COMPUTER ORGANIZATION

COURSE OF GO CLASSES

Lectures by Abdul Kadir and Deepak Poonia

References :

- Computer Organization and embedded systems by Carl Hamacher 6th edition
- Computer organization and design by David A. Patterson 5th edition
- Computer Organization and Architecture by Willam Stallings 10th edition
- Computer System Architecture by M. Morris Mano 3rd edition
- Some University's slides

1. PIPELINING

//Lecture 1

1.1) INTRODUCTION TO PIPELINING :

Multiple instructions are overlapped in execution.

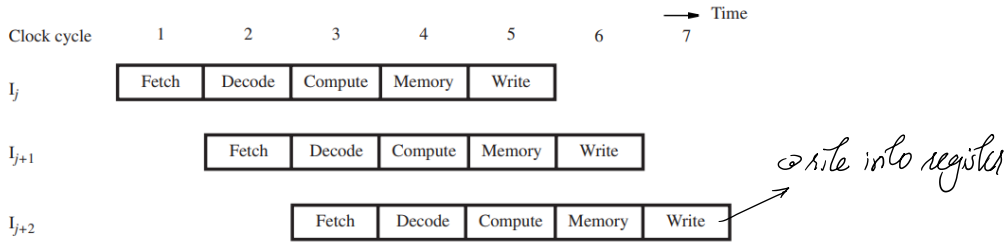
Step	Action
1	Fetch an instruction and increment the program counter.
2	Decode the instruction and read registers from the register file.
3	Perform an ALU operation.
4	Read or write memory data if the instruction involves a memory operand.
5	Write the result into the destination register, if needed.

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

Consider, instruction $I_2 : lw\ r_1, 2r_2$

IF I_2 ; ID Read Reg R_2 ; ALU $R_1 + 2$ → address ;
Access data memory → $M[R_2+2]$ write back in register

Here non-pipeline execution means single cycle execution. Meaning in non-pipeline we consider datapath as a single unit executing one instruction at a time.



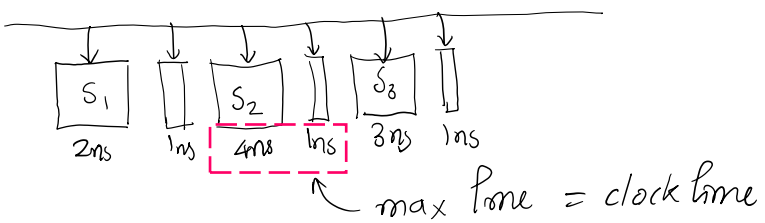
Suppose, one instruction takes Tn times and there are y of them. Then in non-pipeline total time = $y \times T_n$.

But in case of pipelining suppose k step sequence then total time = first instruction time + 1 stage time for remaining = $(k + y - 1) \times \frac{T_n}{k}$

$$\textbf{Speedup} = \frac{T_{orig}}{T_{new}}$$

For large no. of instructions, speedup = $\frac{y \times T_n \times k}{(k+y-1) \times T_n} = k$

But this is in idea scenario where each stage takes $\frac{T_n}{k}$ time. But in real life we take maximum delay a stage can offer.



Throughput : No. of instruction executed per unit time.

//Lecture 2

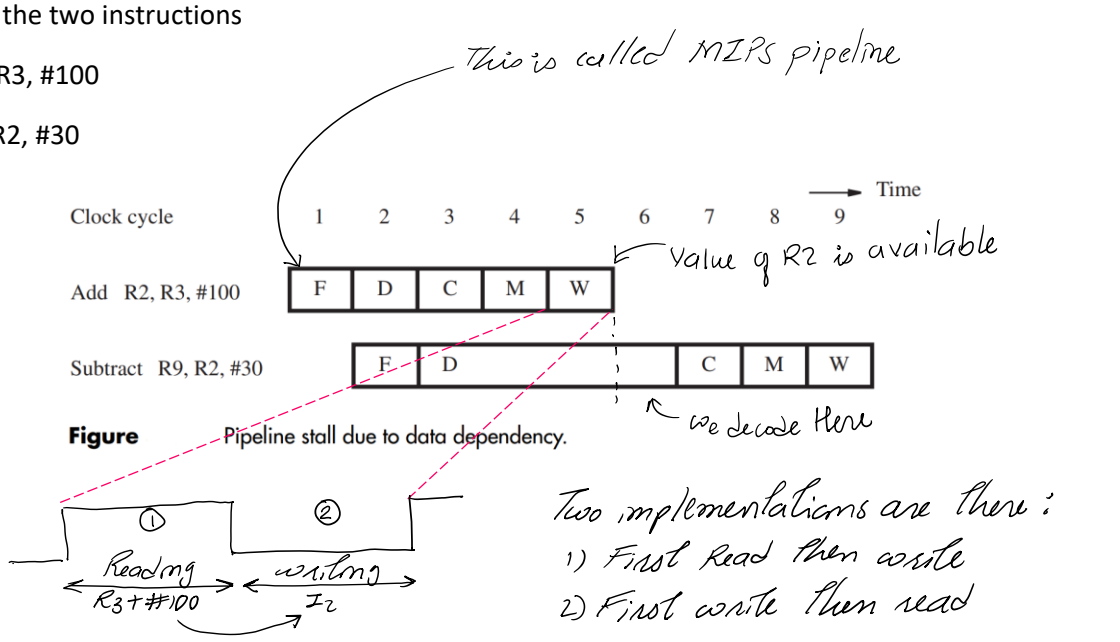
In ideal situations we don't consider interstage buffer delay in non-pipeline execution.

1.2) HAZARDS :

Consider the two instructions

ADD R2, R3, #100

SUB R9, R2, #30

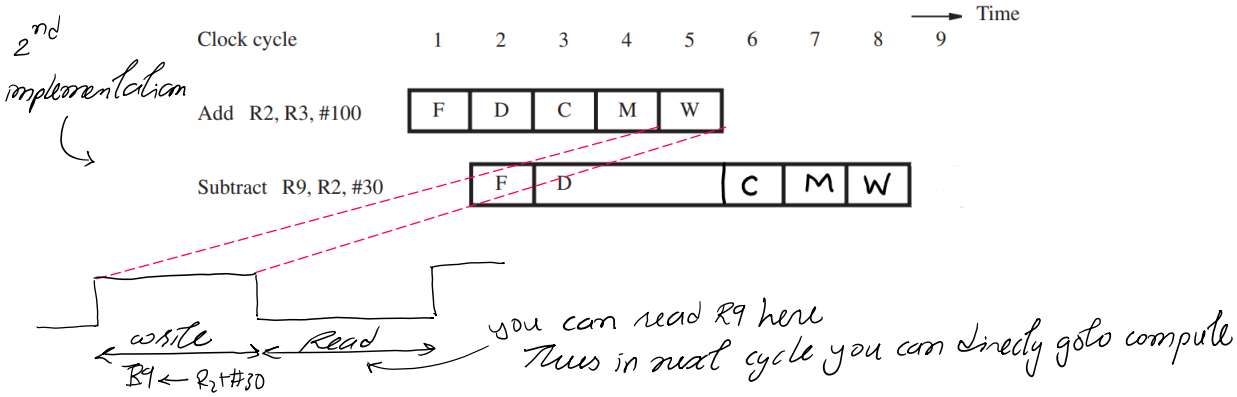


Any condition that causes the pipeline to stall is called **hazard**. We just have described data hazard, where the value of a source operand of an instruction is not available when needed.

There are other types of hazards like structural, control hazards.

1.2.1) DATA HAZARDS :

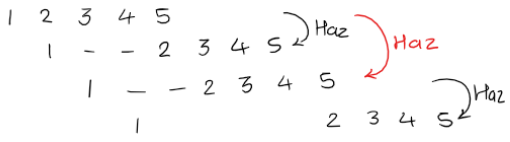
If we have used first write then read implementation then value of R2 will be available just after 1st half cycle in that case we can directly compute the value of R2 + #30 in next cycle.



This write+read is called **split phase**. And by default, we consider read+write.

If split phase is used then consider following example,

$I_1 : R_1 \leftarrow R_2 + R_3$
$I_2 : R_3 \leftarrow R_1 - R_2$
$I_3 : M[R_1 + 1000] \leftarrow R_1$
$I_4 : R_2 \leftarrow R_3 * R_1$



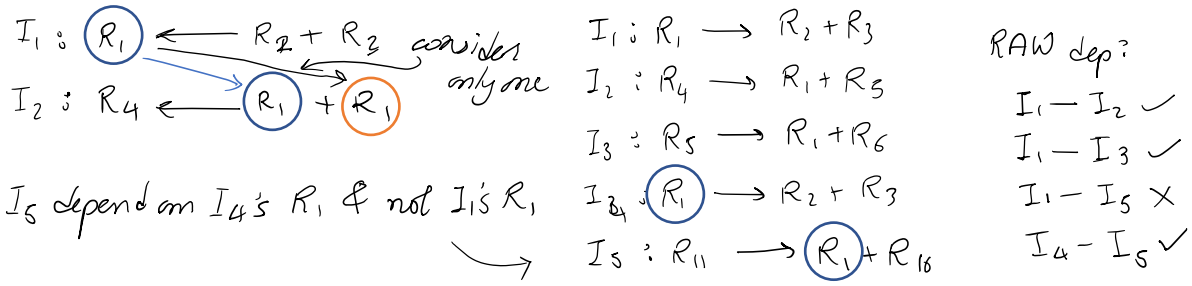
I1 – I3 is dependency causing hazard because it affects How long I3 will go. If cycle of each stage of I1 increase then it will affect I3 because of dependency.

//Lecture 3

1) DATA DEPENDENCY :

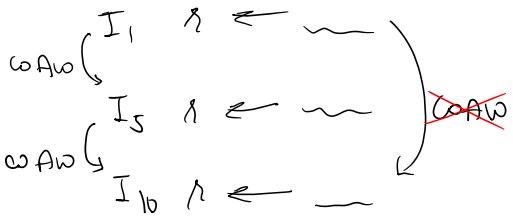
Data dependency is property of code only, it is not property of pipelining.

RAW data dependency : Read after write dependency/ flow dependency

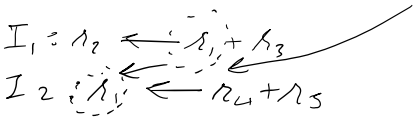


We say a dependence exists between i and j if j reads the result produced by i, and there is no instruction k which occurs between i and j and that produced the same result as i.

WAW data dependency : WAW dependence exists between a and b if both writes but there is no instruction k which occurs which also writes.



WAR data dependency :



RAW → only there can create problem
WAW } → Never create any problem (Stalls) in our simple issues
WAR
anti-dependency

Data hazards are property of code + pipeline

NOTE : Not all data dependencies are data hazards. WAW, WAR dependencies are never a hazard in any single-issue, in-order pipeline (MIPS).

//Lecture 4

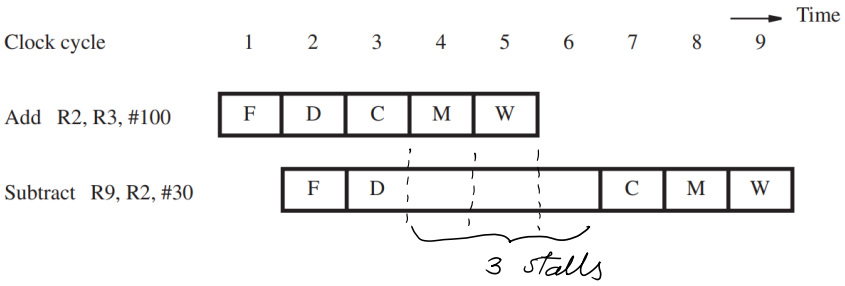
2) HANDLING DATA HAZARDS :

- Don't read wrong value at the first place (Avoid doing mistakes in sort) – **stall**
- Fix the wrong value that you have read (Rectify your mistakes if you have done) – **forwarding**

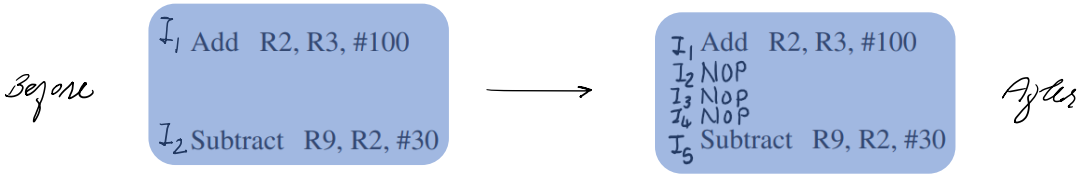
STALL WAY SOLUTION : Again, there are two ways namely hardware-based and software-based solution

- **Hardware solution** : just stall and read correct value

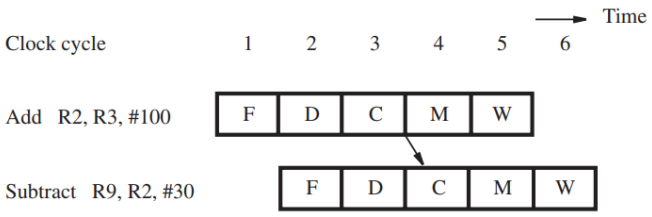
We will take by default, read+write in Write phase,



- **Software solution** : In hardware solution, it's responsibility of hardware to introduce stall where as in software solution is compiler/system's responsibility. Here system simply puts NOP instruction (telling do nothing) to introduce stall.



OPERAND FORWARDING :



We connect output of ALU to input of ALU so that it can be used by upcoming instruction who wants to use them.

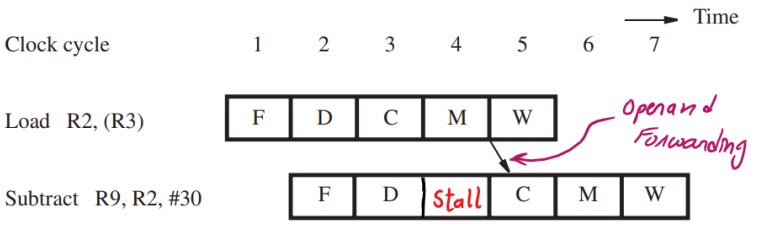
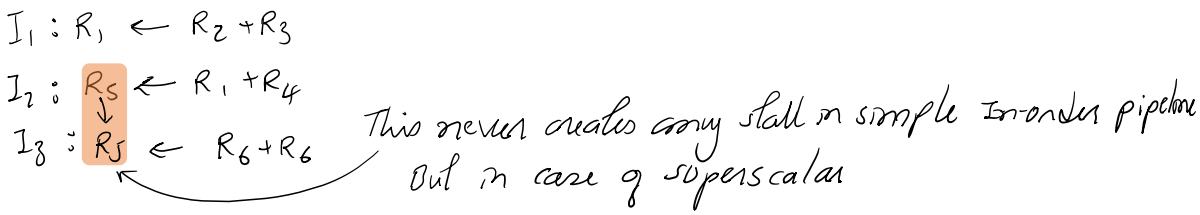


Figure Stall needed to enable forwarding for an instruction that follows a Load instruction.

3) WAR, WAW HAZARDS AND REGISTER RENAMING :

SUPERSCALAR SYSTEM : you can simultaneously fetch many instructions and simultaneously can do ID, EX, MM, WB.



Here I1 and I2 have RAW dependency and thus in superscalar approach I2 stays in ID phase where as both I1 and I3 goes to EX phase thus, final value of R5 will be written by I2 instruction which is false because I3 should write. Thus, all dependency creates problem in superscalar system.

But all WAW, WAR and RAR can be eliminated completely while RAW still remains problem.

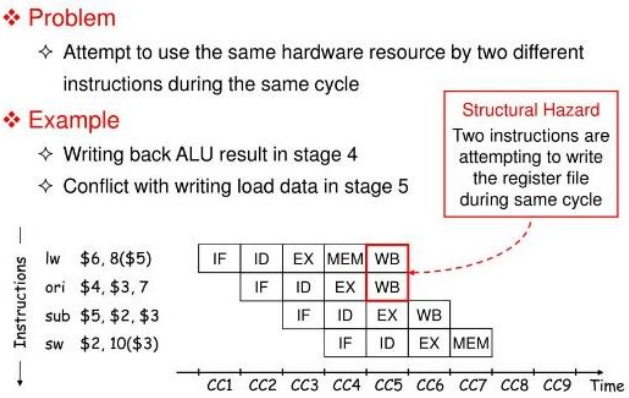
ELIMINATING WAW AND WAR :

Never write in same register. Just do renaming

//Lecture 5

1.2.2) STRUCTURAL HAZARDS :

This type of hazards is created because of lack of resources. Suppose we have only one MM.



Similarly, we can have case where one instruction is executing IF (this requires access to MM) and other instruction is executing MEM (this also requires access to MM) in same cycle. Two instructions accessing same resource in same cycle, this is called structural hazards.

SOLUTION : Put additional hardware (i.e. separate memory for Instruction and data), putting Stall/NOP instruction. Easily curable.

