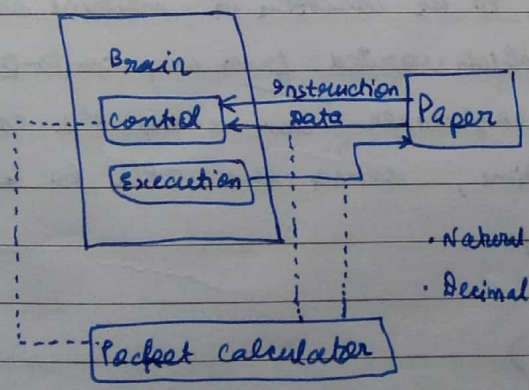
COMPUTER ARCHITECTURE & ORGANIZATION - I

Computer Architecture - interface b/w software and hardware
- Abstract model & programmer's view
  in terms of → instructions
  → addressing modes
  → registers
- describes what computer does
- deals with high level design issues
  ↳ for eg. "is there a multiplication instruction?"

Computer Organisation - deals with component of connection in a system
- expresses the realization of architecture.
- describes how computer does a task
- organisation is done on basis of architecture
- deals with low level design issues
  ↳ eg. "is there a multiplication unit or is it done by repeated addition?"

Human computer

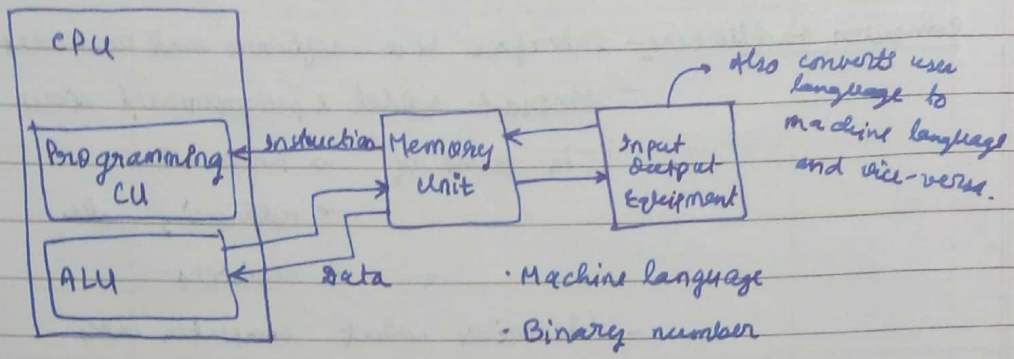


Limitation:
1) speed
2) inaccuracy depending on complexity of instruction.

- control interprets the instruction and ensures that they are executed in proper sequence.

- Paper contains instructions, data or operand and intermediate results.

Machine computer



History of computer

1st generation

- 1G (late 1940's to early 1950's)
  - control of computer centralized to single CPU.
  - all operations in the system, like transfer of a single word of information b/w I/O device and memory, required direct intervention by CPU

~~MSc~~

- very little system software compared to modern machines.
- programs had to be written in machine or assembly languages which varied from computer to computer.
- only one programmer can access the computer at any time, causes the CPU to be idle frequently.

2nd generation

- 2G (1955 - 1964)

- Transistors which had been invented in 1948 at AT&T Bell laboratories gradually replaced vacuum tubes in the design of switching circuit.
- Cathode ray tube memories and delay line memories were replaced by ferrite core and magnetic drum as the technologies in main memory.
- Use of floating point arithmetic hardware.

- Machine independent high level programming languages such as FORTRAN, COBOL were introduced to simplify programming.

- Special processor (for I/O) was introduced to supervise I/O operations.

- Computer manufacturers began to design system software.

• 3G (1964 - 1971)

- Integrated circuits began to replace the discrete transistor circuits used in 2G, resulting in substantial reduction of cost & space.

- Semiconductor memory replace ferrite core.
- Use micro programming to simplify the design of CPU.
- Pipelining and multi-processing were introduced for concurrent or parallel processing to increase the effective speed of execution.
- efficient method for automatic sharing of facilities / resources of the system.
- User interacts through keyboards and monitors interfaced with an operating system.

• 4G (1971 - Present)

- Uses VLSI technology
- Compact, reliable, cheap
- gave rise to PC revolutions
- time-sharing, real time networks, distributed OS introduced
- High level languages like C, C++, etc., were used.
- Development of GUI, mouse, handheld devices
- Great developments in the field of networks.
- Concept of internet was introduced
- More powerful computer could be linked together to form networks

· 5G (in development)

- multiprocessor based system
- use of AI
- use of optical fibre circuits.
- Magnetic enabled chip
- Huge development of storage.

