cache miss : A request for data from the cache that cannot be filled because the data is not present in the cache.
We can now define the steps to be taken on an instruction cache miss:
1. Send the original PC value (current PC – 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.
The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes : Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be inconsistent. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called **write-through**. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably.

One solution to this problem is to use a **write buffer**. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution.
The alternative to a write-through scheme is a scheme called **write-back**. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Here updates are made to main memory based on dirty bit. When data is loaded to the cache it is set to 0, If any byte in the block is modified, set it to 1. When the block is removed from the cache, check the dirty bit if 1 write the block to main memory.

7.4.3) Measuring and Improving Cache Performance :

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system.

CPU time = (CPU execution clock cycles + Memory-stall clock cycles) * Clock cycle time

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

$$\begin{aligned} \text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.05 \times 20 \\ &= 2 \text{ clock cycles} \end{aligned}$$

or 2 ns.

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

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

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

$$\begin{aligned} \frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} &= \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} \\ &= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2} \end{aligned}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

Average memory access time is the average time to access memory considering both hits and misses and the frequency of different accesses; it is equal to the following: AMAT = Time for a hit + Miss rate * Miss penalty. Understand this if miss rate is given then multiply that with miss penalty and if hit rate is given then multiply it with hit time.

Example : Suppose that in 1000 memory references there are 40 misses in L1 cache and 10 misses in L2 cache. If the miss penalty of L2 is 200 clock cycles, hit time of L1 is 11 clock cycle, and hit time of L2 is 15 clock cycles, the average memory access time will be _____ clock cycles.

Answer : According to standard method where you use 1 + something does not apply here. First, we find hit rate because hit time is given. 0.96 x 1 + 0.04 x (0.99 x 15 + 0.01 x 200) = 1.634. 1+ something method will give 1.68 you should try that also.

7.4.4) *Reducing Cache Misses by More Flexible Placement of Blocks* :

Direct mapped, where a block can be placed in exactly one location, is at one extreme. At the other extreme is a scheme where a block can be placed in any location in the cache. Such a scheme is called **fully associative**, fully associative cache - A cache structure in which a block can be placed in any location in the cache. The middle range of designs between direct mapped and fully associative is called **set associative**. set-associative cache - A cache that has a fixed number of locations (at least two) where each block can be placed. Increasing the associativity of cache often improves the hit rate of the cache, since there generally will be fewer conflict misses due to collisions. Thus, this will reduce the latency of overall operation. In fully associative, only one index is present so we not require extra bit to represent it so index field is zero.

In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of sets in the cache)

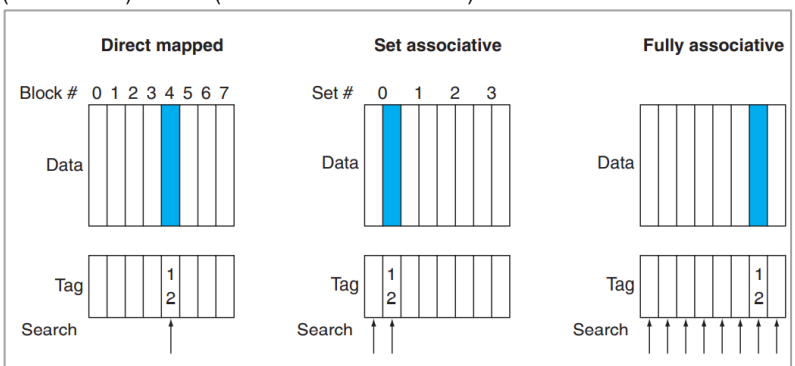
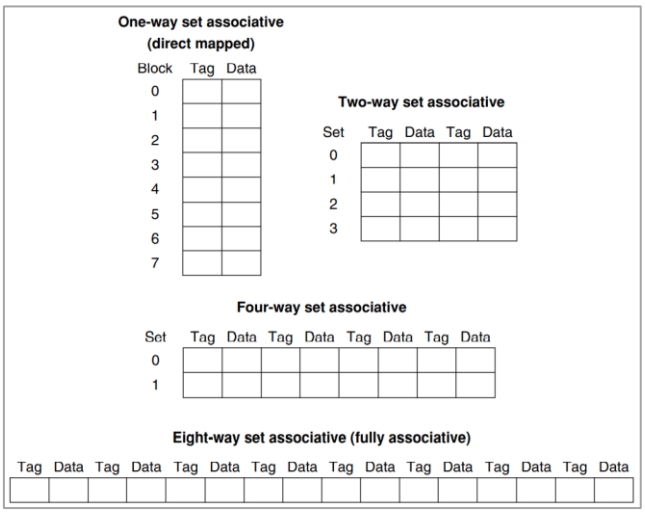


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.



Example **:** Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set-associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Answer : Direct Mapped :

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

2-way Set associative mapped :

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Fully Associative mapped :

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

In fully associative set bit is empty so only offset and tag field is there.

Note : k-way set means per set there will be k block. It does not mean k set.

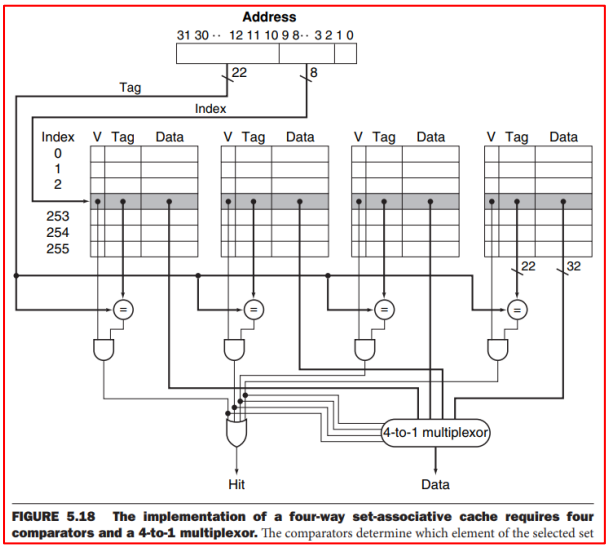


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set

Locating a Block in the Cache :

Now, let's consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 5.17 decomposes the address. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

Above picture is correct imagination. There is also one set of bit also called LRU bit which is one per index irrespective of k-way. All other bits like dirty and valid will be k per index.