1.2.3) CONTROL HAZARDS :

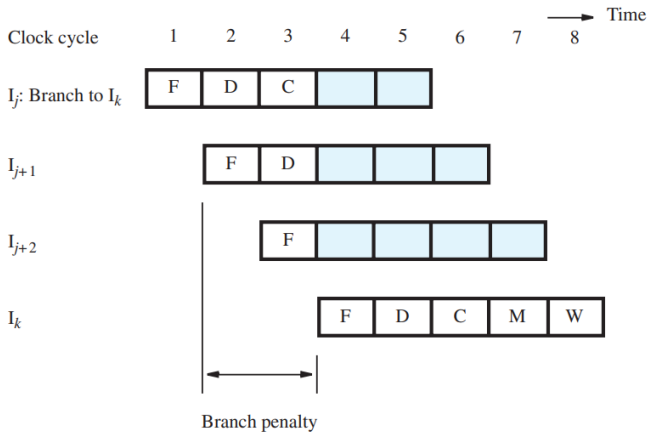


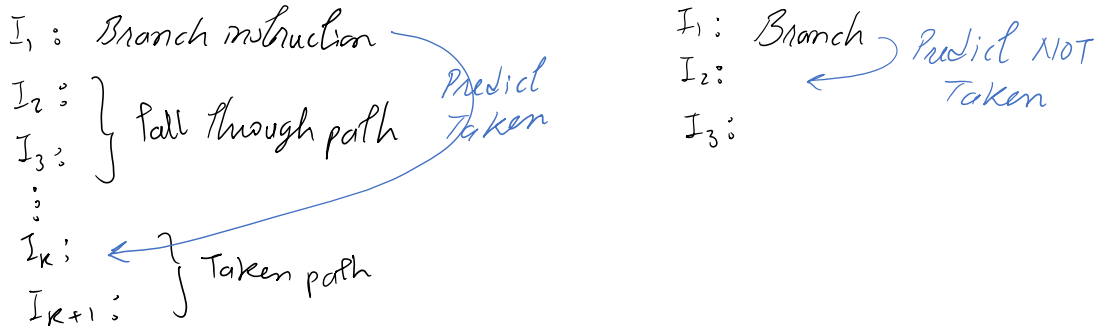
Figure Branch penalty when the target address is determined in the Compute stage of the pipeline.

In above we are resolving branch (finding branch target) at 3rd stage thus getting 2 stalls. In general, if branch is resolved in k^{th} stage, then we will get k – 1 stalls.

Solution :

1) Stop fetching until the branch target is resolved. **Type 1** : After Every instruction, stop fetching until in ID stage we get to know if the instruction is branch or not. Means we will always have 1 stall.
   Type 2 : In IF stage we can put extra hardware which will tell us branch or not. This is by default.
2) Next solution is static branch prediction…

Static Branch Prediction : why after "every" branch stop fetching ? may be branch condition results into false then there would be unnecessary stall, instead keep fetching, if why problem we flush or stall.



Predict not taken means we don't care we will just take next instruction, for every branch we consider as branch is "not taken". If prediction fails then flush or k-1 stalls (if branch resolved in k^{th} stage).

In predict taken in IF stage only we will get to know whether it is branch or not and we will also get to know target address without checking condition on branch instruction. Means whatever label is given we will go to that instruction without checking condition on branch instruction.

Predict not taken is easiest to implement.

Branch Delay Slots :

Assume that the branch target address and the branch decision are determined in the Decode stage, at the same time that instruction I_{j+1} is fetched. The branch instruction may cause instruction I_{j+1} to be discarded, after the branch condition is evaluated.

The location that follows a branch instruction is called the *branch delay slot* (because there will be only one stall due to resolution of branch at decode stage).

Independent instruction

```
        Add              R7, R8, R9
        Branch_if_[R3]=0 TARGET
        I_{j+1}
        ⋮
TARGET: I_k
```

(a) Original sequence of instructions containing a conditional branch instruction

```
        Branch_if_[R3]=0 TARGET
        Add              R7, R8, R9
        I_{j+1}
        ⋮
TARGET: I_k
```

(b) Placing the Add instruction in the branch delay slot where it is always executed

The compiler attempts to find a suitable instruction to occupy the delay slot, one that needs to be executed even when the branch is taken. It can do so by moving one of the instructions preceding the branch instruction to the delay slot.

If a useful instruction is found, then there will be no branch penalty. If no useful instruction can be placed in the delay slot because of constraints arising from data dependencies, a NOP must be placed there instead. In this case, there will be a penalty of one cycle whether or not the branch is taken.

Thus, this independent instruction will execute even after branch because it has nothing to do with branch instruction. This technique is called **delayed branching**.

Q : What does it mean by conditional branch is handled by predicted to be taken(evaluated in ID stage) ? – It means when variable used in it must be available at ID stage of branch instruction if not then it will wait for EX of previous stage to complete and then decode.

//Lecture 20

1.3) FLOATING POINT REPRESENTATION :

Numerical form : $(-1)^s \times 1.M \times 2^E$ ← **implicit representation** because it has higher precision than explicit. In case of **explicit** → 0.M (Implicit bit)

Limitation of fractional binary numbers :

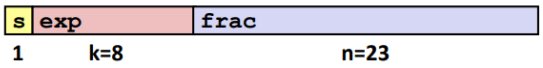
- Can only exactly represent numbers that can be written as x * 2^y
- Other rational numbers have repeating bit representation.
- IEEE754 single precision has 24 bits of precision i.e. it can correctly represent any integer which can be precisely represented in ≤24 bits.

Representation in memory :

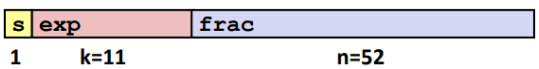
MSB s is sign bit **s**, **Exp** field encodes **E** (but is not equal to E), **frac** field encodes **M** (but is not equal to M)

comes under IEEE

- **Single precision: 32 bits**



- **Double precision: 64 bits**



"Normalized" means the mantissa M has the form 1.xxxxx

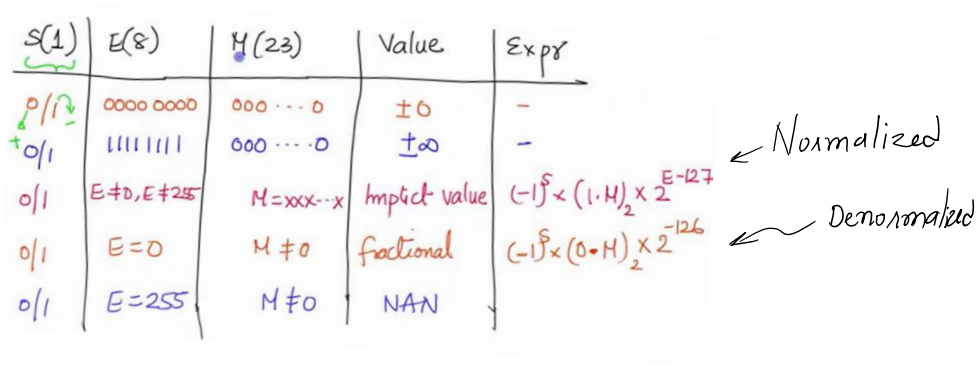
Since we know the mantissa starts with a 1, we don't bother to store it. This is only in case of normalized.

If there are k bits to exp then range of exp field is -2^{k-1} to $(2^{k-1}-1)$

Thus, we write **Actual exp = E +** $(2^{k-1}-1)$

1.3.1) IEEE 754 FLOATING POINT STANDARD :

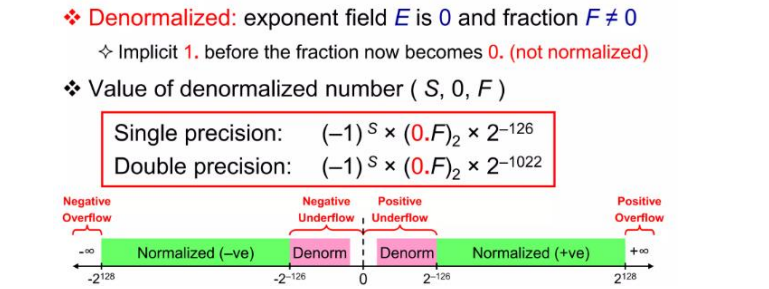
We can say **IEEE 754 is excess 127** because we are adding 127 to E to get actual exponent.

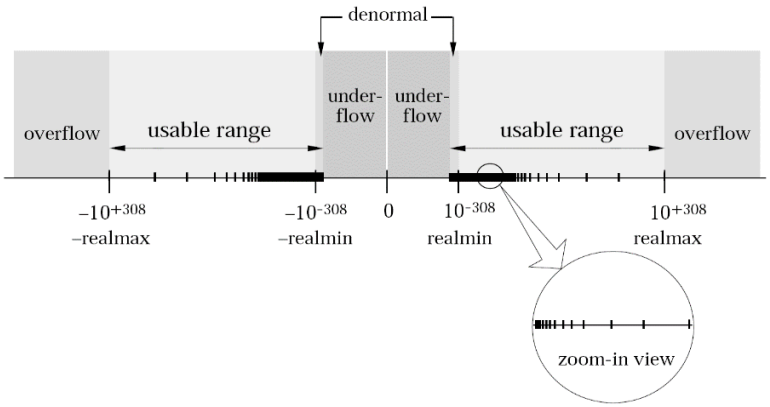


Unlike the representation for integers, the representation for floating-point numbers is not exact.

DENORMALIZED VALUES :

- **Condition** : exp = 000…0
- **Exponent value** : E = exp – Bias + 1 (instead of E = exp – Bias)
- **Significant codded with implied leading 0** : M = 0.xxxx$_2$





1.3.2) MATHEMATICAL PROPERTIES OF FP OPERATIONS :

If overflow of the exponent occurs, result will be ∞ or $-\infty$

Floating point operations are not always associative or distributive, due to rounding!

(3.14 + 1e10) - 1e10 != 3.14 + (1e10 - 1e10)

1e20 * (1e20 - 1e20) != (1e20 * 1e20) - (1e20 * 1e20)

Floating Point Multiplication

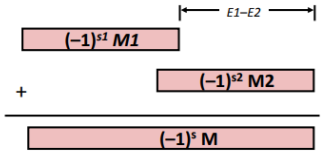
$(-1)^{s1}$ M1 2^{E1} * $(-1)^{s2}$ M2 2^{E2}

- **Exact Result:** $(-1)^{s}$ M 2^{E}
  - Sign s: s1 ^ s2 // xor of s1 and s2
  - Significand M: M1 * M2
  - Exponent E: E1 + E2
- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision

Floating Point Addition

$(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2} *Assume E1 > E2*

- **Exact Result:** $(-1)^{s}$ M 2^{E}
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1



- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit frac precision

Different condition :

- If we transfer bits from mantissa to exponents then we can represent more numbers.
- If we transfer bits from exponents to mantissa then we can improve precision of numbers.

2. Machine instruction and addressing modes

//Lecture 1

Memory is divided into equal parts called **cell**. Each cell is having a unique address. Each cell is associated with two operation namely read/write.

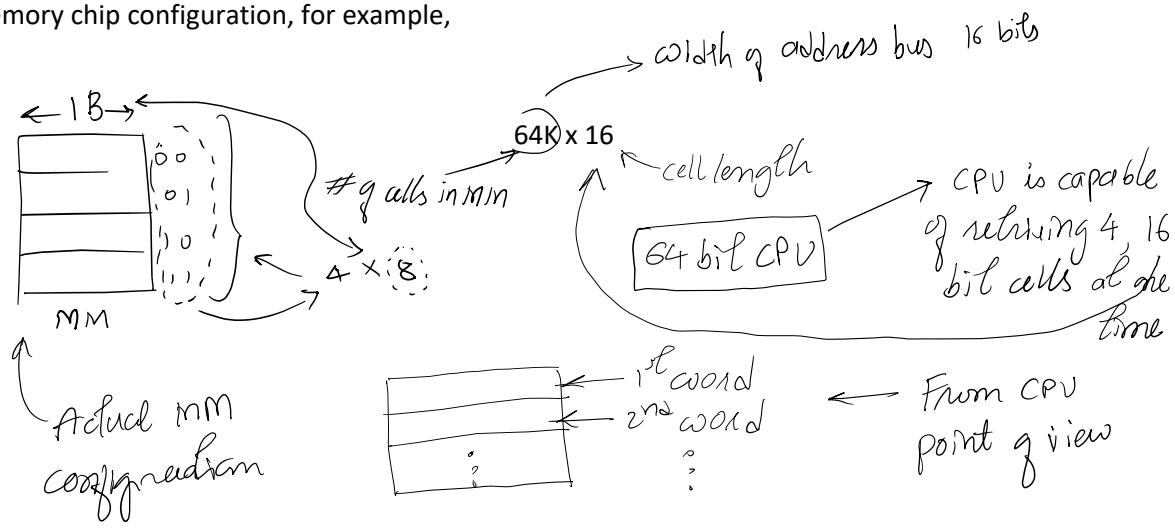
2 types of Memory : RAM (volatile – each time new), ROM (non-volatile – can't be erase)

CPU generates memory request to access data/instruction from MM. CPU request through control signal. For example, Memory_request : 1000, RD' (read operation)

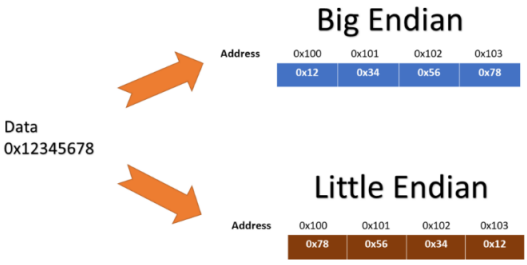
1002, WA, 20 (write operation) … this means CPU will send 20 on data bus, write control signal to control line and 1002 on address line.

2.1) INTRODUCTION TO INSTRUCTION : Set of primitive operations executed by H/W.

Memory chip configuration, for example,



If CPU is byte addressable then it is not necessary to have word length 8 bit




```
struct{
   int      a;     //0x1112_1314                  word
   int      pad;   //
   double   b;     //0x2122_2324_2526_2728        doubleword
   char*    c;     //0x3132_3334                  word
   char     d[7];  //'A','B','C','D','E','F','G'  byte array
   short    e;     //0x5152                       halfword
   int      f;     //0x6162_6364                  word
} s;
```

Big-endian address mapping

Byte address									Byte address
	11	12	13	14					
00	00	01	02	03	04	05	06	07	00
	21	22	23	24	25	26	27	28	
08	08	09	0A	0B	0C	0D	0E	0F	08
	31	32	33	34	'A'	'B'	'C'	'D'	
10	10	11	12	13	14	15	16	17	10
	'E'	'F'	'G'		51	52			
18	18	19	1A	1B	1C	1D	1E	1F	18
	61	62	63	64					
20	20	21	22	23					20

Little-endian address mapping

									Byte address
					11	12	13	14	
	07	06	05	04	03	02	01	00	00
	21	22	23	24	25	26	27	28	
	0F	0E	0D	0C	0B	0A	09	08	08
	'D'	'C'	'B'	'A'	31	32	33	34	
	17	16	15	14	13	12	11	10	10
			51	52		'G'	'F'	'E'	
	1F	1E	1D	1C	1B	1A	19	18	18
					61	62	63	64	
					23	22	21	20	20

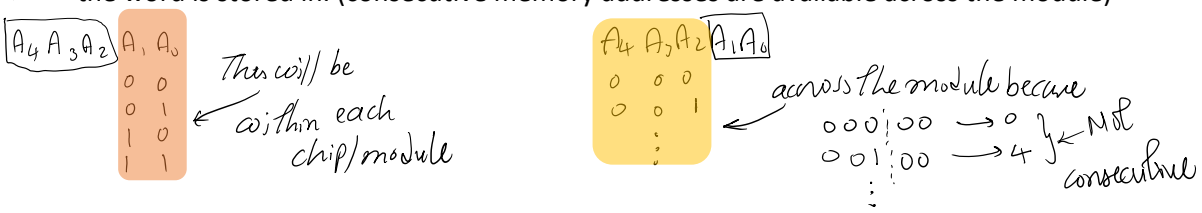
Figure 12.13 Example C Data Structure and Its Endian Maps

Memory expansion :

- *Horizontal expansion* : To increase number of bits per address location (increase word size)
- *Vertical expansion* : To increase number of address locations. (increase no. of words)

For vertical expansion we use concept of **memory interleaving** : in which we have two ways

1) ***High order memory interleaving*** : selecting MSB bits of address to determine which module the word is stored in (consecutive memory addresses are available within the module)
2) ***Low order memory interleaving*** : selecting LSB bits of address to determine which module the word is stored in. (consecutive memory addresses are available across the module)