Instruction

Binary string

0011 1011 0000 0000 0111

← 20 bits →

= size of instruction in 1G

Add content of memory location with accumulator (CPU register)

Symbolic programming

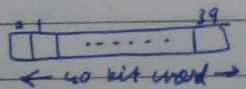
ADD X1

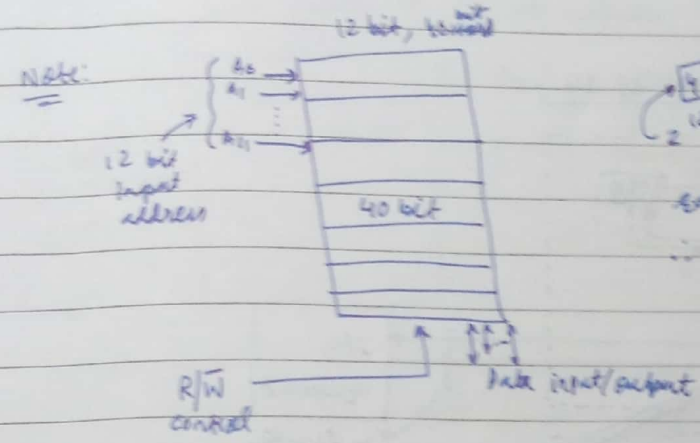
↳ Add content of memory location X1 with accumulator.

1G

· 40 bit word ⇒ it is possible to access 40 bit data at a time from the memory.

· 12 bit address





Note:

4096 = 2^{12} locations possible in memory

each has 40 bit

∴ total = $2^{12} \times 40$ bit memory.

W ⇒ 0
R ⇒ 1 (Read) Signal

40 bit → can be either → data
or → instruction

Consider for the moment → data

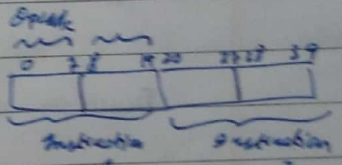
Fixed point number:

Put a decimal anywhere

Floating point number:

Binary position like position 0 & 1 of the 40 bits available.

Now, consider instruction



40 = (0 - 19) + (20 - 39)
↓ Instruction 1 ↓ Instruction 2

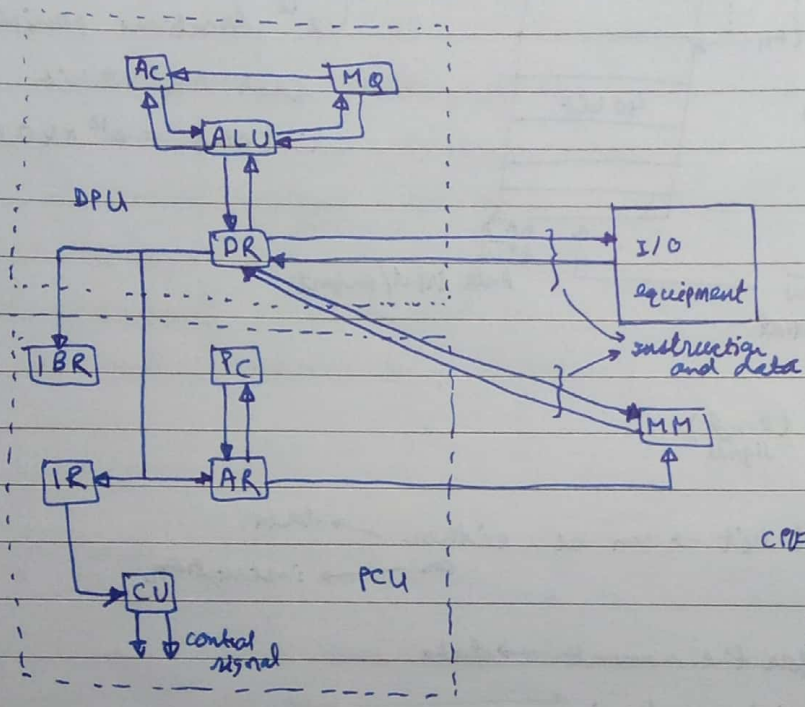
8 bit ← (0 - 7) = Opcode
12 bit ← (8 - 19) = Address
(∵ PC has 12 bit address)

(20 - 27) = Opcode
(28 - 39) = Address

Opcode indicates operation
Address indicates location of available operand

Size of Opcode = n bits → 2^n instructions can be executed by that processor
n varies b/w generations

System Organization of 1G



CPU = DPU + PCU

DPU → Data Processing

PCU → Program control

AC = Accumulator

MQ = Multiplier Quotient

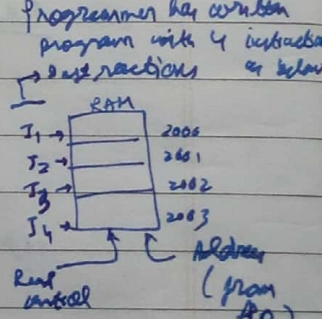
DR = Data Register

IBR = Instruction Buffer Register

PC = Program Counter

IR = Instruction register

AR = Address register.