Choosing Which Block to Replace :

least recently used (LRU) : A replacement scheme in which the block replaced is the one that has been unused for the longest time. For a 2 way set associative cache, each set contains two blocks (block 0 and block 1). We need one bit per set to identify which of these blocks is the LRU block. Note that if block 0 is the LRU block, then block 1 is the MRU block, and vice versa. If the cache was 4 way set associative, each set needs to record the LRU block, the second-LRU block, …, MRU block. There are 4! = 24 permutations of 4 blocks, and hence the cache needs L = ceil($\log_2 4!$) = 5 LRU bits per set.

Reducing the Miss Penalty Using Multilevel Caches :

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

$$\frac{100 \text{ ns}}{0.25 \frac{\text{ns}}{\text{clock cycle}}} = 400 \text{ clock cycles}$$

The effective CPI with one level of caching is given by

Total CPI = Base CPI + Memory-stall cycles per instruction

For the processor with one level of caching,

Total CPI = 1.0 + Memory-stall cycles per instruction = 1.0 + 2% × 400 = 9

With two levels of caching, a miss in the primary (or first-level) cache can be satisfied either by the secondary cache or by main memory. The miss penalty for an access to the second-level cache is

$$\frac{5 \text{ ns}}{0.25 \frac{\text{ns}}{\text{clock cycle}}} = 20 \text{ clock cycles}$$

If the miss is satisfied in the secondary cache, then this is the entire miss penalty. If the miss needs to go to main memory, then the total miss penalty is the sum of the secondary cache access time and the main memory access time.

Thus, for a two-level cache, total CPI is the sum of the stall cycles from both levels of cache and the base CPI:

Total CPI = 1 + Primary stalls per instruction + Secondary stalls per instruction
= 1 + 2% × 20 + 0.5% × 400 = 1 + 0.4 + 2.0 = 3.4

Thus, the processor with the secondary cache is faster by

$$\frac{9.0}{3.4} = 2.6$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache ((2%−0.5%) × 20 = 0.3). Those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory access time, are (0.5% × (20 + 400) = 2.1). The sum, 1.0 + 0.3 + 2.1, is again 3.4.

Size of Tags versus Set Associativity

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a 4-word block size, and a 32-bit address, find the total number of sets and the total number of tag bits for caches that are direct mapped, two-way and four-way set associative, and fully associative.

Since there are 16 (= 2^4) bytes per block, a 32-bit address yields 32−4 = 28 bits to be used for index and tag. The direct-mapped cache has the same number of sets as blocks, and hence 12 bits of index, since $\log_2(4096) = 12$; hence, the total number is (28−12) × 4096 = 16 × 4096 = 66 K tag bits.

Each degree of associativity decreases the number of sets by a factor of 2 and thus decreases the number of bits used to index the cache by 1 and increases the number of bits in the tag by 1. Thus, for a two-way set-associative cache, there are 2048 sets, and the total number of tag bits is (28−11) × 2 × 2048 = 34 × 2048 = 70 Kbits. For a four-way set-associative cache, the total number of sets is 1024, and the total number is (28−10) × 4 × 1024 = 72 × 1024 = 74 K tag bits.

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 28 bits, leading to 28 × 4096 × 1 = 115 K tag bits.

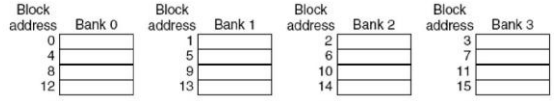
Caches, TLBs, and virtual memory may initially look very different, but they rely on the same two principles of locality, and they can be understood by their answers to four questions:

Question 1:	Where can a block be placed?
Answer:	One place (direct mapped), a few places (set associative), or any place (fully associative).
Question 2:	How is a block found?
Answer:	There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table).
Question 3:	What block is replaced on a miss?
Answer:	Typically, either the least recently used or a random block.
Question 4:	How are writes handled?
Answer:	Each level in the hierarchy can use either write-through or write-back.

Multilevel cache : A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.

Multi-banked Caches

- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
- Interleave banks according to block address:



Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

NOTE :

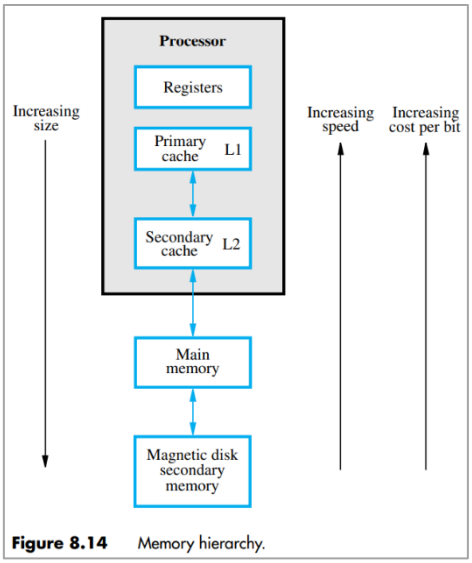


Figure 8.14 Memory hierarchy.

1) If all blocks in the higher-level cache are also present in the lower level cache, then the lower level cache is said to be inclusive of the higher-level cache.
2) Consider multiplexer latency only in set associative mapped cache.
3) Swap space is the area on a hard disk which is part of the Virtual Memory of your machine, which is a combination of accessible physical memory (RAM) and the swap space. Swap space is used when your system decides that it needs physical memory for active processes and there is insufficient unused physical memory available.
4) In set associative mapped don't forget to divide #of cache block by set given.
5) The register that stores the bits required to mask the interrupts is **Interrupt mask register**. The register that stores the bits required for Status is **Status register**. The register that stores the bits required to service the interrupts is **Interrupt service register**. The register that stores the bits required to request the interrupts is **Interrupt request register.**
6) In the Big-Endian system, the computer stores ***MSB of data in the lowest memory address*** of data unit.
7) Speed of memory units : Register > Cache > Main memory (RAM) > Electronic Disk > Magnetic Disk > Optical disk > Magnetic Tape
8) Main memory block bit = tag bit + set bit (index bit in case of direct map.)
9) A **compulsory miss** (also known as a cold miss) occur when first time a location is used, the first miss for any block reference is always treated as compulsory miss. As that block is referenced for the first time. It's not related to the fact that it is mapped to some occupied block. **Capacity miss** occurs because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity)., and a **conflict miss** happens when two locations map to the same location in the cache but if new block number is being mapped to location occupied by some other number then it will be considered was compulsory miss.
10) $$\text{MIPS} = \frac{\text{Clock rate}}{\text{CPI}_{\text{avg}} \times 10^6}$$ Million instruction per second. Note its average CPI
11) When cache uses parallel searching policy, we ignore time of cache access(with miss rate only) in average memory latency calculation.
12) Direct mapped cache is one way set associative it is not all way associative with one set. It has set equal to total number of blocks.
13) If question asks for throughput then for non-pipeline add all stage time without adding delay between stages. And for pipeline throughput = (maximum stage time) + delay between stage.
14) **CPI = CPIexecution + mem stalls per instruction**
    CPIexecution = exaptation of all instruction means their percentage x no. of cycle
    Mem Stalls per instruction = Mem accesses per instruction x Miss rate x Miss penalty
    Mem accesses per instruction = 1 + load/store percentage
15) Global miss rate L3 = local miss rate of L3 x local miss rate of L2 x local miss rate of L1. And you can produce same formula for global miss rate of L2 and L1.
16) A multi-level page table typically reduces the amount of memory needed to store page tables, compared to a linear page table. Because page table does not stores actual pages it stores ID.