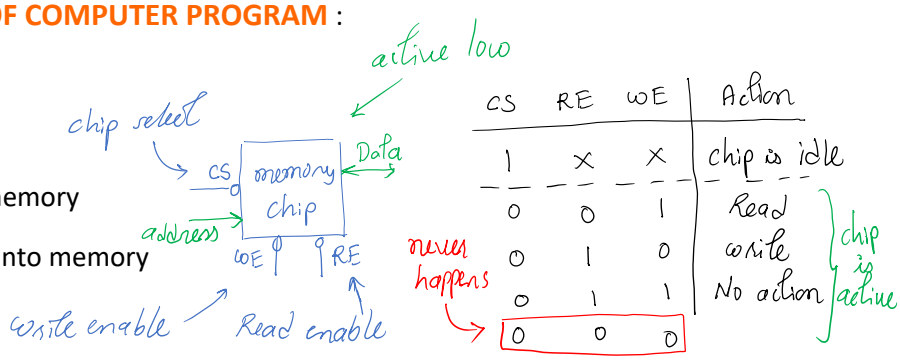
//Lecture 2

2.1.1) COMPONENETS OF COMPUTER PROGRAM :

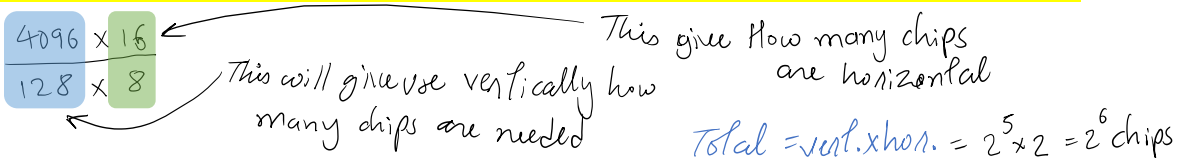
CPU, Memory and I/O

Memory read : CPU reads memory

Memory write : CPU writes into memory

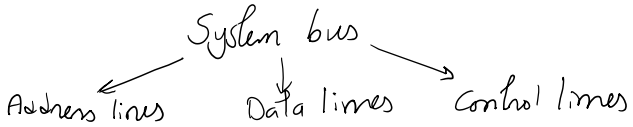


Q : How many 128 x 8 memory chips are needed to provide a memory capacity of 4096 x 16 ? –

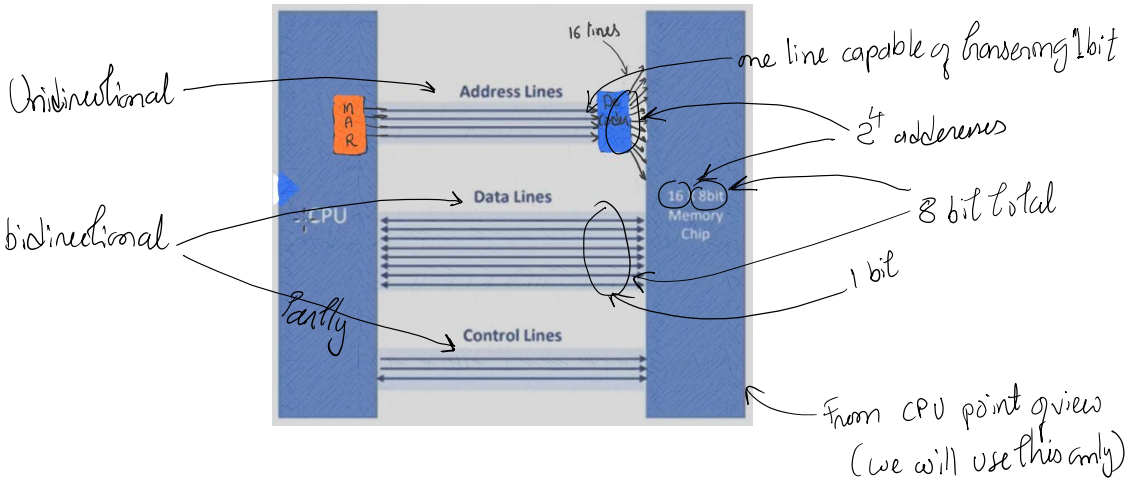


All these three components are connected with each other via system bus. CPU and main memory will be connected directly but I/O is connected through I/O interface which is the circuit.

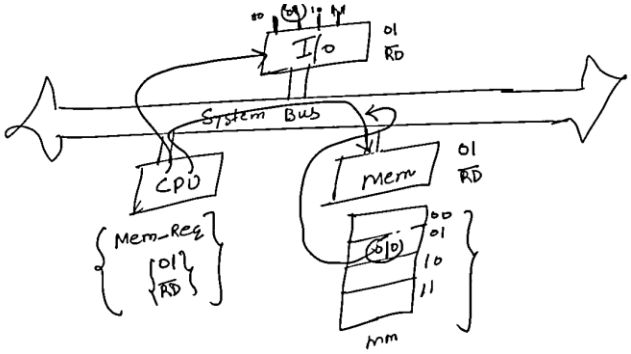
1) **SYSTEM BUS** :



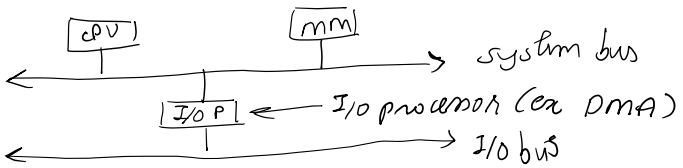
Microprocessor = processor = CPU



Q : What if CPU sends a request on system bus for read operation from 01 address ? - In that case both I/O and MM will get request and there is collision.



Solution 1 : keep I/O bus separate



We are keeping :
⇒ same address
⇒ Same control signal
⇒ Separate bus for I/O

Disadvantage : very costly

Solution 2 : Isolated I/O (I/O mapped IO)

Here we separate control lines. And introduce separate control signals for read from MM and read from I/O. Thus, when MM gets I/O read request it will reject and vice versa.

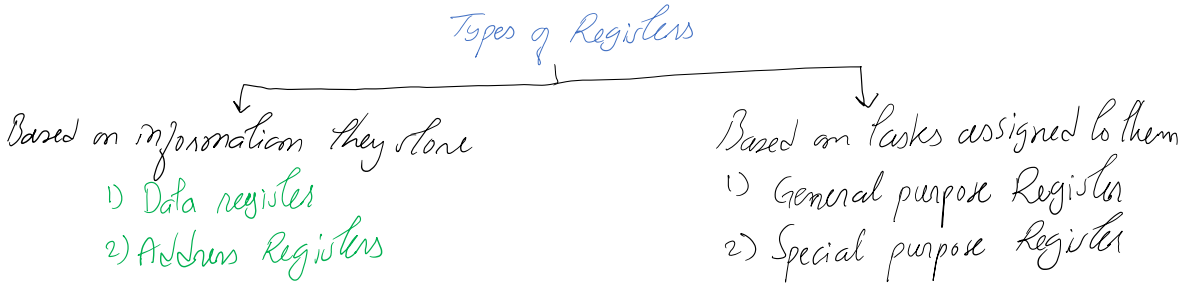
Solution 3 : memory mapped I/O

Here we disable some addresses of MM and disabled addresses = no. of I/O addresses. Here we have same control signals, Common system bus but separate address space.

Disadvantage : memory wastage.

>> 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. it will take extra time in IO mapped IO because of control signal.

2) **Registers in CPU** : Arithmetic logical unit



Special purpose registers :

Address
- Memory address register (MAR) : stores address to access memory
- I/O address register (I/O AR) :
- Program counter (PC) : stores address of next instruction to be fetch for execution. It is both a register and a counter.
- Stack control register/ Stack pointer : Stores address of TOS when CPU is executing some process.

Data
- Memory Buffer register (MBR) : stores data/instruction **to** or **from** memory
- I/O buffer register (I/O BR) :
- Instruction registers (IR) : Stores fetched instruction
- Accumulator register (AC) : Associated to ALU. Stores the result of ALU computation.
- Flag register/program status word : Reflect the result of an instruction executed by the processor (ex. ZF, PF, SF, OF). Metadata

Status register == Flag register == program status register (PSR) == program status word (PSW) == status word

NOTE : ALU doesn't know if your data is signed or unsigned or negative or positive or binary or BCD. ALU just compute result and sets the flags. It is up to control unit (or really the programmer's instructions executed by the control unit) to interpret the results.

Status bit	Name	Description
Z	Zero	Indicates result of ALU in accumulator is 0
S	Negative/Sign	MSB of result stored in accumulator
C	Carry	Set to 1 when there is carry out from the adder
A	Auxiliary carry	1 if there is a carry **out** from bit 3 on addition means C_{in} of nibble (4 bit)
P	Parity	1 if the low byte of a result has an even number of 1 bits.
O	Overflow	In 2's complement ABO' + A'B'O

Instruction	Affects flags
MOV, PUSH, POP	None
ADD/SUB	All
INC/DEC	All except C

2.1.2) BUS ARBITRATION :

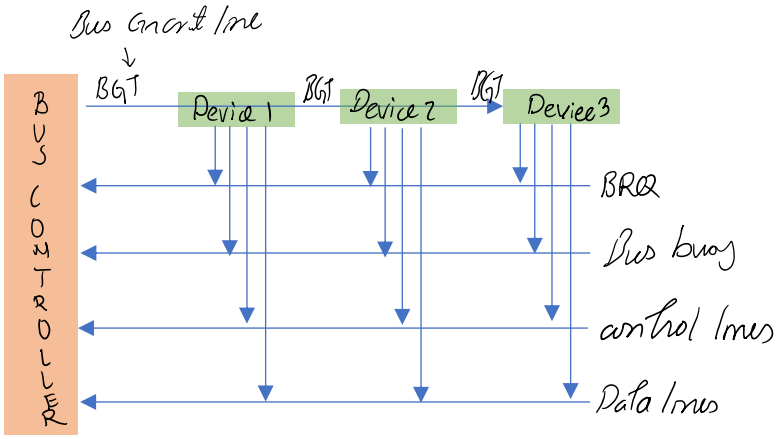
It is the process of determining who will become the next master of the system bus. Because we have common bus and multiple devices (DMA, CPU) are trying to get the access to system bus.

As only one device can be granted the access to system bus.

There are mainly three solutions : Daisy Chaining, Polling/dynamic priorities, Independent request

DAISY CHAINING :

Here we introduce one more line called bus request (BRQ) line through which devices puts their request for use of bus and bus controller will decide to which device bus should be granted.



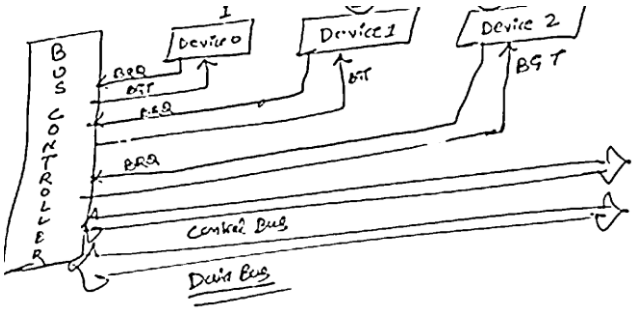
Disadvantage :

- If One node fail whole system will fail
- Unnecessary delay.
- The device that is closest to bus controller have highest priority

POLLING : nearly same as daisy chaining but we introduce some more lines called poll count. And #of poll count lines = log (# of devices connected with bus). Using poll count lines, we ask each device if they want to use bus. First one to answer will get the bus access.

INDEPENDENT BUS REQUEST AND GRANT :

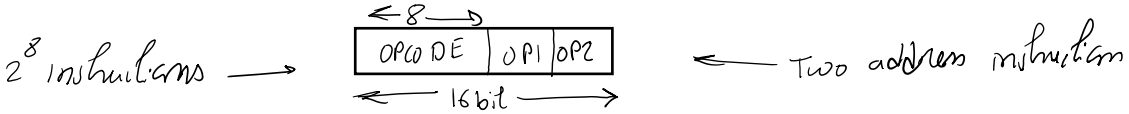
This is same as daisy chaining but we have separate BRQ and BGT for each device. And based on priority bus controller.



//Lecture 3

2.2) INSTRUCTION AND INSTRUCTION FORMATS :

Instruction is a binary sequence that is processed by CPU. Instruction consists of two parts OPCODE (mandatory) and zero or more operand.



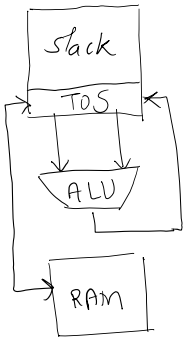
Different CPU organization have different instruction formats. Each machine supports some sets of operation ex. ADD, SUB, MUL… we call this **instruction set** (meaning set of instruction).

Instruction set architecture (ISA) : It is a precise description of how software can invoke and access time. ISA specifies what the hardware provides, not how it's implemented.

2.2.1) CPU ORGANIZATION AND INSTRUCTION FORMATS :

STACK ORGANIZATION :

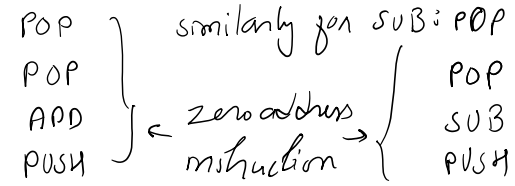
0-address instruction



PUSH : Acc → tos (top of stack)

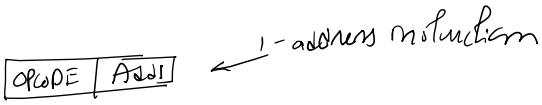
POP : tos → Acc

ADD :



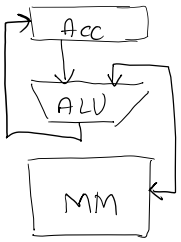
ACCUMULATOR MACHINE/ORGANIZATION :

- In this organization, one operand is always available in accumulator reg and the second operand is in memory
- After the operation compete result is stored back into accumulator
- It supports one address instruction



LOAD : Memory Read

STORE : Memory Write

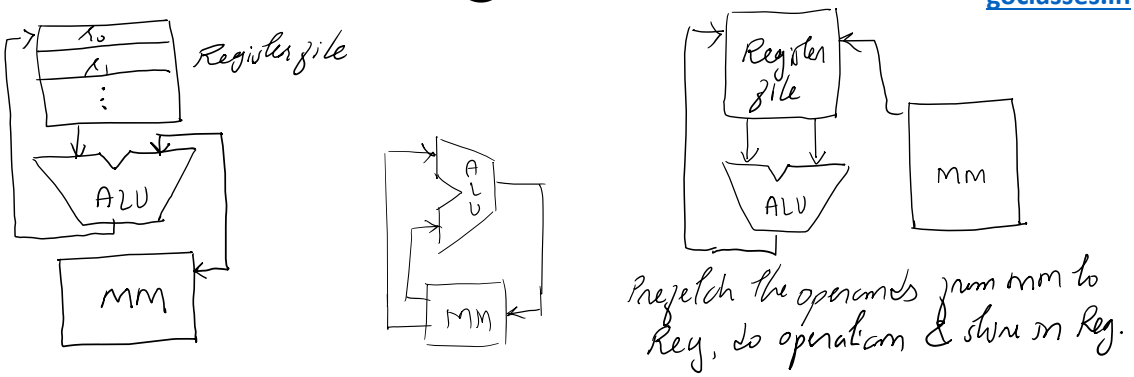


GENERAL REGISTER ORGANIZATION :

Register to memory organization :

Register to Register organization : also known as load-store archi. (RICS)

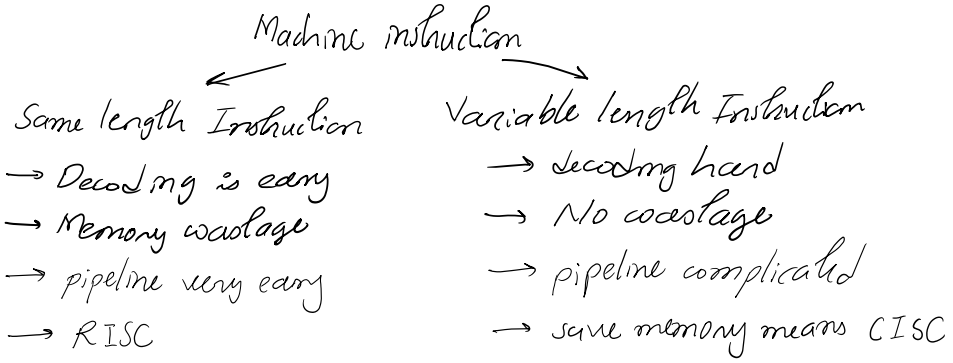
In this process supports more no. of register.