Function of PC: Holds the address of the instruction to be executed.

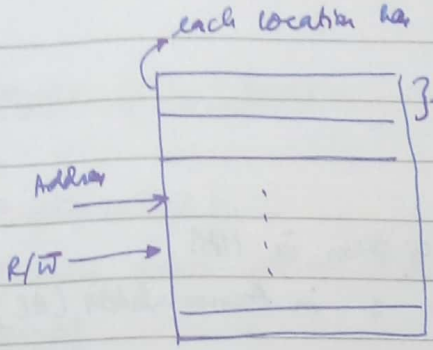
PC sends address of instruction to AR.

AR sends it to main memory.

CU sends control signal to MM to read instruction from the address sent to it by the AR.

4. per 1G, 12 bit address for I_1 is 2000.

∴ MM will be signalled to process 2000 address which contains instruction I_1.



→ location contains 40 bit instruction, [data 40 bit]

↓

Memory can hold 40 bits

But instruction size is 20 bits

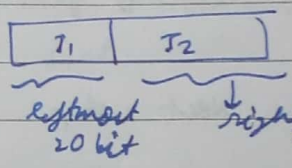
∴ 2 instruction are stored in 1 location

↓

Then from address provided, 2 instruction come out and are stored in the Data register.

∴ DR size is also 40bit

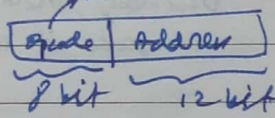
But processor can execute only 1 instruction at a time



∴ 1 instruction enters Buffer Register (IBR)

Let's assume I_2 enters IBR.

Now for I_1 →



From 8 bit opcode, operation to be performed need to be understood. This is called <u>decoding of opcode</u>.

∴ opcode part of the instruction enters IR.

From IR, it goes to CU

IR decodes opcode.

CU generates corresponding signal from this decoded opcode.

From 12 bit address, the address of the operands in main memory can be known. ∴ the address field of the instruction goes to AR. From there it goes to MM. Data comes out and enters DR.

∴ Data is now available at DR.

Next step is execution.

If an operator like '+'
it needs 2 operand.
↙
usually 1 of them in MM
and another 1 in Accumulator (AC)

∴ DR sends 1 operand → sent to ALU
AC sends 1 operand → ALU
CU has already sent control signal from decoded opcode

ALU does operation on the operand
to give output result.

Next step is to store result
↙ either in internal register
↘ or in main memory

Basically, ALU sends it to AC

ALU → DR → MM

To execute instruction I_2,
it is simply retrieved by the process I_2
from IBR, does not need to read from MM
∴ saves time.

MQ is required during multiplication / division

• Floating point Numbers

→ Why needed ?

→ When fixed point numbers not sufficient ?

Answer:

- Range of numbers represented by fixed point number is insufficient for many applications particularly scientific computations where very large and very small numbers are frequently encountered.

  Scientific notation permits such numbers to be represented using relatively few digits.

  eg. 1.0×10^{18} → scientific notation → floating point

  $\underbrace{1000\ldots0}_{18 \text{ zeros}}$ → fixed point

Floating point format:

$$M \times B^E$$

M: Mantissa

B: Base

E: Exponent

→ No. of bits in Mantissa indicate accuracy/precision

eg. $1010.\underbrace{1101}_{3 \text{ bits}}0010 \times 2^2$ [We don't know size of floating point register.

↓

Suppose it can store only 5 bits after decimal

∴ loss of 3 bits of data.

∴ More bits in Mantissa to improve accuracy

→ Exponent field indicates range of the number

eg. For base 2, $E = 8 \Rightarrow (2^0 \rightarrow 2^8)$ is the range.

$= 255$ numbers

1.0×10^{18}
0.1×10^{19}
1000000×10^{12}
0.000001×10^{24}

} same number
∴ unique representation not possible.

↓
But for computers, we would like a unique representation.

∴ Normalization is required to specify a unique normal form for floating point numbers in computer implementation.

The Mantissa part is said to be normalized if there is no leading zero in the magnitude part of the number, that means the bit after the binary point must be 1.

In the above numbers, only 0.1×10^{19} has normalized mantissa.
∴ It is the unique representation of 1.0×10^{18}.