7.5.2) Other Enhancement :

1) **Write Buffer** : To improve performance, a Write buffer can be included for temporary storage of Write requests. The processor places each Write request into this buffer and continues execution of the next instruction. The Write requests stored in the Write buffer are sent to the main memory whenever the memory is not responding to Read requests.

When a new block of data is to be brought into the cache as a result of a Read miss, it may replace an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the processor has to wait for this operation to be completed before the new block is read into the cache.

2) **Prefetching** : Following a Read miss, the processor has to pause until the new data arrive, thus incurring a miss penalty. To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache shortly before they are needed in the program. Then, the processor will not have to wait for the referenced data as in the case of a Read miss. This is the solution for compulsory miss.

3) **Lockup-Free Cache** : A cache that can support multiple outstanding misses is called **lockup-free**. Such a cache must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses.
4) **Split Cache** : By this point we are considering unified cache. i.e. instruction and data are present in same cache memory. Split cache is essentially two small caches, one for instruction and one for data. The benefit is that the processor can read instructions while the data cache is busy, which generally improves overall latency on average. But if data size is same for instruction and data then latency will remain same. Because same bytes/time. Instruction and data are nothing but sequence of 0's and 1's only.

5.5) Virtual Memory :

If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as **virtual memory**. The binary addresses that the processor issues for either instructions or data are called **virtual** or **logical addresses**. Virtual address can be larger than main memory address space.

A special hardware unit, called the **Memory Management Unit (MMU),** keeps track of which parts of the virtual address space are in the physical memory. When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory. Such transfers are performed using the DMA scheme.

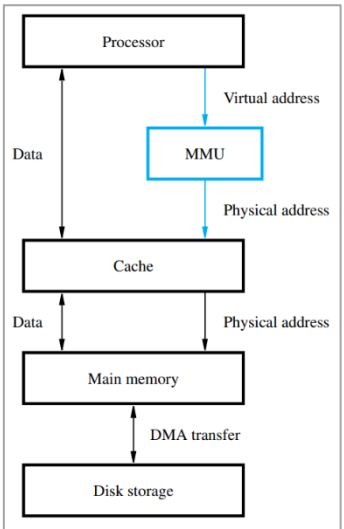


Figure 8.24 Virtual memory organization.

7.6.1) Address Translation :

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. Note that if pages are too large, it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory.

Each virtual address generated by the processor, whether it is for an instruction fetch or an operand load/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.

Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. Another bit indicates whether the page has been modified during its residency in the memory.

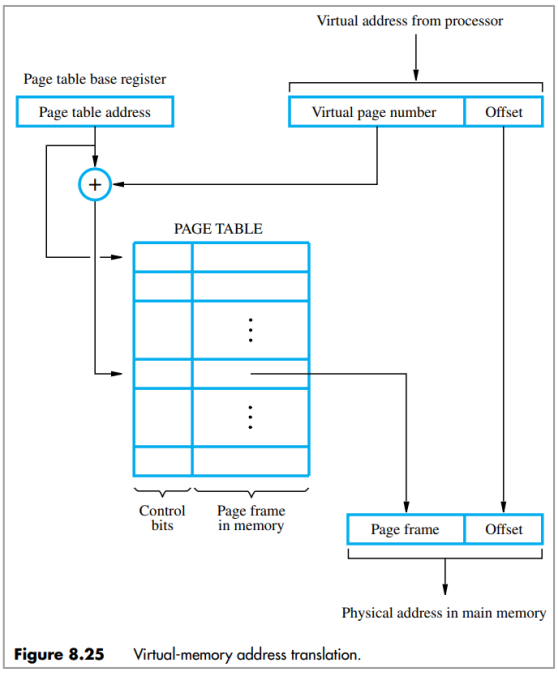


Figure 8.25 Virtual-memory address translation.

7.6.2) Translation lookaside buffer : Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory. The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the Translation Lookaside Buffer (TLB).

Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the 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.

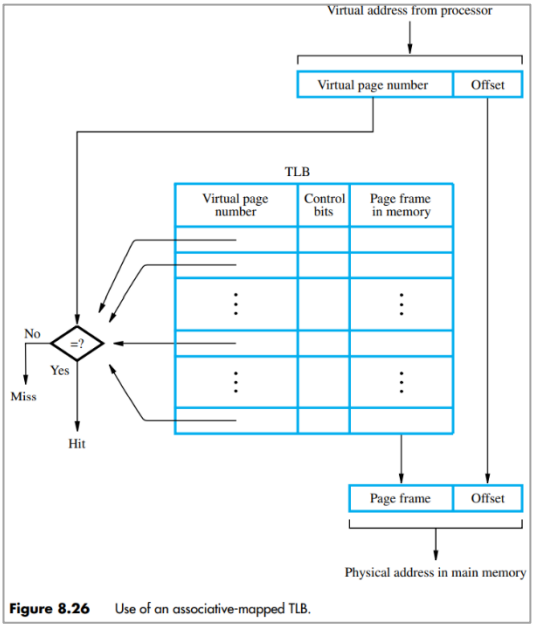


Figure 8.26 Use of an associative-mapped TLB.