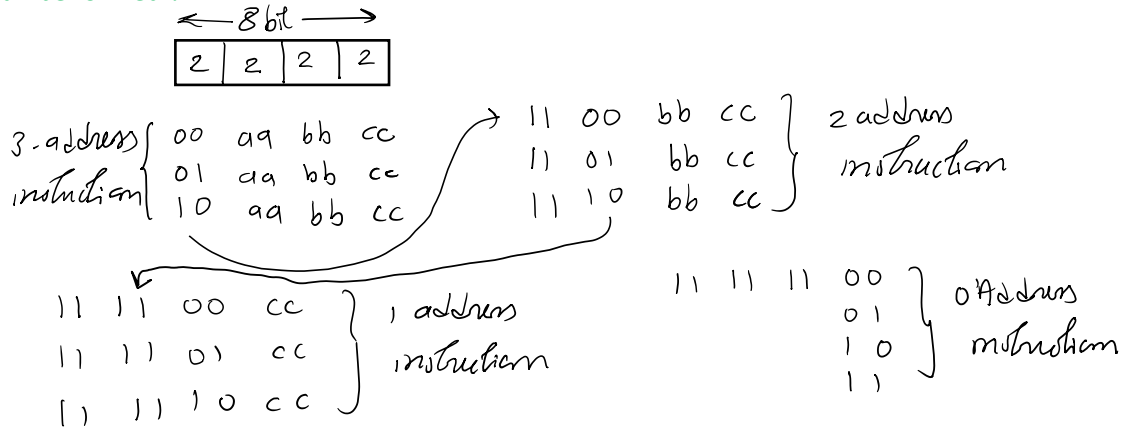
Memory to Memory organization : Most CISCs use this (no explicit operand)

//Lecture 4

2.2.2) EXPANDING OPCODE TECHNIQUE :



Q : Suppose we have 8-bit instruction and 2-bit address then how many 3, 2, 1, 0 address instruction can be formed ? –

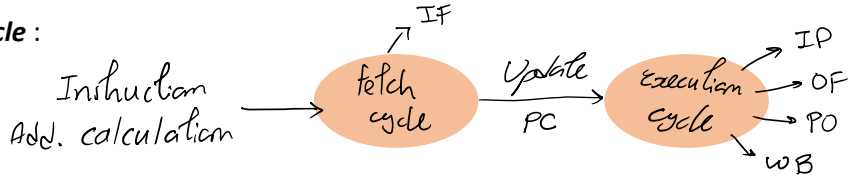


Q : If sizes of some address sequence (n sequence) of 1 program is given then what will be the total size of program ? – If it is fixed length coding scheme is used then size of program max (sizes of n sequence). And if variable length coding scheme is used then size of program will be addition of size of all instruction.

//Lecture 5

2.3) ADDRESSING MODES : Ways by which the location of operands if specified.

Instruction cycle :



INSTRUCTION FETCH IN NUTSHELL :

each generates some signals

$PC \rightarrow MAR$

$m[MAR] \rightarrow MBR$

$PC \leftarrow PC + 1$

$MBR \rightarrow IR$

each operation is called Microoperation

whole program is called Microprogram for 'fetch'

Register which stores instruction

MAR is connected with address bus and MBR is connected with data bus.

Goal : To find effective address (address of operands)

IMPLIED MODE : The Opcode definition itself defines the operand, hence no need to specify external operand.

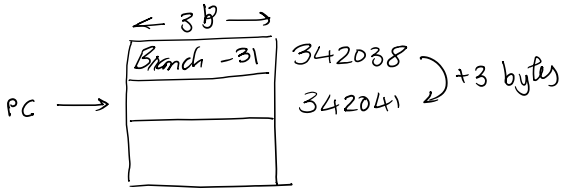
Stack addressing : Stack addressing mode is subtype of direct addressing modes in which we use stack operation. Usually, all function variables are stored on its activation record. Now, if this function is called multiple times, the activation record cannot always have the same address -- especially in case of recursion. To handle this, we store the start address of activation record in a register called **Stack Pointer** and now each invocation of a function gets its own memory space for activation record but uses same offsets for all their variables -- by using stack addressing. This is example of implied addressing.

Name	Assembler Syntax	Addressing function	Uses
Immediate	ADD r1, r2, #100	EA = Instruction itself , Operand = value	To define constants, initialize variables
Direct / Absolute	ADD m1, m2, m3	EA = value, Operand = [value]	To access local, global, static variables
Register	ADD r1, r2, #100	EA = Rn, Operand = [Rn]	When a value is in a register, small address and no memory reference
Memory Indirect	ADD m1, m2, (m3)	EA = [An], Operand = [[An]]	Used to define pointers
Register Indirect	ADD r1, m2, (r2)	EA = [Rn], Operand = [[Rn]]	Less memory reference
Displacement based			
Indexed	LOAD r6, 1000(R5)	EA = R (Index reg.) + A (base), Operand = [[R] + A]	Uses in array implementation
Relative		EA = PC + A (signed offset), Operand = [PC + A]	Uses in jump, conditional branches

Base		EA = Base + A (unsigned offset), Operand = [Base+A]	Relative, base register both are used for relocation.
Others Addressing modes			
Autoincrement	(R)+	Exactly same as register Indirect AM, but after cal. EA, content of R will be + by operand bytes	
Autodecrement	-(R)	Decrement content of R by operand byte and then find EA using content of R.	

Q : A PC-relative mode branch instruction is 3 bytes long. The address of the instruction in decimal is 342038. Determine the branch target address, if the signed displacement in the instruction is -31 –

EA = [PC − 31]
= [342041 − 31]
= 342010



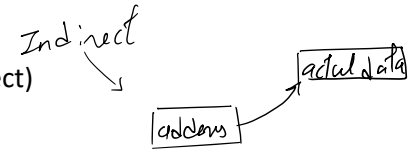
Position independent codes are independent of the memory address in which they are stored, meaning we can relocate them to other position. *Base addressing/base register and relative mode is example of position independent codes.*

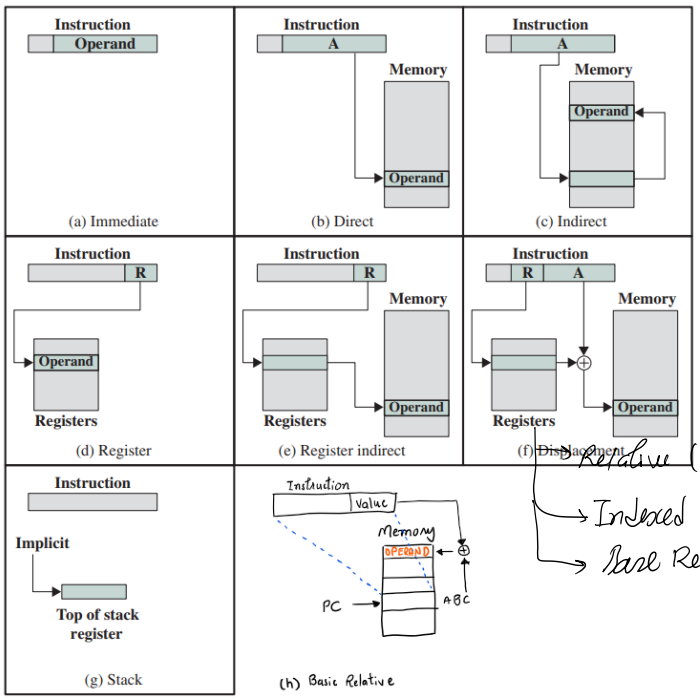
When the OS loads the program, it sets the **base register** to the physical address where the program should start and the CPU adds this value to every address that the process uses when it is executing.

//Lecture 6

Indirect cycle : (used while calculating effective address for indirect)

MAR ← IR(address)
MBR ← m[MAR]
IR(address) ← MBR





Relative (R = PC)
Indexed (R = IndexReg)
Base Reg (R = base Reg)

Q : how the control unit (CPU) knows AM of a operand ? – 2 ways. 1st : Opode itself indicates AM to control unit and 2nd : Specify in instruction such as 

//Lecture 21

2.4) DIRECT MEMORY ACCESS :

Table I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

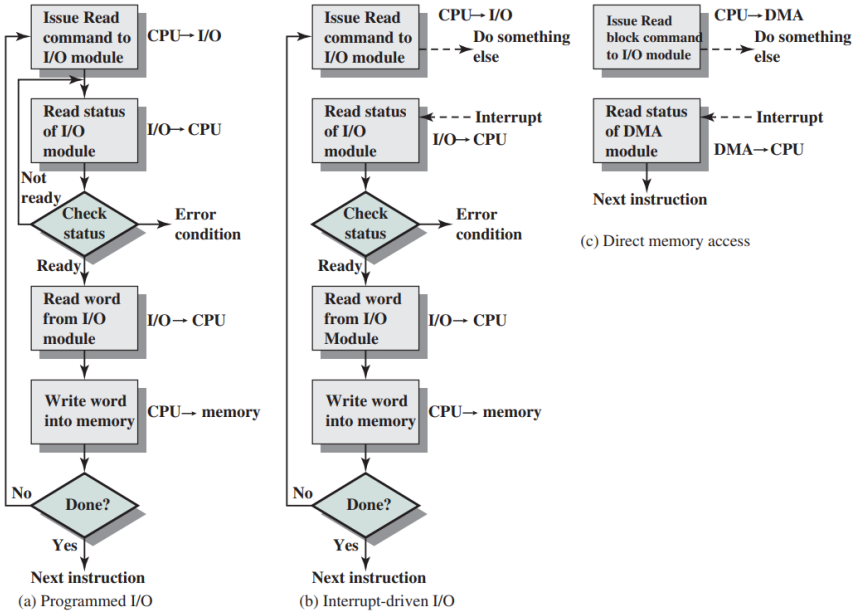


Figure Three Techniques for Input of a Block of Data

Problem with programmed I/O : The problem with **programmed I/O** is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

We use interrupt-driven I/O !

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

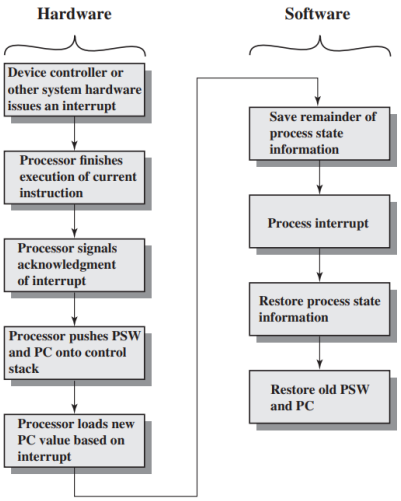


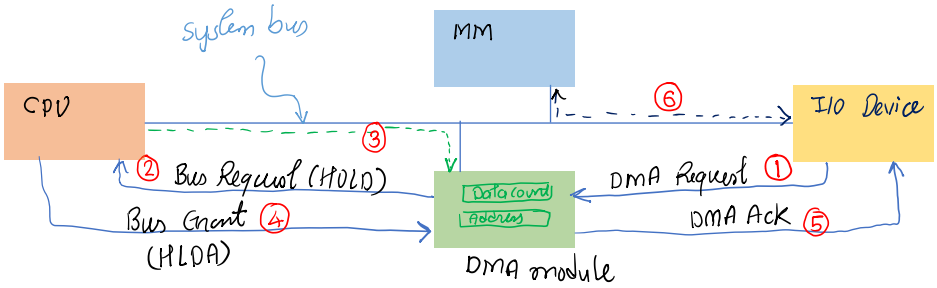
Figure 7.6 Simple Interrupt Processing

Q : interrupt-driven I/O is always faster than programmed I/O ? – If the devise transfer rate is fast then context switch happens again and again. and there is some overhead associated with context switch thus, interrupt-driven I/O is efficient but not always faster.

Problem with interrupt-driven I/O : Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

2.4.1) DMA FUNCTION :



DMA module is a special purpose processor which generates address and control signals for data transfer between memory and I/O.

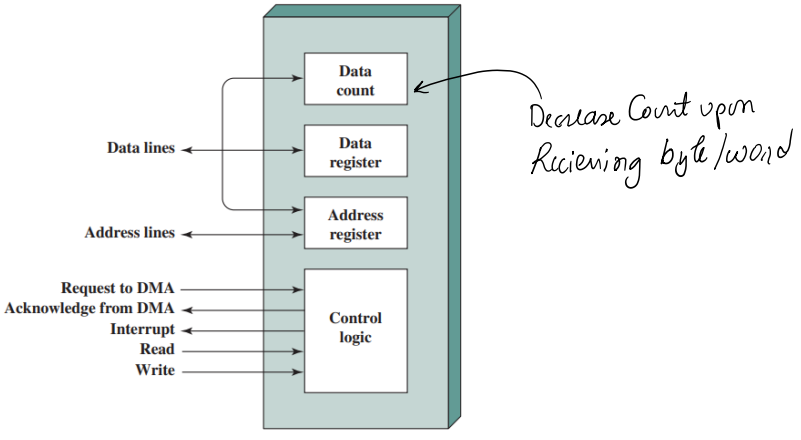


Figure . . Typical DMA Block Diagram

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 (mostly CPU will be blocked because there are very few operation without bus). 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.

Q : Should DMA transfer entire data in one time and keep CPU blocked for long time ? – for this we have modes of DMA transfer…

2.4.2) TYPES OF DMA TRANSFER :

BURST TRANSFER :

DMA returns the bus after complete data transfer. A register is used as a byte count, being decremented for each byte transfer, and upon the byte count reaching zero, the DMA will release the bus. When the DMAC operates in burst mode, the CPU is halted for the duration of the data transfer.

CYCLIC STEALING :

An alternative method in which DMA controller transfers one word at a time after which it must return the control of the buses to the CPU. The CPU delays its operation only for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

INTERLEAVING MODE : Whenever CPU perform internal operations and does not require the buses then DMAC will be given the control of the buses. Hence CPU will never be blocked due to DMA.

Let P = *time required to prepare the data in I/O*

T = *Time required to transfer data from I/O to memory*

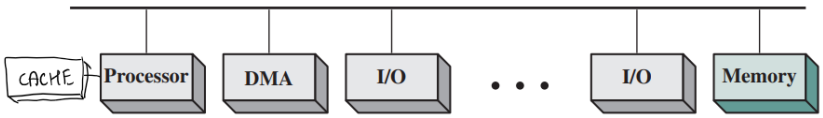
Type of mode	% CPU idle/blocked	% CPU busy
Burst mode	$\frac{T}{P+T} \times 100$	$\frac{P}{P+T} \times 100$
Cycle stealing mode	$\frac{T}{P} \times 100$	$\frac{P}{T} \times 100$

Interleaved mode	*0*	100

NOTE that DMA controller takes over bus for a cycle.

In cycle stealing when u transfer data from I/O to MM new data is parallelly prepare in I/O so prepare time will overlap with transfer time thus only X.

2.4.3) CACHE COHERENCE :



Consider, DMA is performing its function at that time DMA will be the master of system bus and at that time most of the working of processor is based on cache. Consider in RAM at location 1234 we update the value (say x). and at the same time Cache will update its value to y. Then this is not valid as at the same location one value should be there.

Cache coherence is when Multiple copies of the same data can exist in different memory simultaneously

Q : A DMA module is transferring characters to memory using cycle stealing from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1 MIPS by how much will the processor be slowed down due to the DMA activity ? –

Total 9600/8 = 1200 characters per sec are being transferred. As we are using cycle stealing and transferring character to memory. Thus, we will take 1200 cycles of processor out of 1M. thus,

Slow down = CPU idle = 1200/1M x 100 = 0.12%

Silly mistakes :

A 64-bit processor has 64 registers and uses a 20-bit instruction format. It has two types of instructions M-type and R-type. Each M-type instruction contains an opcode and a memory address. Each R-type instruction contains an opcode and two register names. Main memory is 8K words, and it is byte addressable. If there are 10 distinct M-type opcodes, then the maximum number of distinct R-type opcodes is _____

Answer : Here 64 bit processor means processor takes data or instruction in chunks of 64 bit or 8 bytes. So, 1 word = 8 byte here thus, address size = 16 bit.

3. INSTRUCTION SET AND CPU

//Lecture 6

3.1) INSTRUCTION CYCLE :

The registers, the ALU, and the interconnecting bus are collectively referred to as the **datapath**.

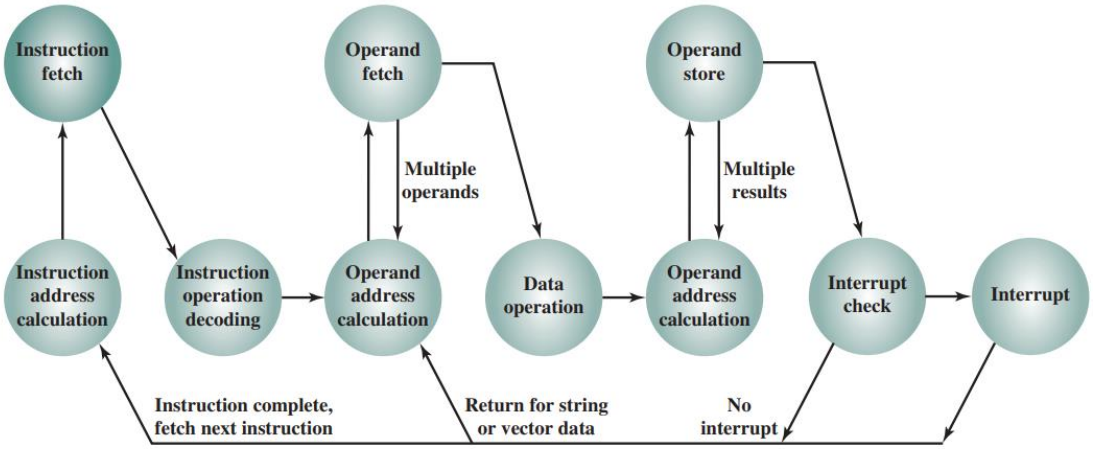


Figure Instruction Cycle State Diagram, with Interrupts

If some interrupts come then CPU first complete execution cycle then move to interrupt cycle stage to service interrupts.