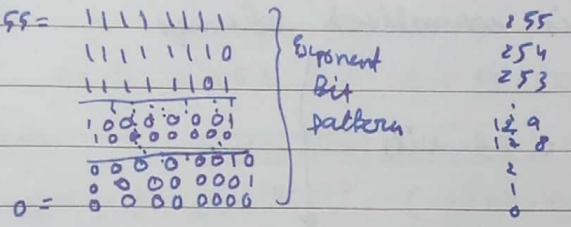
Clearly, the other 3 forms can be normalized, by shifting mantissa towards right or left.

Encoded or Biased exponent

For very small numbers, we need negative exponent.

∴ to represent floating point number, we need signed exponent.

Using 8-bits, list of unsigned exponents is as follows.



Biased exponent = Desired exponent + k
↓ Bias

Bias for 8-bit exponent = 127 or 128
↳ consider this by default, if not specified.

Unsigned exponent	signed exponent	
	Bias = 127	Bias = 128
255	128	127
254	127	126
253	126	125
⋮	⋮	⋮
129	2	1
128	1	0
127	0	-1
⋮	⋮	⋮
2	-125	-126
1	-126	-127
0	-127	-128

unsigned exponent = signed exponent + Bias

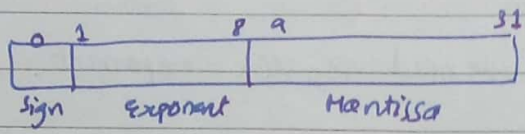
Bias = 127

$(-127 \text{ to } 128) \rightarrow -2^{n-1}+1 \text{ to } 2^{n-1}$

where n = no. of bits in the exponent field

$(-128 \text{ to } 127) \rightarrow -2^{n-1} \text{ to } 2^{n-1}-1$

IEEE 754 standard Floating point Number Format:



32-bit representation.

- -ve numbers ⇒ sign-bit = 1
- Precision is determined by mantissa
- Mantissa is normalized always.

- Mantissa = 23 bits

∴ Numbers are like

$0.1xyz\,abc\ldots$

↳ always 1

∴ don't waste space to store it

Do → $1.xyz\,abc\ldots$

store this in the mantissa (23 bits)

Implicitly adjust the value in exponent value.

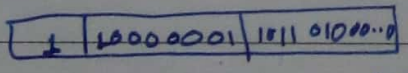
$1.111\ldots1 = M_{max}$

$1.000\ldots0 = M_{min}$ (23 bits)

$$N = (-1)^S \cdot (2)^{(E-127)} \cdot (1.M)$$

$$0 < E < 255$$

eg.



$$N = (-1)^1 \cdot (2)^{(129-127)} \cdot (1.1011\,0100\,00\ldots0)$$

Eg. $N = -1.5$

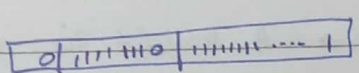
Represent it using 32 bit IEEE floating point format.

$$N = (-1.5)_{10} = (-1.1)_2 = (-1.1) \times 2^0$$

E = desired exponent + bias

$\therefore E = 0 + 127 = 127 \rightarrow (10000000)$

Max value of $E = 254$ $(\because E \in (0, 255))$

$\therefore$ Max number → 

$$N = (-1)^0 \cdot (2)^{254-127} \cdot (1.1111\ldots1)$$

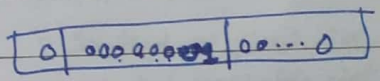
$1.1\ldots1111 = 1 + 0.1111\ldots1$

↳ sum of GP $= \frac{2^{-1}(1+(2^{-1})^{23})}{1-2^{-1}} = 1 - 2^{-23}$

$$= 1 + 1 - 2^{-23}$$

$$= 2 - 2^{-23}$$

$$\therefore N_{max} = (2^{127})(2 - 2^{-23})$$

Min number → 

$$N_{min} = (-1)^0 \, 2^{(1-127)} (1.00\ldots0)$$

$$= 2^{-126}$$

Similarly, -ve N_{max} and -ve N_{min} can be found.

Eg. How is $-6\frac{5}{8}$ represented in 32 bit format

$$-6\frac{5}{8} = -\left(4 + 2 + \frac{1}{2} + \frac{1}{8}\right) = -(110.101)_2 = -(1.10101)_2 \times 2^2$$

(Normalization → write in $(1.M)$ form)

$E = 127 + 2 = 129 = 10000001$

$M = \underbrace{10101}_{5 \text{ bits}} \underbrace{0000\ldots0}_{18 \text{ bits}}$

Note: 0 → can't be normalized

→ represented as $0.M$ where M has 23 zeros

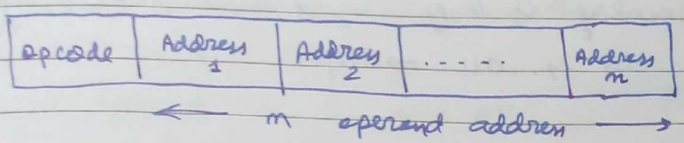
0×2^E → you can consider any value for $E \in (0, 255)$

- Result of a floating point operation is not a valid floating point number. Special code referred to as NaN is used
  eg. $\frac{0}{0}$, $\sqrt{(-ve)}$

- Overflow → Magnitude of the number is too big to represent in used format. Result is referred to as ∞.