7.6.3) Page Faults : When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The entire page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the program that generated the page fault is interrupted, and control is transferred to the operating system. The operating system copies the requested page from the disk into the main memory. Since this process involves a long delay, the operating system may begin execution of another program whose pages are in the main memory. When page transfer is completed, the execution of the interrupted program is resumed. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system periodically clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

Memory management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the ***system space****, that is separate from the virtual space in which user application programs reside. The latter space is called the* ***user space****. In fact, there may be a number of user spaces, one for each user.*

Let us first consider the most basic form of protection. Most processors can operate in one of two modes, the ***supervisor mode*** *and the* ***user mode****. The processor is usually placed in the supervisor mode when operating system routines are being executed and in the user mode to execute user programs. In the user mode, some machine instructions cannot be executed. These are* ***privileged instructions****. They include instructions that modify the page table base register, which can only be executed while the processor is in the supervisor mode. Since a user program is executed in the user mode, it is prevented from accessing the page tables of other users or of the system space.*

Some Privileged instruction :

1) Set the timer value
2) Change mapping from virtual to physical address. This requires TLB excess which is hardware in OS.
3) Clear all RAM
4) Changing base register or limit register.

Question on Cache and Main Memory :

1) A hard disk with a transfer rate of 10 Mbytes/second is constantly transferring data to memory using DMA. The processor runs at 600 MHz, and takes 300 and 600 clock cycles to initiate and complete DMA transfer respectively. If the size of the transfer is 20Kbytes, what is the percentage of processor time consumed for the transfer operation?

5.0% *0.5%*

1.0% *0.1%*

Answer : Total time = initiation + transfer + termination

And question asks us for (initiation + termination)/total time… remember here DMA module is doing transfer operation. So in transfer processor will be absent.

(2microsec/2microsec+2000microsec) × 100 = 0.0999 ≃ 0.1%

2) A system designer has put forth a design for a direct mapped cache C. Suppose that reading a memory address A is anticipated to have an overall expected average latency T(A, C) [including the average cost of cache miss on C]. Which of the following statements is are true?

- If C contains several words per cache line, then the index of the cache line for A is given by the rightmost bits of A.
- Suppose C is unified cache and CO is a comparable split cache with the same total capacity and cache line size as C. Then, generally T(A, C0) < T(A, C).
- Suppose C00 is a two-way set associative cache with the same total capacity and cache line size as C. Then, generally T(A, C00) < T(A, C).
- None of the above

Answer : first option is incorrect as index in direct mapping is given by middle bits of address A. second option is false

3) A byte addressable computer has a small data cache capable of holding sixteen 32 bit words. Each cache block consist of four 32 bit words. For the following sequence of addresses (in hexadecimal). The miss ratio if 4 way set associative LRU cache is used is _____. (Upto 1 decimal places)

100, 104, 108, 104, 107, 108, 105, 102, 108, 103

Answer : ***0.1***

$$\text{Number of blocks in cache} = \frac{\text{Cache size}}{\text{Cache block size}} = \frac{16 \times 32}{4 \times 32} = 4$$

$$\text{Number of blocks in set} = \frac{\text{Total number of blocks}}{\text{Number of way associative}} = \frac{4}{4} = 1$$

i.e., one set contains 4 cache blocks.

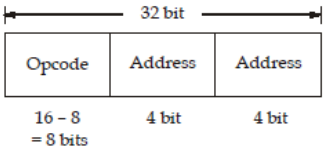
so, it is fully associative means we can place data on any one of the 4 blocks and size of one block is 4 x 32bits = 16bytes.

So, block can place data from 0-F (total of 16 for one block), 10 – 1F, 20 – 2F,…, 100 – 10F,…

Now we insert element so first is 100 in hexadecimal and we have miss as initially cache is empty. So, we go into main memory and fetch all data from 100 – 10F because block size is 16bytes. As you can see now, we have all 104, 108, 107, 105, 103 included in first block itself so we will have hit every time. Therefore, number of misses out off 10 references is 1.

4) In a 16 bit instruction format computer, the size of address field is 4 bits. The computer uses expanded opcode technique. It has four 2-address instructions and 4000 one address instruction. How many zero address instructions can be formulated _____?

Answer :



Number of operations = $2^8 = 256$

Number of free opcodes after 2-address = 256 – 4 = 252

Number of 1 address instructions = $252 \times 2^4 = 4032$

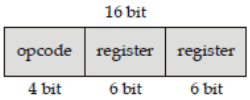
Free opcodes = 4032 – 4000 = 32

Therefore number of zero address instructions = 32×2^4

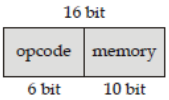
$\Rightarrow$ $2^9 = 512$

5) Consider a hypothetical CPU which supports 16 bit instruction, 64 registers and 1 KB memory space. If there exist 12, 2-address instruction which uses register reference and 12, 1-address memory reference instructions how many 0-address instructions are possible?

Answer :

1. 

2. Number of 2-address instruction = $2^4 = 16$
3. Number of free opcodes = (16 – 12) = 4
4. Number of 1-address memory reference instruction



$= 4 \times 2^2 = 16$

5. Number of free opcodes = (16 – 12) = 4
6. Number of 0-address instruction opcodes = $4 \times 2^{10} = 4096$

6) A 32-bit wide main memory unit with a capacity of 1GB is built using 256MB x 4-bit DRAM chips. The number of rows of memory cells in the DRAM chip is 2^{14}. The time taken to perform one refresh operation is 50ns. The refresh period is 2 milliseconds. The percentage (rounded to the closest integer) of the time available for performing the memory read/write operations in the main memory unit is

Answer : one refresh operation takes 50ns. There are 2^{14} rows so total they will take 50ns x 2^{14} = 819200ns
Total refresh period which include all operation including read/write operation. So, percentage of the time available for refreshing is 819200ns/20ms = 40.96%. we subtract it from hundred we will get 59%.

7) Consider a CPU with an average CPI of 1.4 when all memory accesses hit on the cache. Assume an instruction if the cache miss rate is 1.6%and miss penalty is 60 cycles, calculate the effective CPI for a unified L1 cache using write back and write allocate policy with the probability of a cache block being dirty being 0.15 (round off to 2 decimal places).