Fetch: Read the next instruction from memory into the processor.

Consider instruction is of 2 words then in fetch cycle we fetch first word and in decoding we fetch remaining part of the instruction from memory.

Execute: Interpret the opcode and perform the indicated operation.

Interrupt: If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

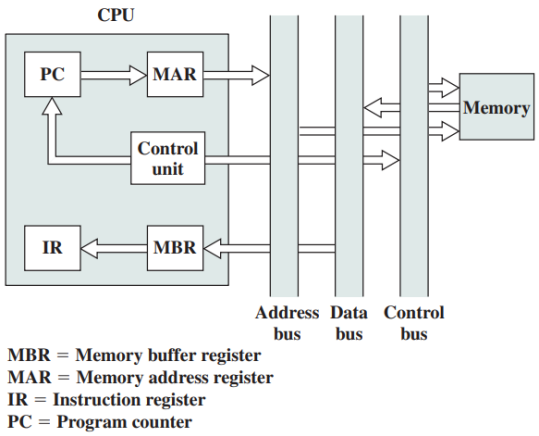


Figure 14.6 Data Flow, Fetch Cycle

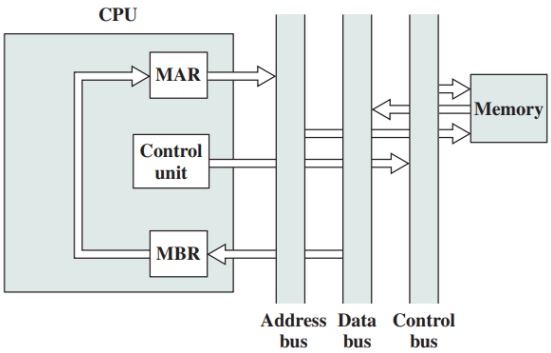


Figure 14.7 Data Flow, Indirect Cycle

INDIRECT CYCLE : The rightmost N bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

INTERRUPT CYCLE : The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine. As a result, the next instruction cycle will begin by fetching the appropriate instruction.

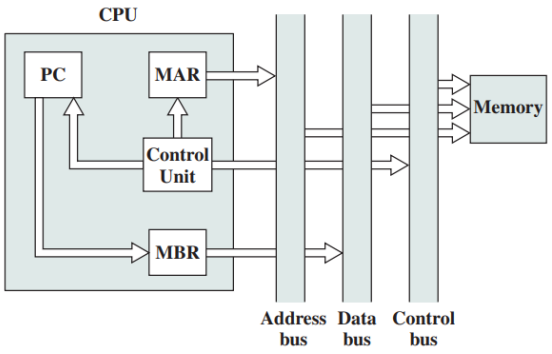
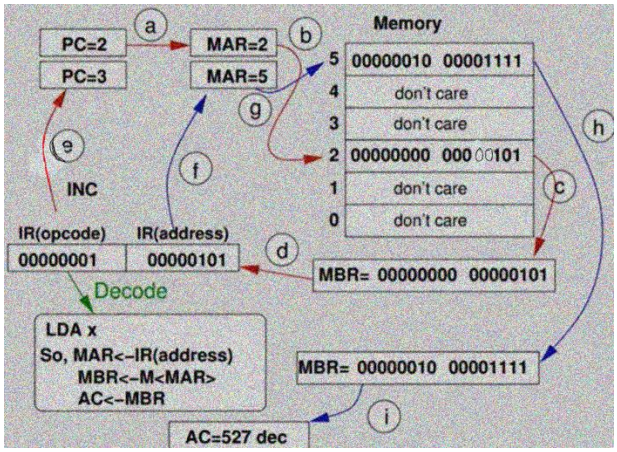


Figure 14.8 Data Flow, Interrupt Cycle

After finishing ISR, RET (return) instruction is executed and return to the point in the interrupted program.

NOTE : A stack pointer register is a small register that stores the address of the last program request in a stack. For example, nested loop uses stack pointer register.



Fetch/Decode
Fetching : abcde
Execution : fghi

Q : A processor checks for interrupts before executing a new instruction. – This is false. There is a difference between fetching an instruction and executing an instruction. Again, see the first diagram.

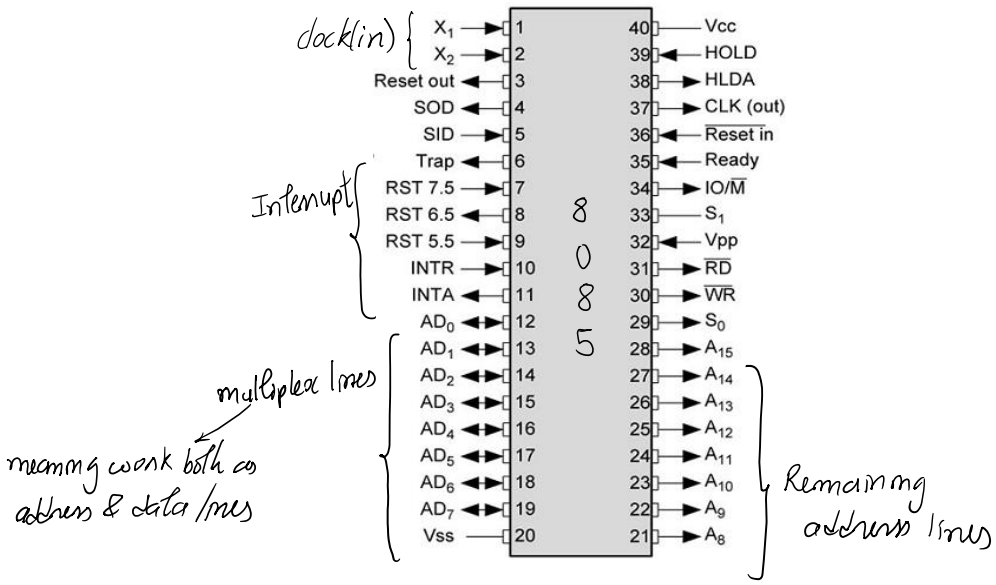
//Lecture 7

3.1.1) INTERRUPTS :

Interrupts are provided primarily as a way to improve processing efficiency.

Common class of interrupts includes : **Interrupts by program** (generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, or reference outside a user's allowed memory space. Then **interrupts by Timer, by I/O, by hardware failure**, etc.

8085 microprocessors : Came from 80's, 8 bit and runs on 5 voltage.



Trap (RST 4.5), RST 5.5, RST 6.5, RST 7.5, INTR all are hardware interrupts.

The processor has a **status register** (**PS**), which contains information about its current state of operation. Let one bit, IE, of this register be assigned for enabling/disabling interrupts. Then, the programmer can set or clear IE to cause the desired action. When **IE = 1**, interrupt requests from I/O devices are accepted and serviced by the processor. When **IE = 0**, the processor simply ignores all interrupt requests from I/O devices.

TYPES OF INTERRUPTS :

1) ***VECTORED INTERRUPT & NON-VECTORED INTERRUPT*** :

The processor uses the vector as a pointer to the appropriate device- service routine. This avoids the need to execute a general interrupt- service routine first. This technique is called a **vectored interrupt**.

In non-vectored interrupt, the interrupting device will simply supply the base address of interrupt-service routine.

All the interrupts which include numbers in their names are vectored interrupts thus, Trap, RST 5.5, RST 6.5, RST 7.5 are vectored interrupts. In 8085 microp. Vector address can be calculated as

$$\textbf{Vector Number} \times \textbf{data bus} = \textbf{vector address}$$

Interrupt service Routine is stored at this location

INTR is exmple of non-vectored interrupt. Here vector address is not predefined.

2) **SOFTWARE & HARDWARE INTERRUPT** :

Hardware interrupts comes from the hardware lines connected to CPU whereas, software interrupts are generated by programs.

In 8085 microprocessors, we have RST0, RST1, …., RST7 these are software interrupts.

3) **MASKABLE & NON-MASKABLE INTERRUPTS** :

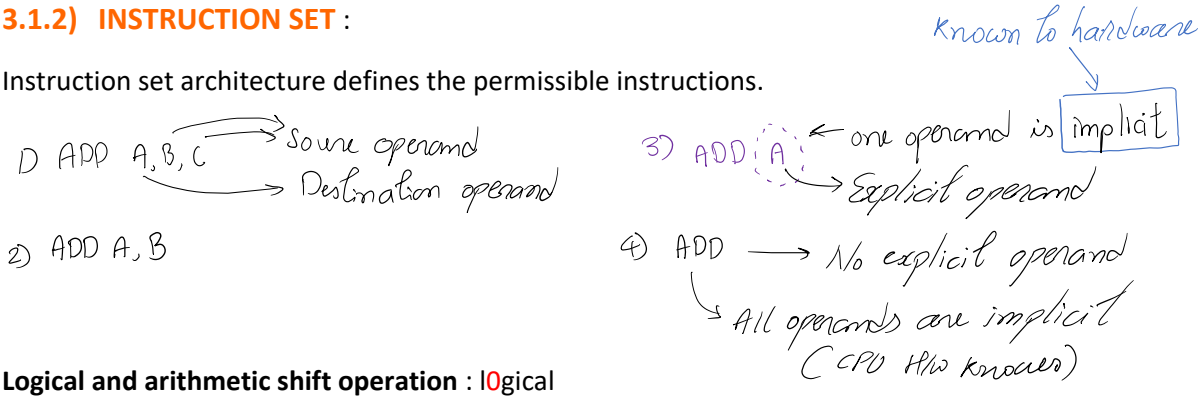
Maskable interrupts can be enabled or disabled. And contrary non-maskable cannot be disabled. TRAP is non-maskable interrupt and rest are maskable in 8085.

NOTE : All these three types are not mutually exclusive types. There are interrupts which are both vectored and maskable.

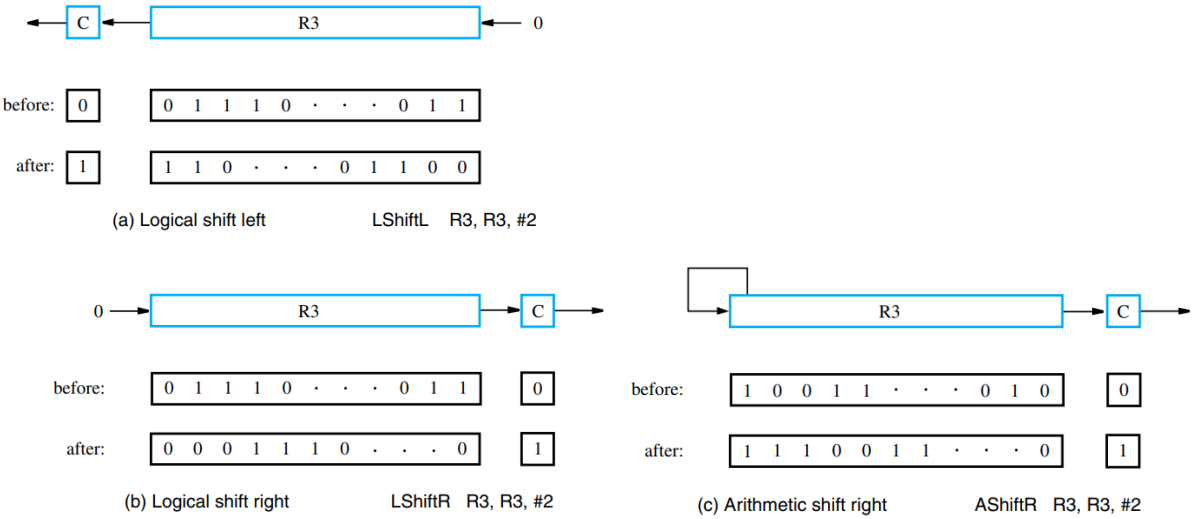
$$\textbf{priority} : TRAP > RST\ 7.5 > RST\ 6.5 > RST\ 5.5 > INTR$$

3.1.2) INSTRUCTION SET :

Instruction set architecture defines the permissible instructions.

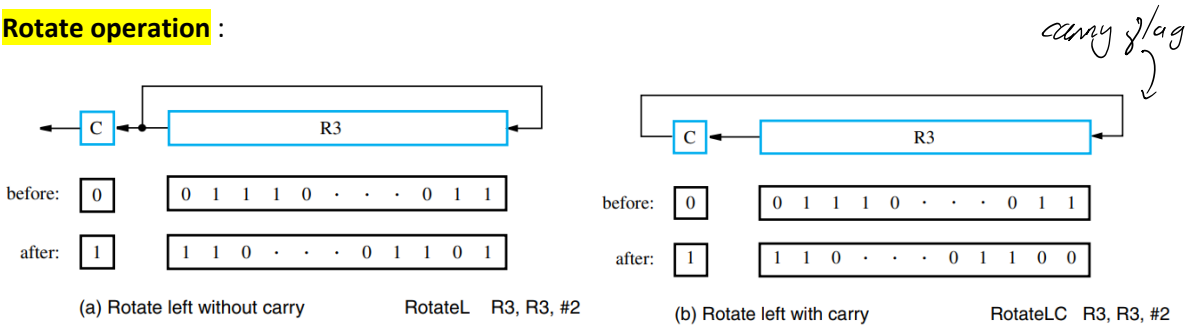


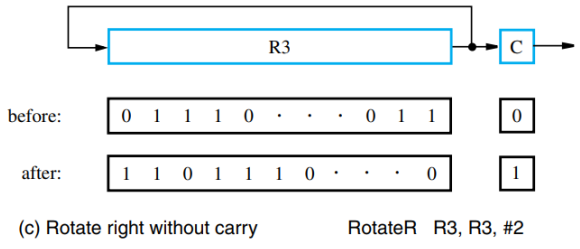
Logical and arithmetic shift operation : l0gical

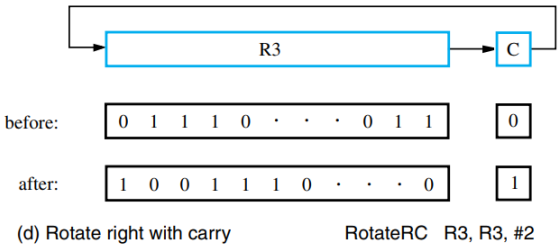


The Arithmetic-shift-left is exactly the same as the Logical-shift-left

Rotate operation :







//Lecture 2B 2023

3.1.3) RISC AND CISC ARCHITECTURE :

In old days, Memory was very expensive… but time wasn't (so, CISC ISA was used) now, it's the opposite. (So, RISC ISA is used)

CISC	RISC
Complex Instruction Set Architecture	Reduced Set Instruction Set Architecture
Focuses on minimizing the size of program	Focuses on minimizing execution time
Memory expensive → Program must be short → number of instructions per program must be less → Memory Memory ISA	Memory cheap → Program must take less time to execute → time per Instruction must be less → Register Register ISA
We have to decrease size of program → Do lots of coding → Leads to complex instruction	Decrease execution time → Less coding (straight forward implimentation) → leads to reduced Instruction
Emphsis on hardware … Hardware have to do all work	Emphsis on Assembler … Assembler have to do most of the work due to lots of Instruction
Assembly programmins life easy → Compilation Easy → Hardware heavy	Assembly program's life hard → Compilation Hard → Software heavy
Load, store happens internally thus we say LOAD, STORE incorporated in instructions	LOAD, STORE are independent instructions
More addressing modes	Less addressing modes
Extensive use of microprogramming	Complexity in compiler
Pipelining is difficult	Pipelining is easy

Now, we know that $\frac{time}{program} = \frac{time}{cycle} \times \frac{cycles}{instruction} \times \frac{instructions}{program}$

CICS will Reduce

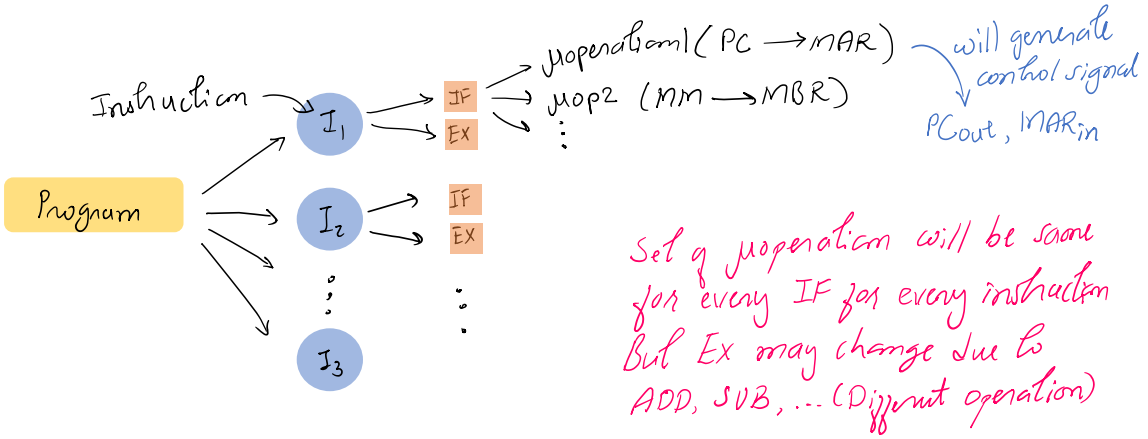
RISC will reduce this

Q : How to decrease execution time ? –

- Increase no. of Cycle per time
- Inrease the size of instruction cache and data cache (which increases spacial locality and reduces miss panalty)

//Lecture 9

3.2) CONTROL UNIT DESIGN :



The **control unit of a processor** performs two tasks :

- **Sequencing** : It causes the processor to step through a series of micro-operation in the proper sequence, based on the program being executed, and
- **Execution** : It generates the control signals that cause each micro-operation to be executed.