- Underflow → A result is non-zero, but too small to represent as a normalized number. Such a number is encoded in de-normalized form characterized by E = 0 and it is represented as 0.M.

- Instruction format



Processor that allows n operand addresses in the address field of the instruction, is called n-address machine.

Problem: Opcode → 8 bit
Address → 10 bits
∴ Total instruction size = 8 + 10n
↓
Which is large → lot of memory needed to store instruction.

∴ we need to reduce the size of instruction

- Reduce opcode size
  - Instead of fixed size, we use variable size depending on how frequently those instructions are needed.
- Reduce address part of the instruction

Method 1:

Variable opcode size (Variable opcode in brackets)

For fixed sized opcode, we would have needed 3 bits per instruction

Fixed length	I	Probability of occurrence P_i			
000	I_1	0.5 (1)	0.5 (1)	0.5 (1)	→ (1)
001	I_2	0.3 (01)	0.3 (01)	0.5 (0)	→ (0)
010	I_3	0.08 (0001)	0.12 (001)	0.2 (00)	
011	I_4	0.06 (0001)	0.08 (000)		
100	I_5	0.06 (0010)			

↳ gives regular Huffman coding.

Average no. of bits per instruction
$= \sum_{i=1}^{5} P_i \cdot c_i = 0.5 + 0.6 + \dots = 1.82$

∴ savings $= \frac{3 - 1.82}{3} \times 100$

Advantage of fixed length opcode:

- Decoding is easy. just read first bit, bit by bit thereafter.
  - ↳ eg. 011 ⇒ I_4

Disadvantage of variable length opcode.

$I_1 \rightarrow 1$

$I_2 \rightarrow 01$

$I_3 \rightarrow 000$

$I_4 \rightarrow 0011$

$I_5 \rightarrow 0010$

Method 1:

Address has to be specified in the instruction if the operands are in memory,

But if operands are in internal register of CPU, we don't have to specify the address.

Operands

Implicit
- stored in CPU registers

Explicit
- stored in physical memory

eg. ADD R_1 (register)

⇒ 1 operand in AC and 1 in Register R_1

$AC = AC + R_1$

Let, if $AC = 03$ (previously)
$R_1 = 04$ (from ADD R_1)
then $AC = 07$ (now)

∴ Instruction contains only opcode. By decoding opcode, processor understands that

Opcode : ADD R_1 | opcode

↓

Decoded by processor to understand that:

→ Operation = Addition

→ Operand in R_1 and AC

→ Destination of result = AC

Not only size of instruction ↓

also speed ↑

(∵ we are using only registers)

ADD M(x) | opcode | x |

⇒ AC = AC + content of memory location whose address is x.

3 address M/C

MULTIPLY T, A, B

(T = A × B)

2 address M/C

MULTIPLY T, B

(T = T × B)

1 address M/C

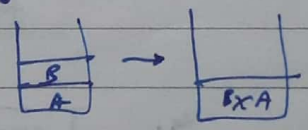
MULTIPLY B

(AC = AC × B)

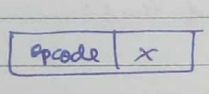
0 - address M/C

MULTIPLY

Stack

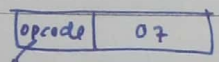


Addressing Modes



To execute instruction, processor requires the current value of x, which can be specified in several ways, known as addressing modes.

→ **Immediate Mode** ----------→ Operand itself specified in address field of instruction.

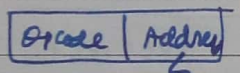
eg. 

eg. ADD x is constant
value of x is 07.

∵ instruction is in memory
2 memory access by processor
→ 1 to read opcode
→ 1 to read operand

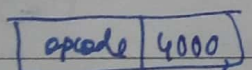
ADD 07 in immediate mode would mean Acc ← AC + 07
(Accumulator)

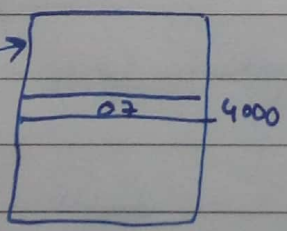
→ **Direct / Absolute Mode**



Address of the operand

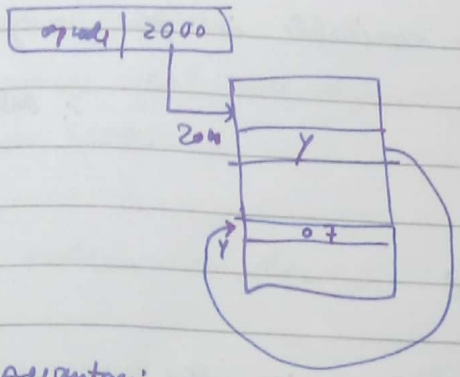
3 memory access by processor.
→ 1 to read opcode
→ 1 to read operand address
→ 1 to read operand

eg. 