ALU	45%
LOAD	20%
STORE	20%
BRANCH	15%

Answer : See notes 14th point.
Mem access per instruction = 1 + 0.2 + 0.2 = 1.4
Mem stalls per instruction = 1.4 x 0.016 x (0.15 x 120 + 0.85 x 60)
CPI = 1.4 + 1.5456 = 2.9456 = 2.95

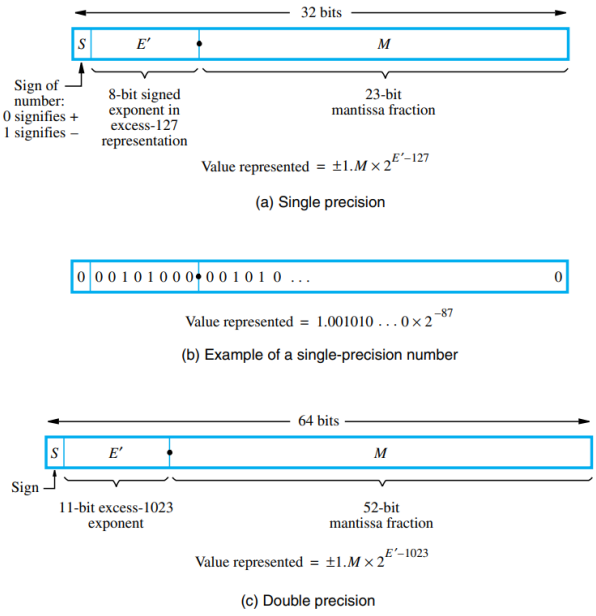


Figure 9.26 IEEE standard floating-point formats.

Questions on CPU control design and Interfaces :

1) Assume that RISC processor contains 16 global register, 12 local registers, 8 in registers and 8 out registers. Processor contain 14 register window. Number of registers prevent in the processor and windows size is

Answer : Register file size = W(L+C) + G
Window size = L + 2C + G

2) Consider the 2 GHz clock frequency processor used execute the following program segment.

Instruction	Meaning	Size (in words)
I_1 : MOV r_0, @ 3000	$r_0 \leftarrow$ m[[3000]]	4
I_2 : MOV r_1, [2000]	$r_1 \leftarrow$ m[2000]	2
I_3 : ADD r_0, [2000]	$r_0 \leftarrow r_0 +$ m[2000]	1
I_4 : Sub r_0, r_1	$r_0 \leftarrow r_0 - r_1$	1
I_5 : Mov @ 3000, r_0	m[[3000]] $\leftarrow r_0$	4
I_6 : Halt	Machine halts	1

Assume that memory reference consumers 4 cycles and ALU operations consumes 2 cycles then total time required to complete the program execution (in ns) is

Answer : if you don't understand this question go to COA + OS multitest 2022 question 28.

	IF	ID	OF	PD & WB
I_1	1 m-ref	3 m-ref	2 m-ref	–
I_2	1 m-ref	1 m-ref	1 m-ref	–
I_3	1 m-ref	–	1 m-ref	2 cycles
I_4	1 m-ref	–	–	2 cycles
I_5	1 m-ref	3 m-ref	–	2 m-ref
I_6	1 m-ref	–	–	–

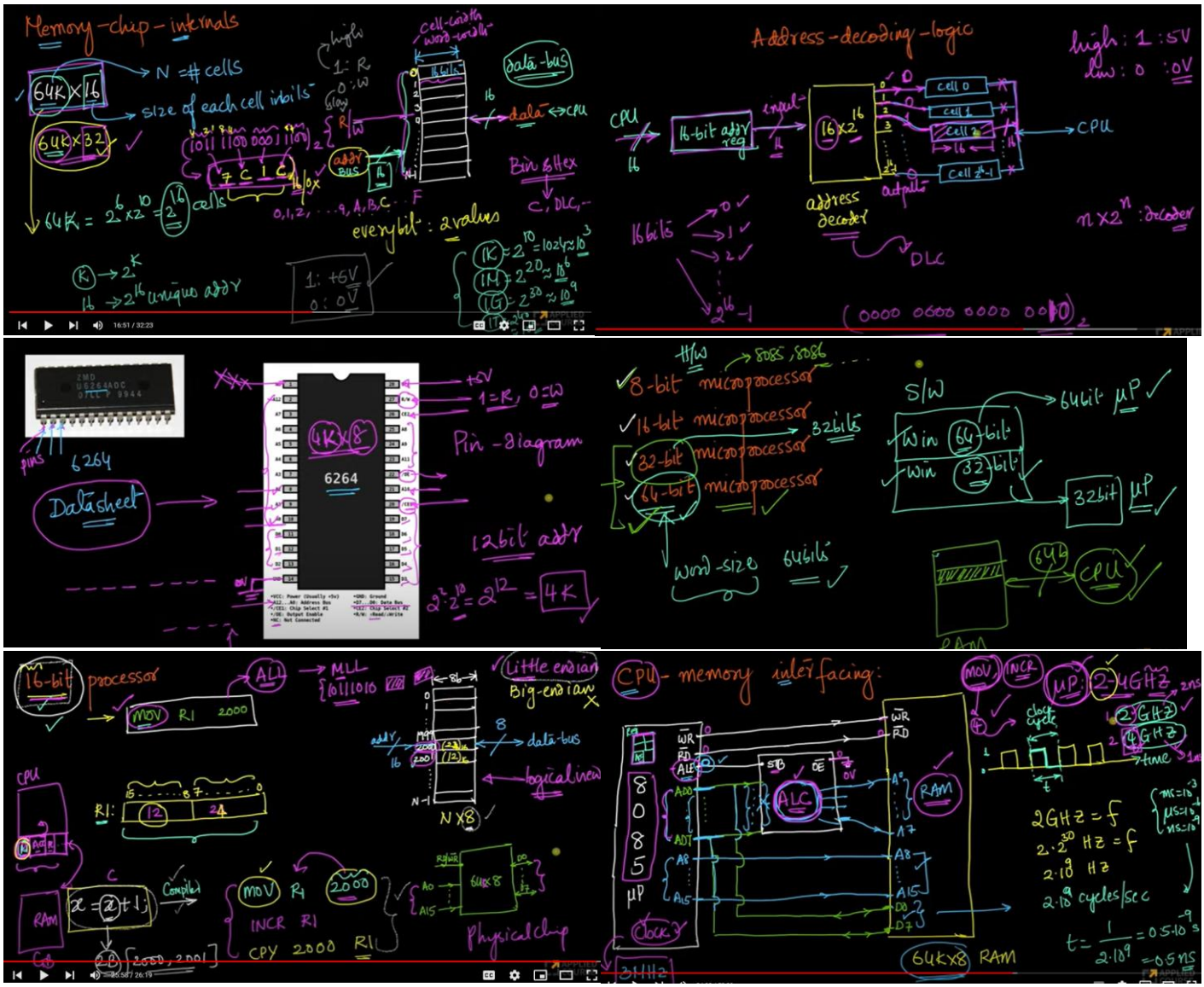
Total time = 80 cycles × 0.5 ns
= 40 ns

IF contains only 1 memory reference because it will take 1 word in IF and remaining words in ID phase. There are total 19 memory references and 2 ALU operation so in total there are (19 x 4 + 2 x 2) = 80 cycle operation.