You may have noticed that in fetch cycle we first execute $PC + 1 \rightarrow PC$ and then $MBR \rightarrow IR$ because we can perform $MM \rightarrow MBR$ and $PC + 1 \rightarrow PC$ simultaneously as one is accessing memory to register and other incrementing it's value so both are independent operation thus, can be performed in same cycle. As a result, we need only 3 CPU cycles to fetch the instruction from memory.

	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

There are two main control unit design,

- *Hardwired implementation (used by RISC)*
- *Microprogrammed implementation (used by CISC)*

3.2.1) HARDWIRED IMPLEMENTATION : RISK

Here control signals are directly implemented on the hardware circuit. Thus, it is fast. Control signals are represented in the form of SOP expression XY + YZ + Y'X

Disadvantage : not flexible

Q : Consider a hypothetical processor which supports 3 instruction (I1, I2, I3) and all uses 4-control signals (S0, S1, S2, S3), each instruction requires 4 microp. (t1, t2, t3, t4)

	I_1	I_2	I_3
T_1	S1, S3	S0, S3	S2, S4
T_2	S0, S1	S0, S2	S0, S1
I_3	S3, S1	S1, S3	S1, S3
T_4	S0, S1	S0, S2	S0, S3

$S_0 = T_1 I_2 + T_2(I_2 + I_3) + T_4$

$S_3 = T_1(I_1 + I_2) + T_3 + T_4 I_3$

↖ control expression of S_3

Register Window Size:

G: No. of global registers.

L: No. of local registers in each window.

C: No. of registers common to two windows

W: No. of windows.

- **Window size (S):** No. of registers available for each window.

Window Size= S= L+2C+G

- **Register file (F):** Total no. of registers needed in the processor.

Register File= F= (L+C)W+G

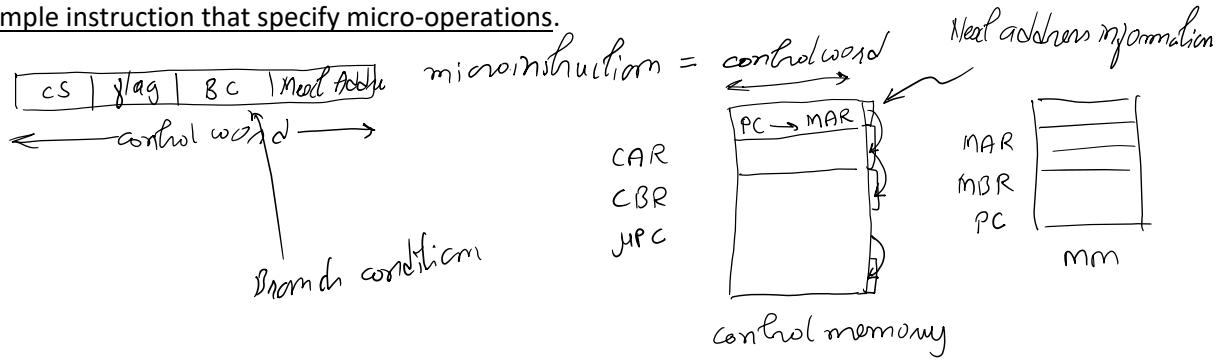
//Lecture 10

3.2.2) MICROPROGRAMMED CONTROL UNIT : CISC

Idea : Instead of generating control word of any specific micro-operation using hardware (like flip flop, gates, etc) every single time we can store them beforehand in specific memory called **control memory**.

An alternative to a hardwired control unit is a **microprogramed control unit**, in which the logic of the control unit is specified by a **microprogram**.

A microprogram consists of a sequence of instructions in a microprogramming language. There is very simple instruction that specify micro-operations.



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

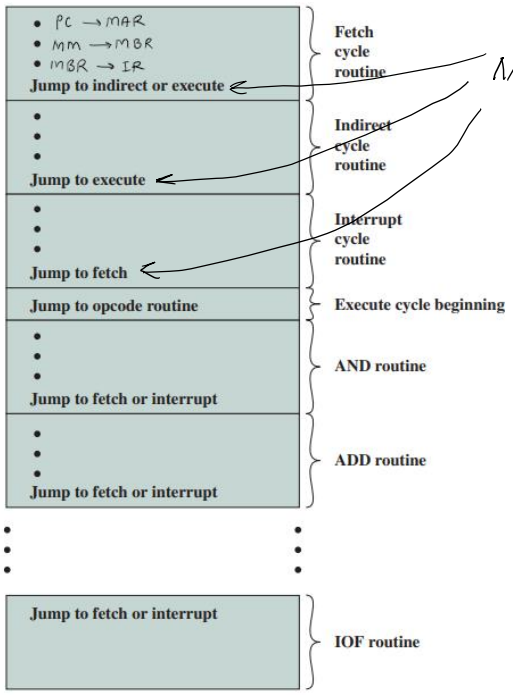


Figure Organization of Control Memory

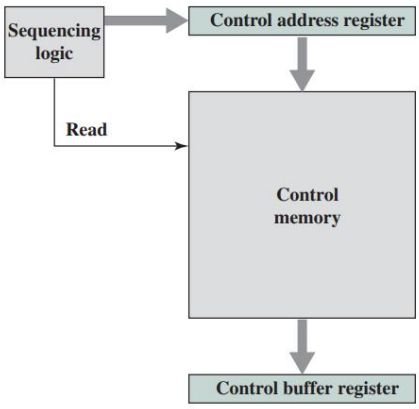
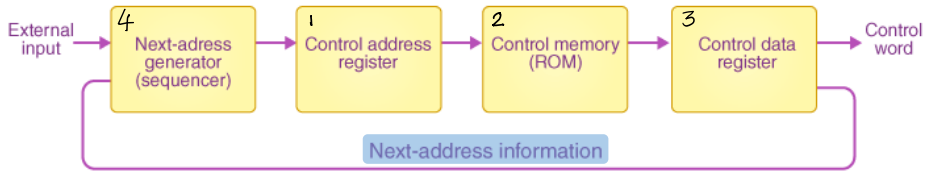


Figure Control Unit Microarchitecture

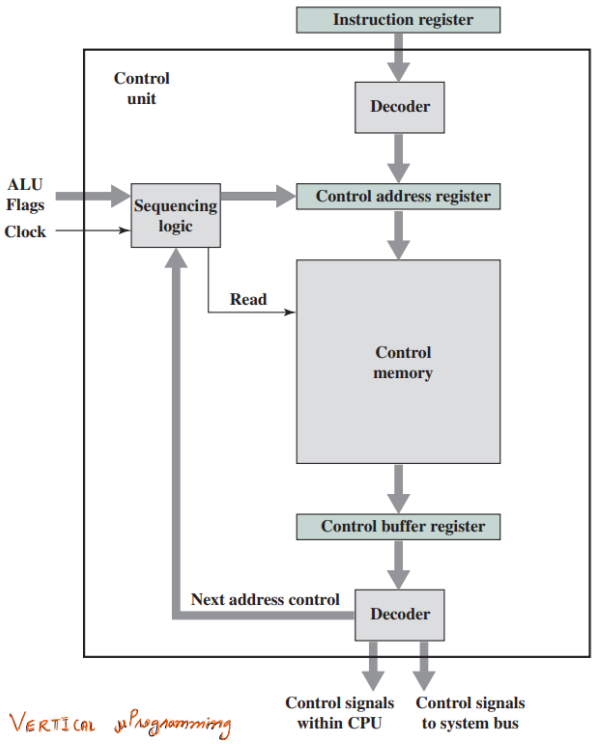
CONTROL MEMORY :

- The set of all possible microinstructions is stored in the control memory.
- The *control address register* contains the address of the next microinstruction to be read.
- When microinstruction is read from the control memory, it is transferred to a *control buffer register.*
- *Control buffer register (CBR)* is connected with *control lines* (going out of control unit)
- Thus, *reading a microinstruction* from the control memory is the same as *executing that microinstruction.*

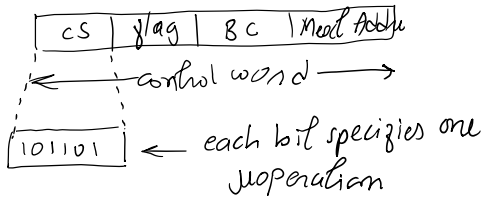
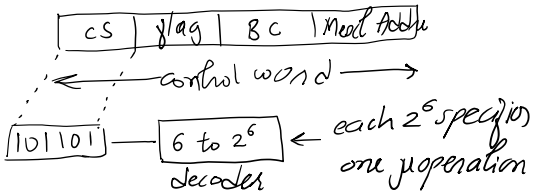


WORKING : All this happen during one clock pulse.

- To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
- The word whose address is specified in the CAR is read into the CBR
- The content of the CBR generates control signals and next address information for the sequencing logic unit
- The sequencing logic unit loads a new address into the CAR based on the next-address information from the CBR and the ALU flags.



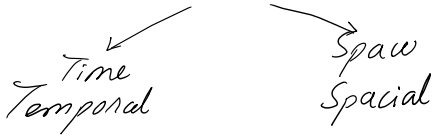
In horizontal μpCU, We will be having signal (control signals) in decoded form. Whereas in vertical μpCU, we will have signals in encoded form so.

Horizontal Microprogramming (decoded binary formate)	Vertical Microprogramming (encoded binary formate)
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 (due to decoder)
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	Optimize programming
More flexible	Less flexible
$CS = \#bits\ in\ CS$ $Flag = log_2(\#of\ flags)$ $BC = \log_2(\#\ of\ conditions)$ $Next\ Add. = \log_2(\#\ of\ Control\ Word)$	$CS = \log_2(\#bits\ in\ CS)$ $Flag = log_2(\#of\ flags)$ $BC = \log_2(\#\ of\ conditions)$ $Next\ Add. = \log_2(\#\ of\ Control\ Word)$

4. CACHE MEMORY DESIGN

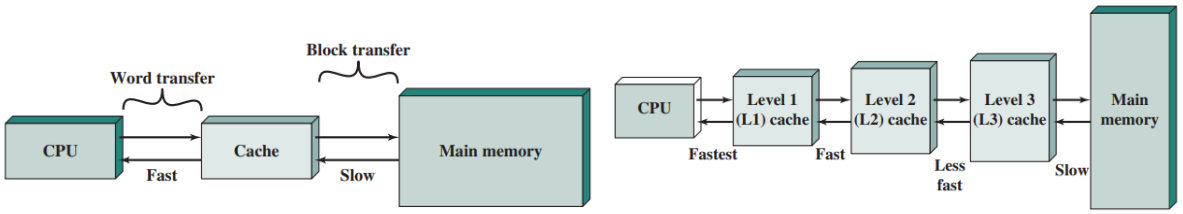
//Lecture 11

Cache memory is based on concept of locality of reference.



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.



(a) Single cache (b) Three-level cache organization

Cache block : Refers to a set of continuous address locations of some size. It is also called **cache line**.

Cache hits and miss :

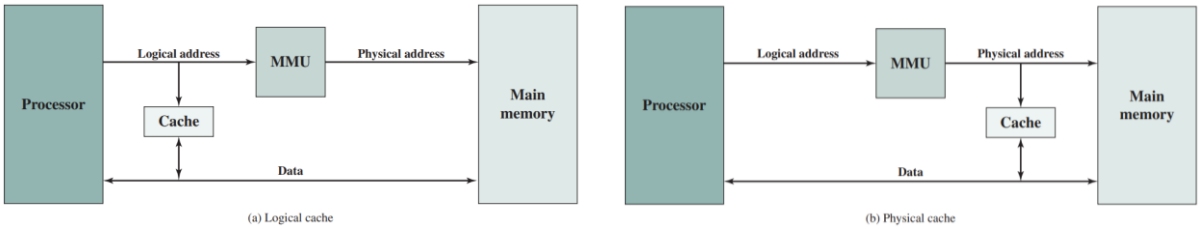
The Cache control circuitry determines whether the requested word currently exists in the cache.

- If it does, the Read or Write operation is performed on the appropriate cache location. A *read or write hit* is said to be occurred.
- If it doesn't, the block of words containing the requested word to be copied from MM into the cache. This constitutes a *read miss*.

For a Write operation, the system can proceed in one of two ways. In the first technique, called the **write-through protocol**, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the **dirty** or **modified bit**. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room for a new block. This technique is known as the **write-back**, or **copy-back**, protocol.

//lecture 12

4.1) CACHE MAPPING TECHNIQUES :



(a) Logical cache (b) Physical cache

A **logical cache**, also known as a virtual cache, stores data using virtual addresses. The processor accesses the cache directly, without going through the MMU.

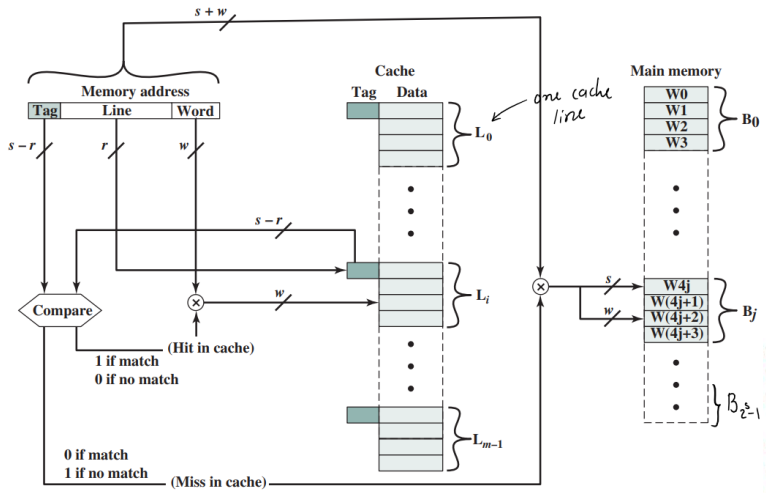
A **physical cache** stores data using main memory physical addresses. Thus, Logical cache is fast but we use combination of both.

Tag tells us total how many blocks of main memory gets mapped to each cache line. Suppose, RAM has 32 blocks and cache has 4 blocks thus, 8 blocks maps to each cache line.

4.1.1) DIRECT MAPPING :

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as i = j modulo m

where i = cache line number, j = main memory block number, m = number of lines in the cache



- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = $m = 2^r$
- Size of cache = 2^{r+w} words or bytes
- Size of tag = $(s - r)$ bits

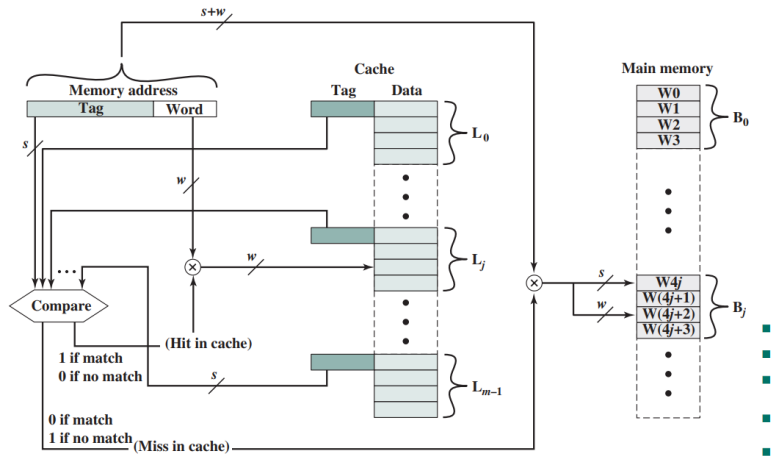
Figure Direct-Mapping Cache Organization

Tag directory size = #of tag **bits** × #of lines in cache, **#of comparator** : 1, **#of mux** = 1

Advantage : Simple and inexpensive

Disadvantage : There is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as **thrashing**).