- 4000 applied to address bus
- 07 came out of data bus

→ Indirect Addressing mode



Address field contains a pointer that points to a memory location where the operand address is available.

4 memory access to read by processor:

Advantage:

Y can be easily changed to Z for using other operands.

→ Register Addressing mode

ADD B

⇒ AC ← AC + B

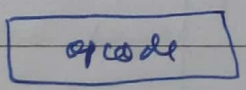
B: Register inside the processor.

∴ It is used, when operand is available inside register

Processor understands that operand is in register B and not register C

∴ opcode → ADD C
≠ opcode → ADD B
} are different.

Instruction format:



∴ size of instruction = size of opcode.

- Only 0 memory access.
- ∴ very less time needed.

→ Register indirect

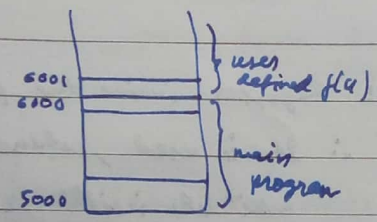
Address of operand is available in the register.

∴ only 1 memory access would be needed

→ Relative Addressing Mode

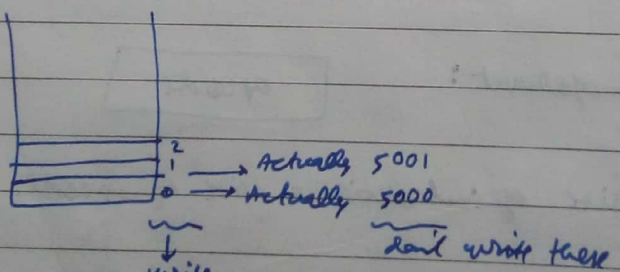
In the address field, we specify the displacement of the operand, not the address of the operand.

- what is displacement of operand?



After modification of main program, it may not fit within (5000 - 6000)
∴ user-defined function may be over-written.

∴ Take R = 5000 → Register



from register + from user program
5000 + 0 = 5000
5000 + 1 = 5001

Base address + Displacement → Effective Address → Applied to address bus of memory to get data.

To avoid the problem of over-writing user-defined function memory,

Take R = 8000

assuming that user-defined function doesn't cross 8000 starting from 6000.

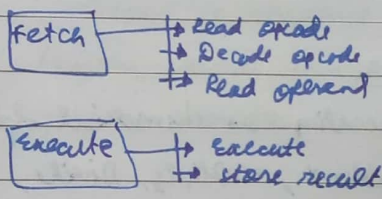
∴ problem resolved.

To represent 5000 → 2^{13} (bits) needed ∴ 16-bit register

Displacement = distance from base address.

ALU is the one which calculate the (+) operation in

Base (+) Disp. = Effective



In some processors, fetch and execute operations go on simultaneously. Then, if they have only 1 ALU,
→ above problem happen.

R = Base register / PC / Index register

↓ used to access indexed items (elements of an array)

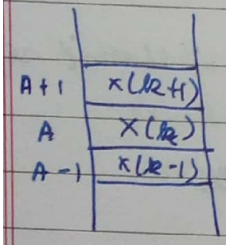
X(0), X(1) ... , X(E)

↓

R

→ Auto increment / decrement addressing mode

Indexed items are frequently addressed sequentially so that a reference to x(k) at memory location A is immediately followed by a reference to x(k+1) or x(k-1) at location A+1 or A-1 respectively.



∴ A sequence of items can be accessed by auto incrementing or decrementing an address.

Instruction implementation

Instructions are divided into 5 major groups:

(1) Arithmetic instruction
- perform operation on Numerical data

eg. Add, subtract, Multiply, Divide,
Absolute, Negate, Increment, Decrement

Absolute → replace operand by its absolute value
Negate → change sign of operand

(2) Logical instruction
- Boolean / non-numerical operation

eg. AND, OR, NOT, XOR, XNOR, Shift,
Rotate, Convert → change the format.

10110 1101 → Rotate → 11010110

(3) Data transfer Instruction

- from source to Destination

eg. Move, Store, Load, Exchange, Clear, Set

Exchange → swap source & destination content

Clear → Transfer all 0's to destination

Source - Memory location
Destination - Register
} Load

Source - Register
Destination - Memory location
} Store

(4) Program control Instruction

- If not used, execution of instructions in program is always in sequence by default

∴ Program control instruction changes sequence of program execution.