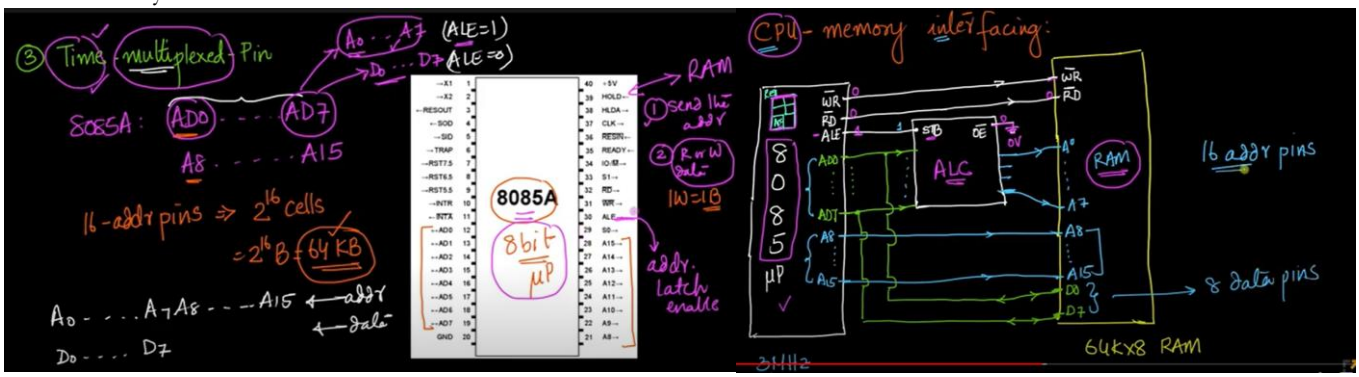
Note :

1) In word memory reference if there are n words then in IF stage we require 1 memory reference to fetch 1 word and then we do decode and after decoding we found that it is n word instruction so fetch remaining n-1 word from the memory thus another n-1 memory references.

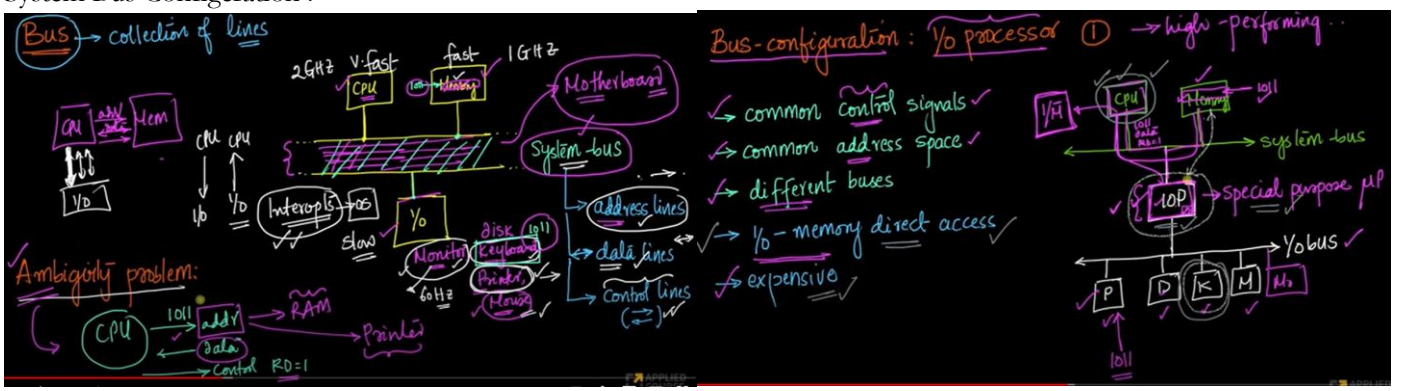
2) There is difference between register references and memory reference. As the name suggest act accordingly.