4.1.2) FULLY ASSOCIATIVE MAPPING :



- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

Figure Fully Associative Cache Organization

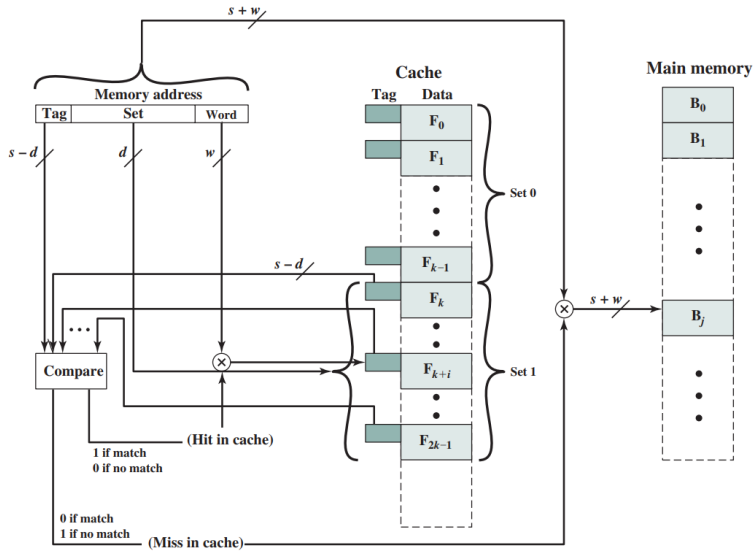
Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache.

#of comparator = #of lines in cache

#of mux = 2 (1^{st} mux … data line selector -> #of Cache line to 1 **and** 2^{nd} mux … offset/data selector-> #of words to 1)

//Lecture 13

4.1.3) SET ASSOCIATIVE MAPPING :



- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $m = kv = k \times 2^d$
- Size of cache = $k \times 2^{d+w}$ words or bytes
- Size of tag = $(s - d)$ bits

Figure . k-Way Set-Associative Cache Organization

In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are m = v * k

i = j modulo v

where i = cache set number, j = main memory block number, m = number of lines in the cache, v = number of sets, k = number of lines in each set. This is referred to as **k-way set-associative mapping**.

Mapping	Address format	#of comparator	Size of comparator	#of mux.
Direct	Tag, index, offset	1	Tag bits	1
Set associative	Tag, set, offset	K (k-way)	Tag bits	2 (k to 1, blocksize to 1)
Fully associative	**Tag, offset**	#of blocks	Tag bits	2 (#of blocks to 1, blocksize to 1)

Q : Direct-mapped cache can never have a lower miss rate than an associative cache of the same size (number of blocks) ? – this is false. consider 4-line direct cache and fully set associative cache. Consider block request 0, 1, 2, 3, 4, 3, 4, 3, 4, …. With most recently used scheme. Note that in case of direct mapping does not support any page replacement algo.

⋆ **CACHE MISSES** :

Miss Type	Description	Hotel Analogy
Compulsory or Cold	The first reference to a block of memory, starting with an empty cache.	The hotel is empty and the first guest has not yet arrived.
Capacity	The cache is not big enough to hold every block you want to use.	The hotel has no vacancies.
Conflict	Two blocks are mapped to the same location and there is not enough room to hold both.	A particular floor of the hotel which a guest **has** to stay on has all rooms occupied.

+ means there's an improvement in the cache miss rate, 0 means no change and - means the situation gets worse.

	Cache Parameter		
Miss Type	Cache Size	Block Size	Associativity
Compulsory	0/-	+	0
Capacity	+	0	0
Conflict	0	0/-	+

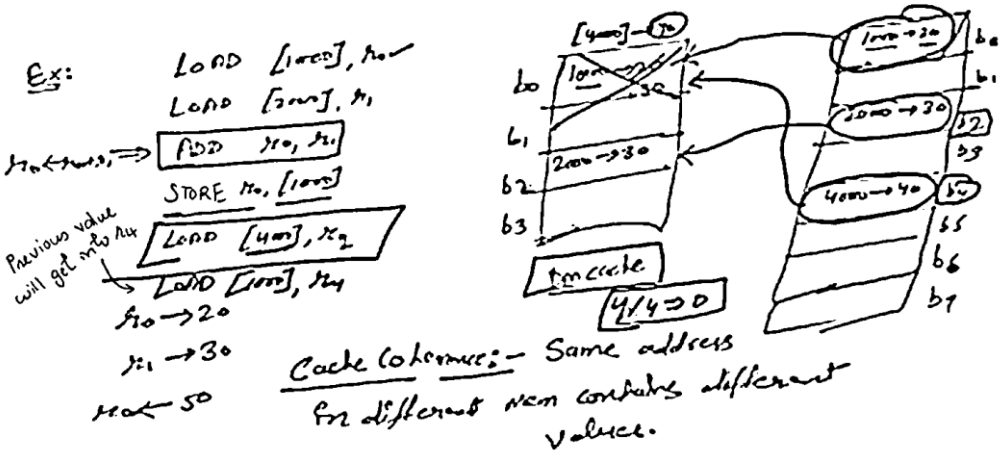
NOTE : Cache entry = no. set and Cache line = no. of cache block.

Outstanding cache miss : When a cache miss happens, it has to fetch the data from the next level cache/memory, until this data arrives, you need to hold this request somewhere. This is called outstanding cache miss meaning request received but not acknowledged.

//Lecture 15

4.2) CACHE UPDATING TECHNIQUES :

Problem with write operation : **CACHE COHERENCE** – same address in different memory contains different values.



Solution : cache updating technique (write through, write back)

4.2.1) TECHNIQUES :

//Lecture 08 Cache PYQ

WRITING TO CACHE : Multiple copies of data exists, must be kept in sync.

Write-hit policy : write through, write back (needs dirty bit)

Write miss policy : Write allocate, No write allocate

Typical Caches : Write back + write allocate, usually

Write through + no write allocate, occasionally

1) **WRITE THROUGH** : The information is written to both the block in the cache and to the block in the lower-level memory. (update simultaneously on modification)
2) **WRITE BACK** : The information is written only to the block is the cache. The modified cache block is written to main memory only when it is replaced. (update MM on replace)

To reduce the frequency of writing back blocks on replacement, a feature called the **dirty/modify bit** is commonly used. When dirty bit

- Set to **1** then block is modified (need to update MM on replacement)
- Reset to **0** then block is not modified (nothing! Enjoy)

In case of miss, we have following policy

Write allocate : We bring data from MM to cache and perform write to cache.

No write allocate : We directly write to MM

- ***Write though policy*** : Here by default we take no – write allocate policy.

$$T_{avgr} = H_c T_c + (1 - H_c)(T_m + T_c)$$ ← For Hierarchical

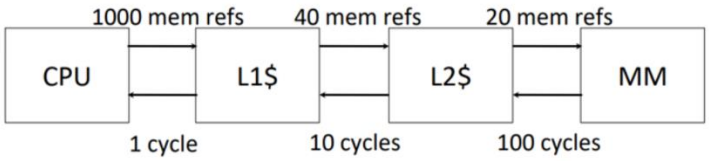
$$T_{avgw} = H_c(\max(T_m, T_c)) + (1 - H_c)(\max(T_m, T_c))$$

$$T_{avgr} = H_c T_c + (1 - H_c)T_m;\ T_{avgw} = T_m$$ ← for parallel

- ***Write back policy*** : Here by default we take write allocate policy

$$T_{read} = T_{write} = H_c T_c + (1 - H_c)[\%\ of\ dirty\ bits \times (T_m + T_m + T_c) + \%\ of\ No\ dirty\ bit \times (T_m + T_c)]$$

Memory latency : The time between initiating a request for data and the beginning of the actual data transfer.



Local miss rate = $\frac{Number\ of\ cache\ misses}{Total\ Access\ to\ this\ cache}$

Global miss rate = $\frac{number\ of\ misses\ in\ cache}{Total\ number\ of\ memory\ access\ from\ the\ processor}$

Here in this case, Global miss rate of L1 = local miss rate of L1 = $\frac{40}{1000}$

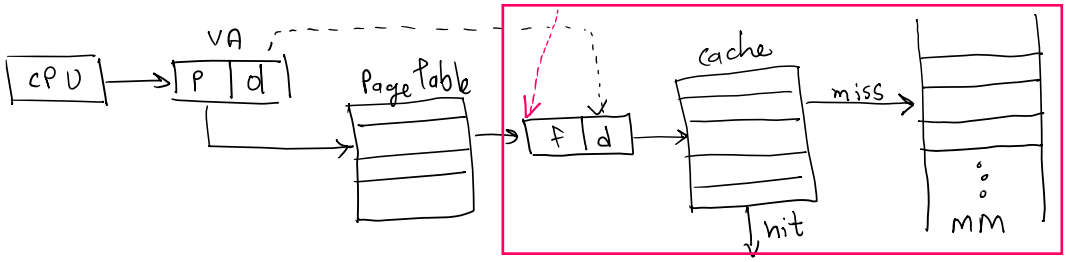
Local miss rate of L2 = $\frac{20}{40}$, Global miss rate of L2 = $\frac{20}{1000}$

NOTE : EMAT = 1 + L1miss x (1 + L2miss) here we use local misses and (1 + L1'miss + L2'miss) here we use global miss (it is just shortform so always use local miss) There is difference between access time, hit time, miss time, miss penalty. So be aware.

//Lecture 16

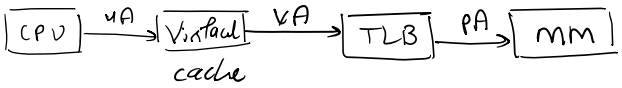
4.2.2) VIRTUAL AND PHYSICAL CACHE :

Till now we have seen cache mapping, access from this as starting point



Now, we will see how effectively access or map cache from virtual memory.

Alternate approach : place the cache before the TLB or page table.



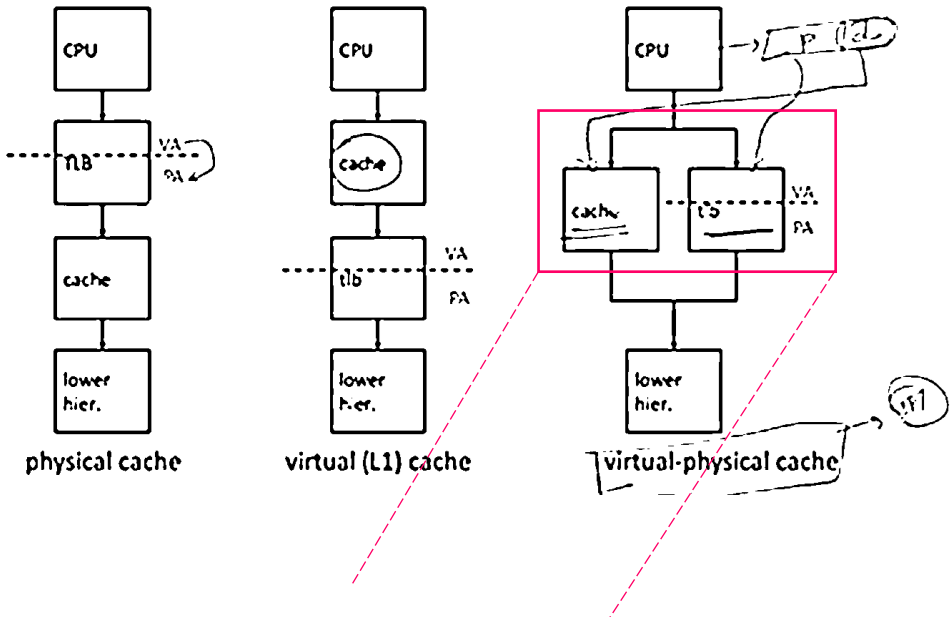
Advantage of this approach : one step process in case of a hit

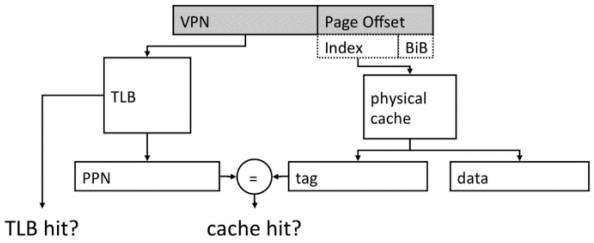
Disadvantage : Cache needs to be flushed on a context switch (because new process has it's own virtual mapping) unless address space identifiers (ASIDs) included in tags

- Synonym : two virtual addresses map to one physical address
- Homonym : two physical addresses map to one virtual address

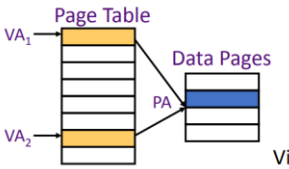
Both synonym and homonyms are called **aliasing**.

VIRTUAL – PHYSICAL CACHE :

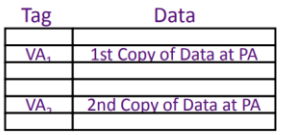




Synonyms :

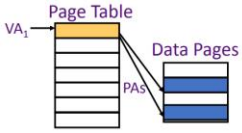


Two virtual pages share one physical page



Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

Homonyms :

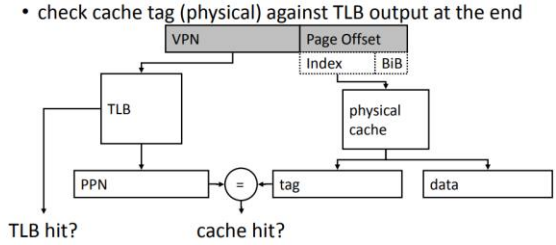


One virtual page maps to two physical pages

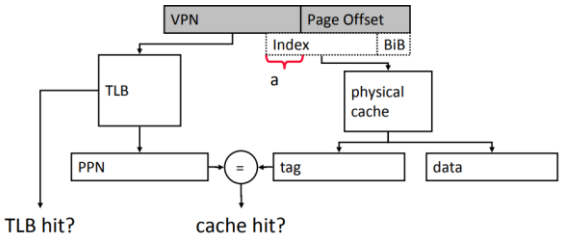
Tag may not uniquely identify cache data
Solution: Add ASID with tag
Or
Physical tags
Or
Flush on context switch

VIRTUALLY INDEXED PHYSICALLY TAGGED :

- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



- If C>(page_size × associativity), the cache index bits include VPN ⇒ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?



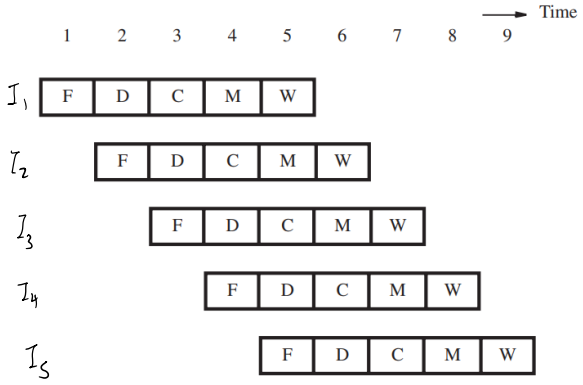
Solution to synonym problem in VIPT cache : coloring, we use extra "a" bits for coloring purpose.

5. PIPELINING

//Lecture 17

5.1) INTRODUCTION TO PIPELINE :

In case of pipeline execution,



# instruction	pipeline cycle	CPI
1	5	5
2	6	3
3	7	2.33
4	8	2
⋮	⋮	
100	104	1.04

Thus ideally CPI = 1

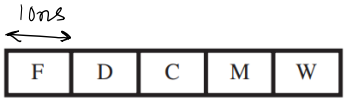
But in case of non-pipeline execution, each instruction takes cycle = # of stages.

$$\textbf{Speedup} = \frac{n \times T_n}{(k+n-1)T_p} = \cdots \textbf{under ideal condition} = \frac{T_n}{T_p} = k$$

No. of stages

max achievable speedup

Also,



In non-pipeline, Execution Time = 50ns and in pipeline, as ideally CPI is 1 thus, Execution Time = 10ns.

Thus, theoretically ➔ $T_n = k \times T_p$

Thus, efficiency is $\eta = \frac{Speedup}{k}$ ← your performance / ← your max performance

5.1.1) TYPES OF PIPELINING :

UNIFORM DELAY PIPELINE : $T_p = t_p + buffer\ delay$

NON-UNIFORM DELAY PIPELINE : $T_p = \max(stage\ delay) + buffer\ delay$

5.1.2) RICS PIPELINE :

The basic operation of a RISC instruction set includes :

- ALU instructions (like ADD, SUB, MUL, …)
- Load and store instructions
- Branches and jumps

Every instruction in RICS subset can be implemented in at most 5 clock cycles.

5.1.3) ISSUES WITH PIPELINING :

Pipelining increases the CPU instruction throughput – the number of instructions completed per unit of time – but it does not reduce the execution time of an individual instruction.

- In fact, it usually increases the execution time of each instruction due to overhead in the control of the pipeline but if we have only instruction then both non-pipeline and pipeline will take same time as long as stages are balanced.