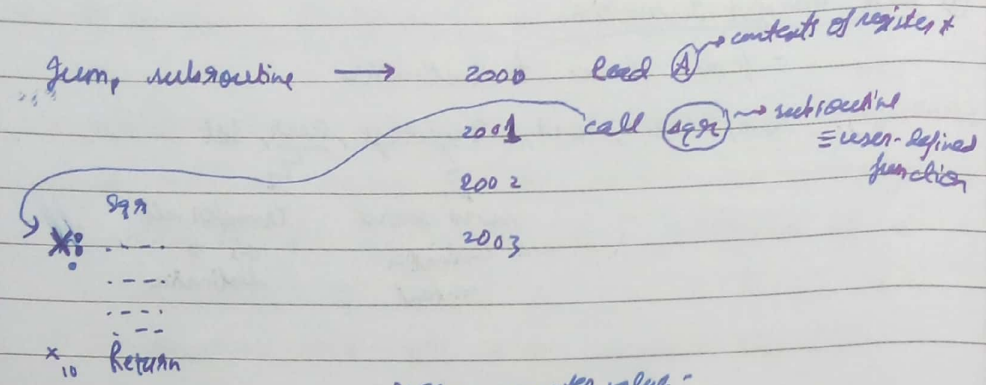
eg. Jump, Jump conditional, Jump to subroutine, Return, Execute, Skip, Test, Compare

2001 : I_1
2002 : I_2
2003 : JUMP X ⇒ I_1 I_2 (Go to instruction at address X and execute it) I_4 I_5
2004 : I_4
2005 : I_5
X : ~~

Jump conditional →

if ($R_1 > 0$)
execute I_2
else
execute I_3

Program counter → contains address of instruction to be executed.



program counter value:

2000
2001
2002 → 2002 calls the subroutine actually.
X_0
X_{10}
2002

2002 stored in stack.
control goes to X
control returned when "Return" to 2002 → popped from stack / stored in program counter.

Skip → Increment program counter to skip next instruction

Test → Specified condition based on outcome.

(5) I/O Instruction

- causes information to be transferred b/w processor or main memory and external I/O devices.

eg. Input, Output, Start I/O, Test I/O, Halt I/O.

For 8085 Machine

Input → read data from i/p device

IN <Port No> → address of input device

IN F4H → Hex value

↓

reads input data and loads into accumulator

OUT F4H → data from accumulator goes to i/p

Start IO → CPU reads instruction
Transfer it to I/O processor (see history of computers)
to initiate I/O.

Test IO → Transfer status information operation from
IO system to specified destination.

Halt IO → To terminate IO operation.

* Arithmetic instruction implementation
  - Hardware requirement in CPU to execute such instructions.
  - Algorithms used for implementation

→ Addition and subtraction with signed magnitude data

	Operation	Add magnitude	subtract magnitude		
			A>B	A<B	A=B ⇒ result = 0, 0 = +0, 0 ≠ −0
Addition	(+A) + (+B)	+ (A+B)			
	(+A) + (−B)		+ (A−B)	− (B−A)	+ (A−B)
	(−A) + (+B)		− (A−B)	+ (B−A)	+ (A−B)
	(−A) + (−B)	− (A+B)			
Subtraction	(+A) − (+B)		+ (A−B)	− (B−A)	+ (A−B)
	(+A) − (−B)	+ (A+B)			
	(−A) − (+B)	− (A+B)			
	(−A) − (−B)		− (A−B)	+ (B−A)	+ (A−B)

A + B

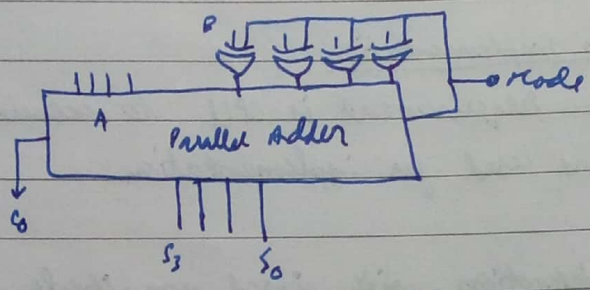
- Two register → initially store A, B; finally one stores result.
- One adder
- One FF (to store carry)