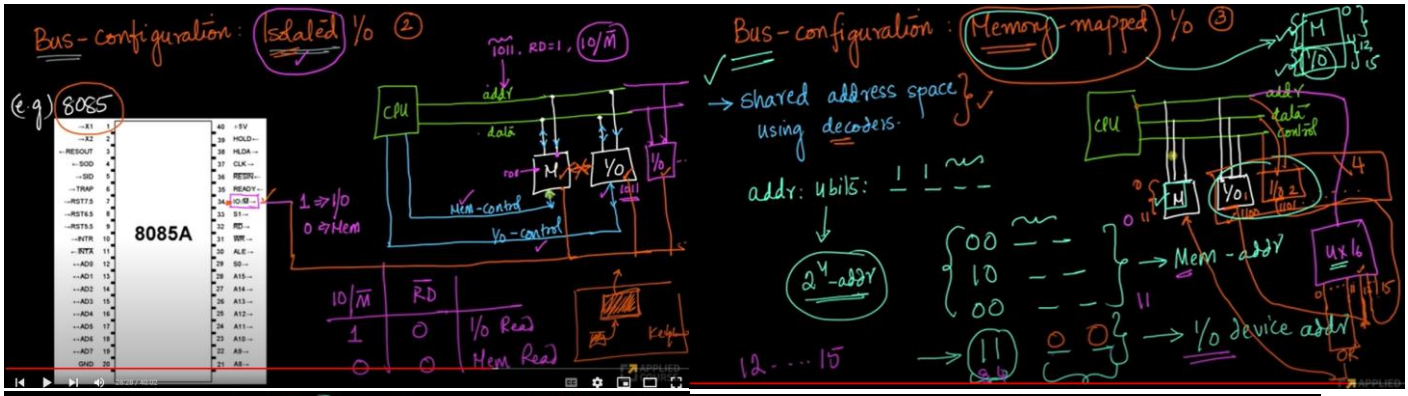


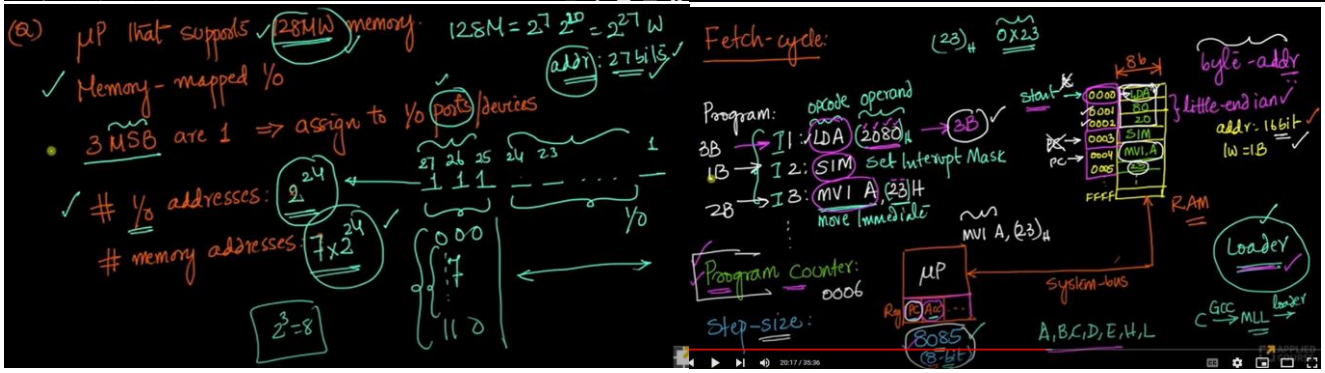
CPU Memory Interface :



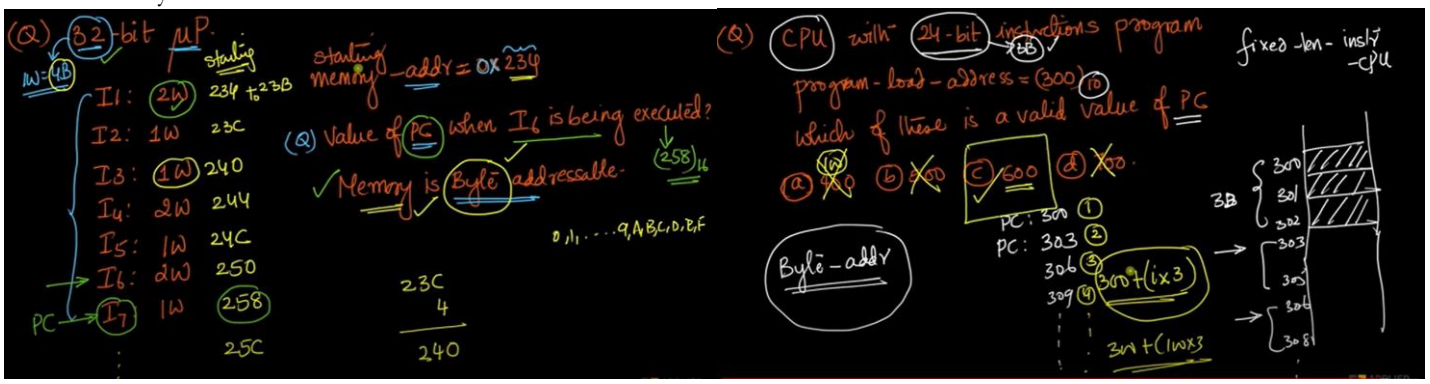
System Bus Configeration :



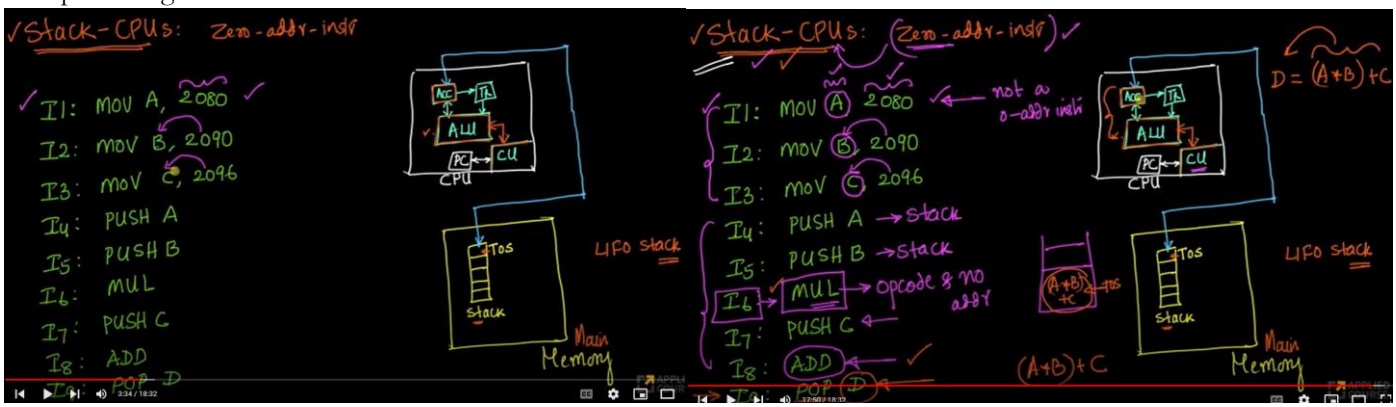




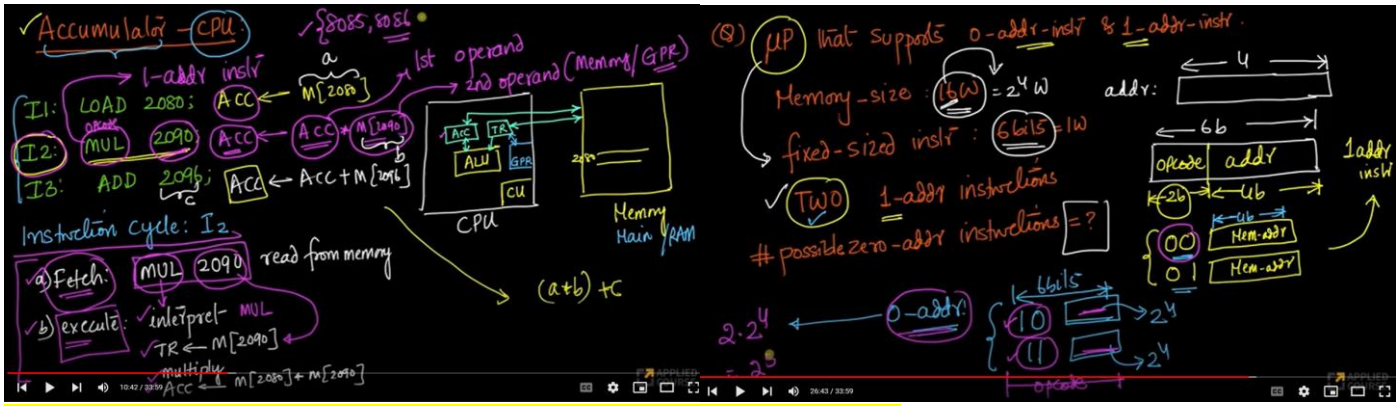
Instruction Cycle :