Imbalance among the pipe stages reduce performance since the clock can run no faster than the time needed for the slowest pipeline stage.

//Lecture 2

5.2) PIPELINE HAZARDS :

Hazards prevent the next instruction in the instruction stream from executing during it is designed clock cycle. Hazards in pipeline can make it necessary to **stall** the pipeline. There are three classes of hazards :

Structural Hazards : arise from resource conflict when the hardware cannot support all possible combination of instructions simultaneously in overlapped execution.

Data Hazards : Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

Control hazards : arise from the pipelining of branches and other instructions that changes the PC.

5.2.1) PERFORMANCE OF PIPELINES WITH STALLS :

Speedup from pipelining : $\frac{Average\ instruction\ time\ unpipelined}{Average\ instruction\ time\ pipelined} = \frac{CPI\ unpipelined}{CPI\ pipelined} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$

The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute pipelined CPI :

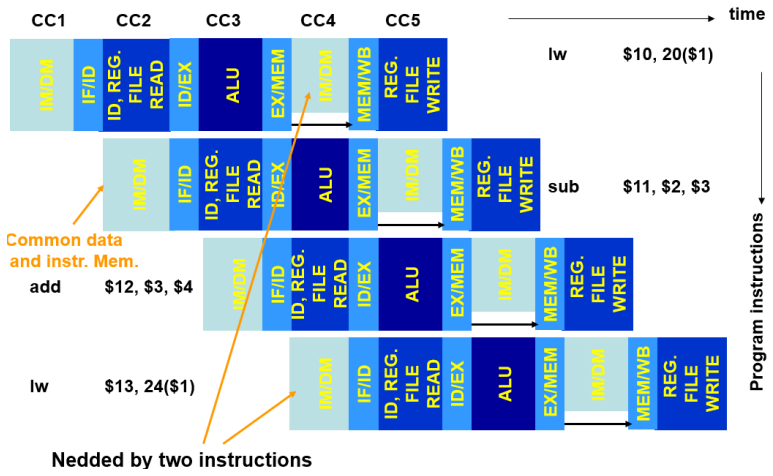
CPI pipelined = ideal CPI + pipeline stall clock cycles per instruction

= 1 + pipeline stall clock cycles per instruction.

$$\text{Speedup} = \frac{CPI\ unpipelined}{1+pipeline\ stall}$$

5.2.2) HANDING HAZARDS :

STRUCTURAL HAZARDS :



Consider a computer with common data and instruction memory. The fourth cycle of MEM instruction requires memory (memory read) and at the same time the first cycle of the fourth instruction requires instruction fetch (memory read). This will cause a memory resource conflict.

Solution : introduce stall, two separate lines one for IM and one for DM.

DATA HAZARDS : dependencies that causes stall

There are two types of data dependence – **True data dependence** (RAW) and **name dependences** (WAR – anti dependence, WAW)

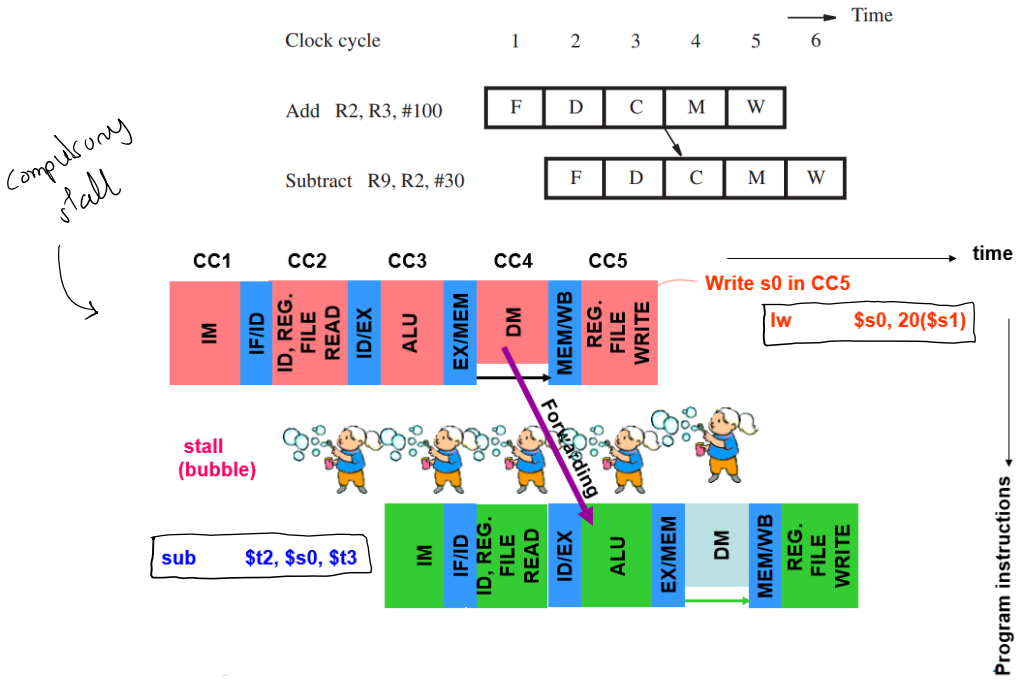
- **Name dependence** : occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instruction associated with that name.

Anti-dependence : WAR

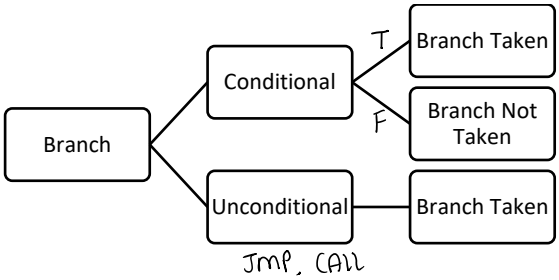
Output dependence : WAW

Solution for name dependence : **Register Renaming**. (hardware or compiler will do)

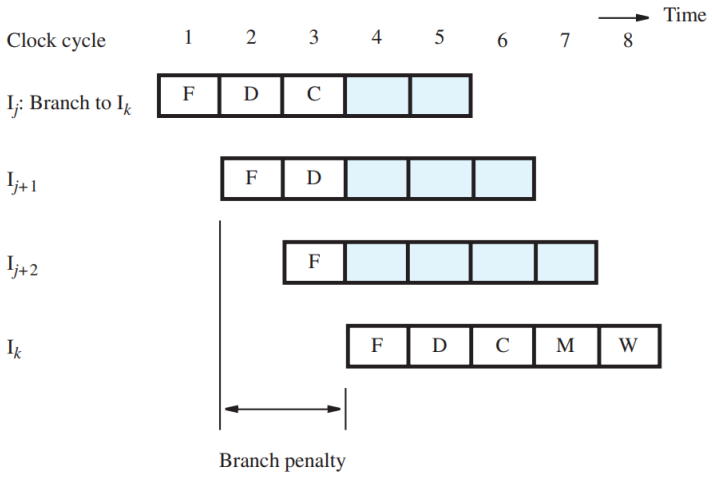
Solution for true data dependence : Operand/data forwarding/ bypassing



5.2.3) HANDLING CONTROL/BRANCH HAZARDS :



UNCONDITIONAL BRANCH :

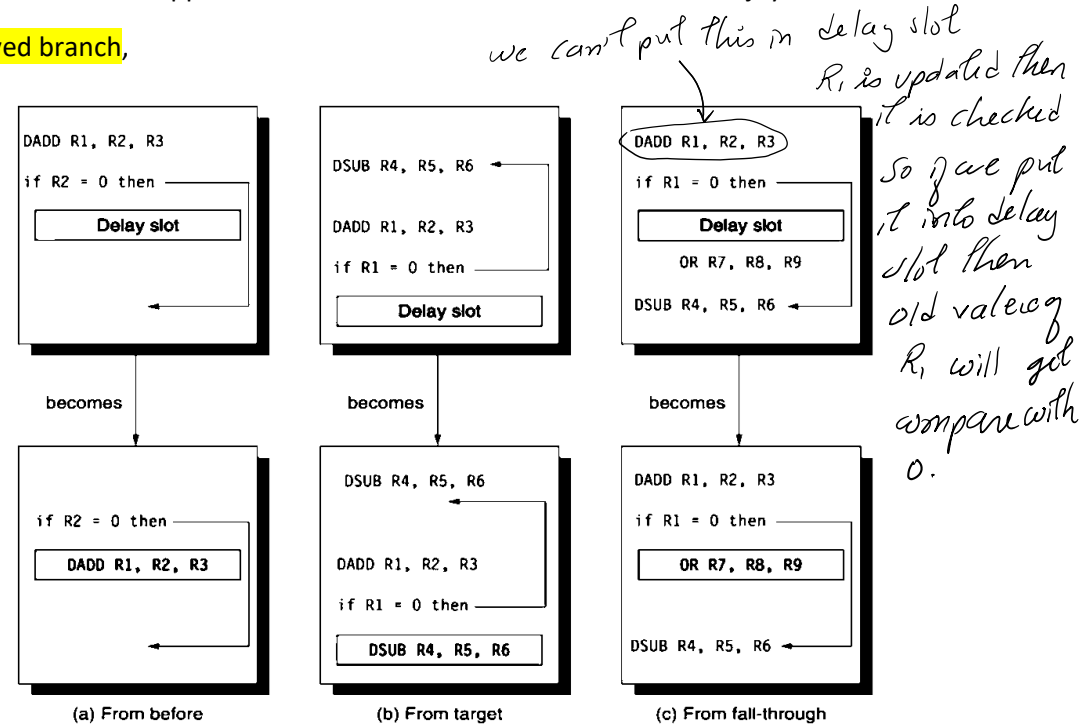


Reducing pipeline branch penalties (both conditional and unconditional) :

1) **Freeze/Flush** : The simplest approach, deleting any instructions after the branch until the branch destination is known.
2) **Branch prediction** : A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed.
3) **Delayed branch** : In a branch penalty slot, we can schedule other instruction which are independent of current execution.

In the simple five-stage pipeline, new *predicted-not-taken* or *predicted-untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. Just like procrastinator whatever happens we will see in future for now on let'em enjoy.

Back to delayed branch,



PERFORMANCE OF BRANCH SCHEMES :

Pipeline speedup = $\frac{pipeline\ depth}{1+pipeline\ stall\ cycles\ from\ branches}$

Pipeline stall cycles from branches = branch frequency × branch penalty

Thus, **Pipeline speedup** = $\frac{\boldsymbol{pipeline\ depth}\ (Total\ stages)}{\boldsymbol{1+(branch\ frequency \times branch\ penalty)}}$

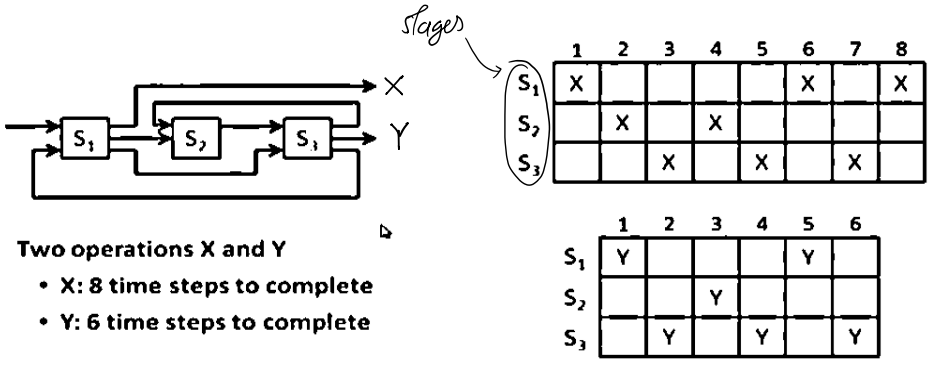
NOTE :

- **Dynamic branch prediction cannot eliminate all branch penalties.**
- **Million instruction per second can be found by execution time = $Total\ instruction \times CPI \times clock\ time$. Now, this will give us execution time for total instruction, from this we find no. of instruction executed in 1 sec then represent it in terms of 10^6.**
- **Conditional branch does not require to save PC contents.**
- **M[3+r5] ← r3, Here 2 true hazards can come if r5 and r3 is not available.**

//https://www.youtube.com/watch?v=Z9gvdQU6IAE

5.3) SCHEDULING OF NON-LINEAR PIPELINES :

Till now we have seen only linear pipeline in which we never return back to previous stage for something.



In first table, X is going through s1 – s2 – s3 – s2 – s3 – s1 – s3 – s1

We need to take where X and Y operation do not access same stage simultaneously.

5.3.1) LATENCY ANALYSIS :

The number of time units between two initiations of a pipeline is called the **latency** between them.

Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a **collision**.

The latencies that can cause collision are called **forbidden latencies**.

In table *forbidden latencies = Distance between two X's in the same row of the reservation table*.

In X's table, forbidden latencies = {2, 4, 5, 7}

Latency sequence : A sequence of permissible non-forbidden latencies between successive task initiations.

Latency cycle : A latency sequence that repeats the same subsequence.

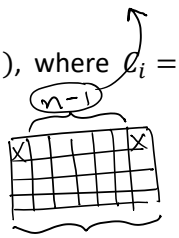
Thus, our

Main objective : Obtain the shortest average latency between initiations without causing collisions.

STEP 1 : We define a **collision vector**

otherwise it won't be shortest

- If the reservation table has n columns, the maximum forbidden latency is $m \leq n-1$.
- The permissible latencies p will satisfy : $1 \leq p \leq (m-1)$
- The collision vector is an m-bit binary vector $C = (C_m, C_{m-1}, \ldots, C_2, C_1)$, where $C_i = 1$ if latency i causes collision, and $C_i = 0$ otherwise.
- C_m is always 1.

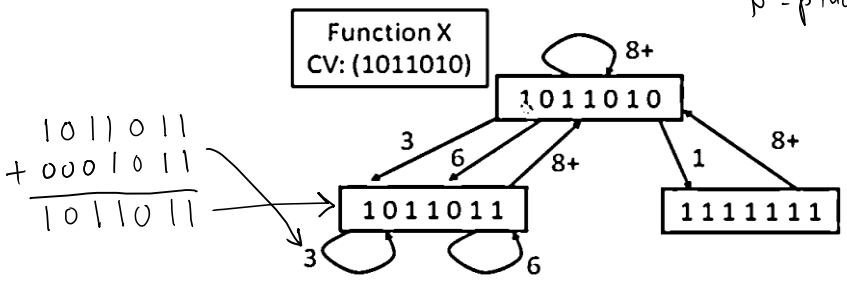


STEP 2 : From the collision vector (CV), we can construct a state diagram specifying the permissible state transitions among successive initiations.

- The collision vector corresponds to the initial stage of the pipeline
- The next state at time t+p is obtained by shifting the present state p-bit to the right and OR-ing with the initial collision vector C.

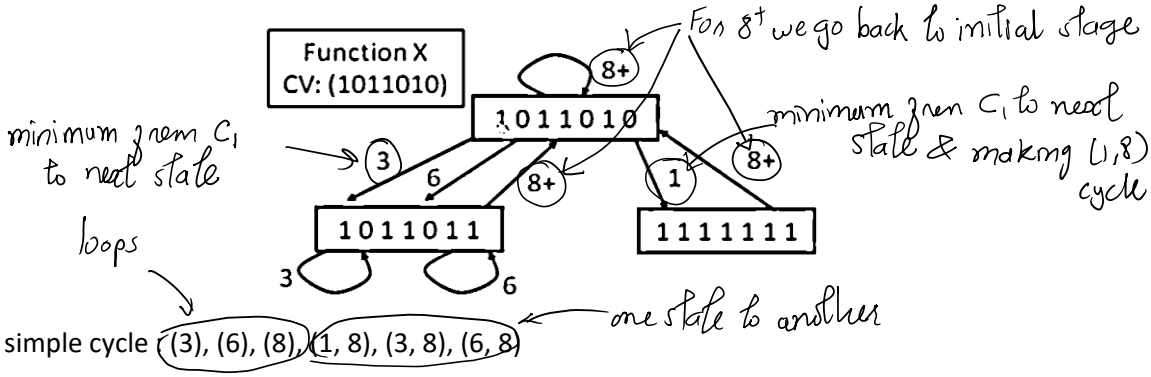
p = place where we have 0 bit in collision vector (here 1, 3, 6, 8)

we take by default



STEP 3 : From the state diagram, we can determine latency cycles that result in minimum average latency (MAL).

- In a simple cycle, a state appears only once.
- Greedy cycle, outgoing edges with minimum latencies. This should be from initial stage only.



Here simple cycle : (3), (6), (8), (1, 8), (3, 8), (6, 8)

Greedy cycles : (3), (1, 8)

MAL = min(greedy cycles)

NOTE : in greedy cycles if we have (1, 8) then we take average and consider value for example, here (1+8)/2 = 4.5 but as 3 is min, MAL = 3 for X.

Optimizing non-linear pipelining :

- Introducing dummy stage between two stages to shift X from one cell to another but operation remains same.