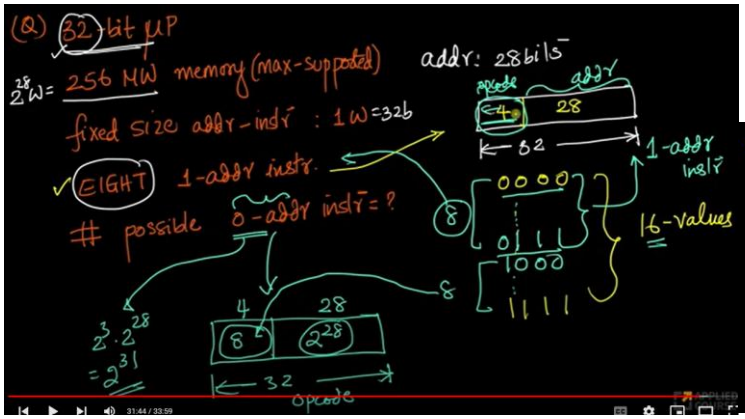
Computer Organization : Stack CPU

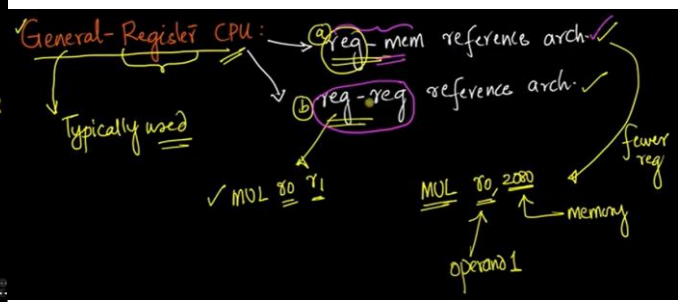


Accumulator CPU :

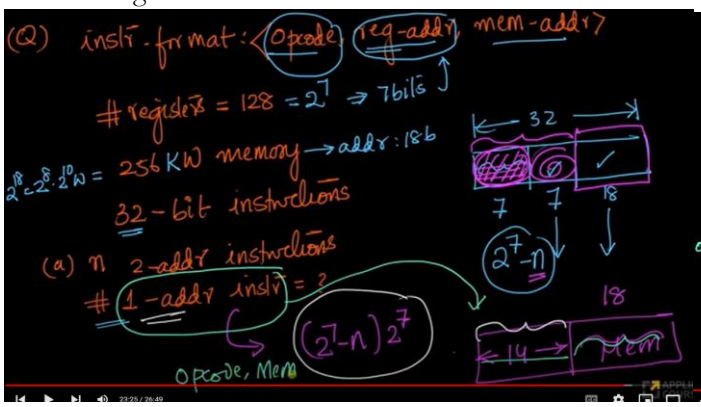


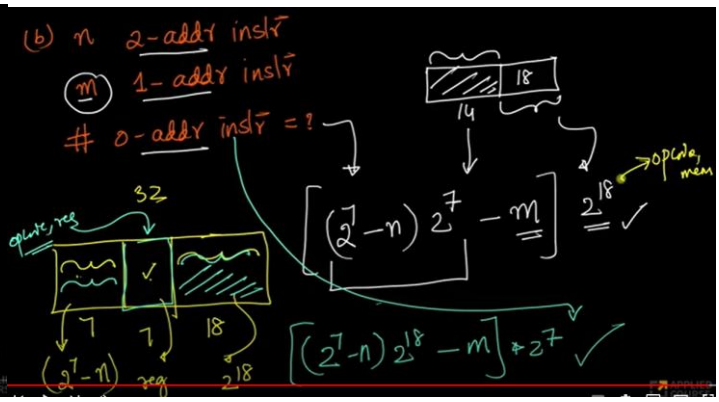
See accumulator CPU instruction. All instructions are one address instructions.





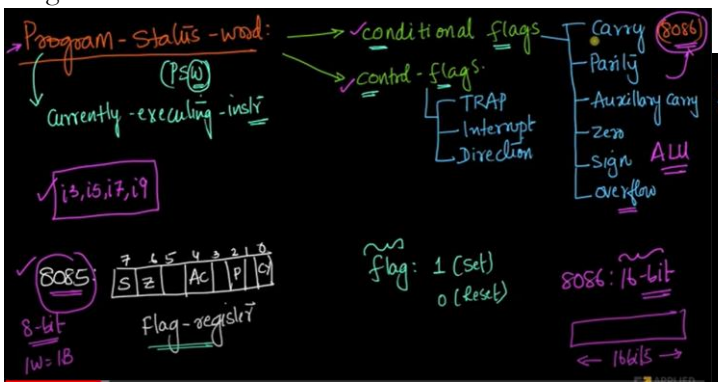
General Register CPU :

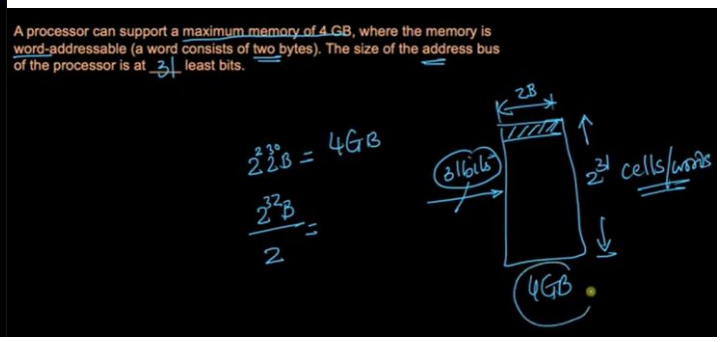




Register-Register Reference CPU : This we have covered in this word file.

Program Status Word :





More Solved Problems :