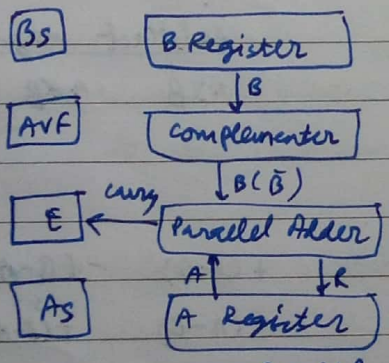
A − B

↳ $A - B = A + \bar{B} + 1$

$\bar{B} + 1$ → 2's complement of B

$\bar{B}$ → 1's complement of B





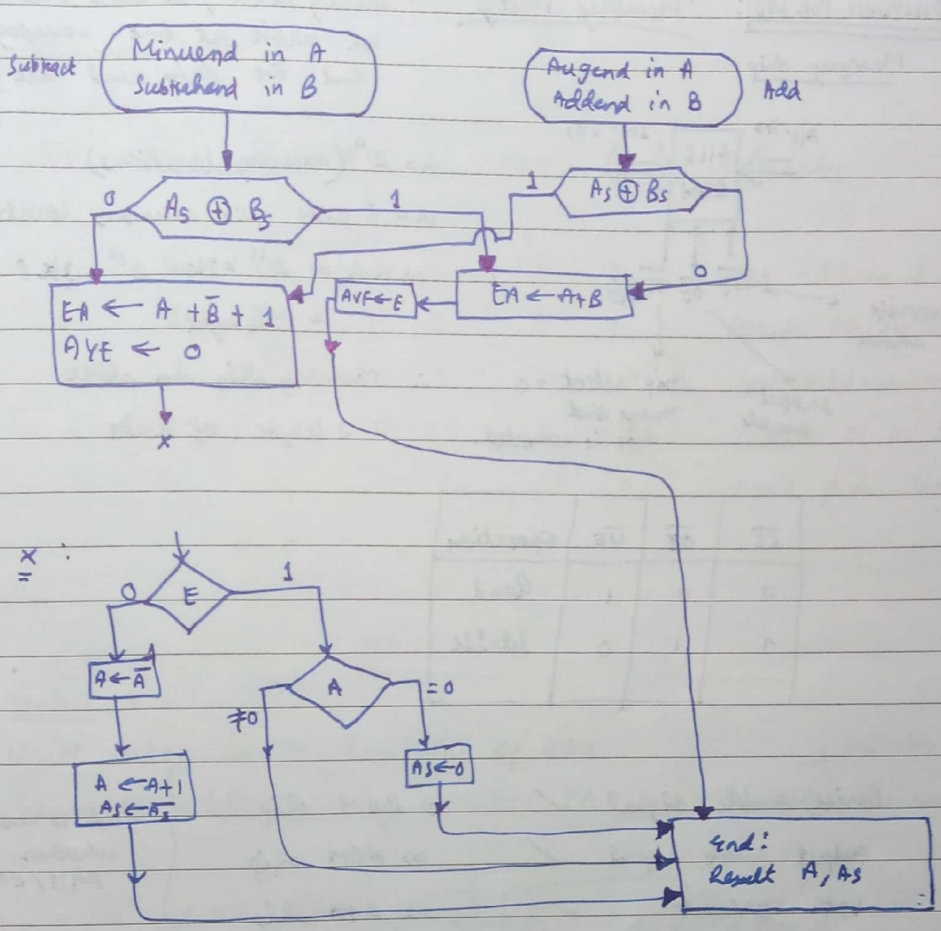
$R = A + B$ or $A + \bar{B} + 1$

As, Bs
↳ for sign magnitude

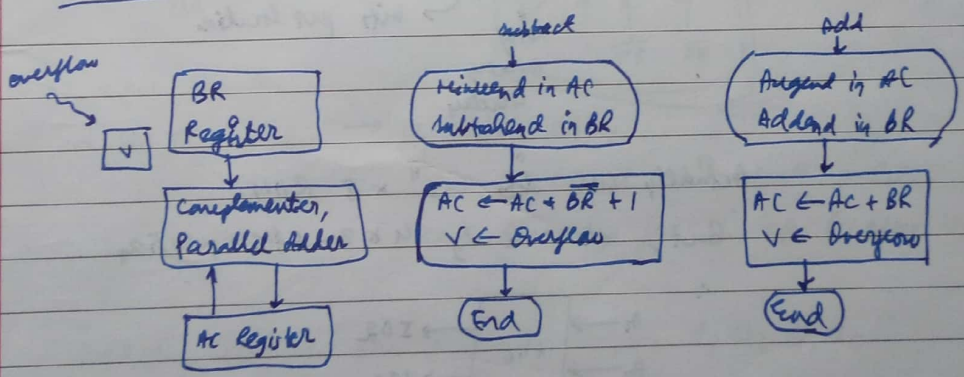
As, Bs = 0 ⇒ +ve

As, Bs = 1 ⇒ -ve

AVF: Overflow (⇒ carry bit generated)
flag



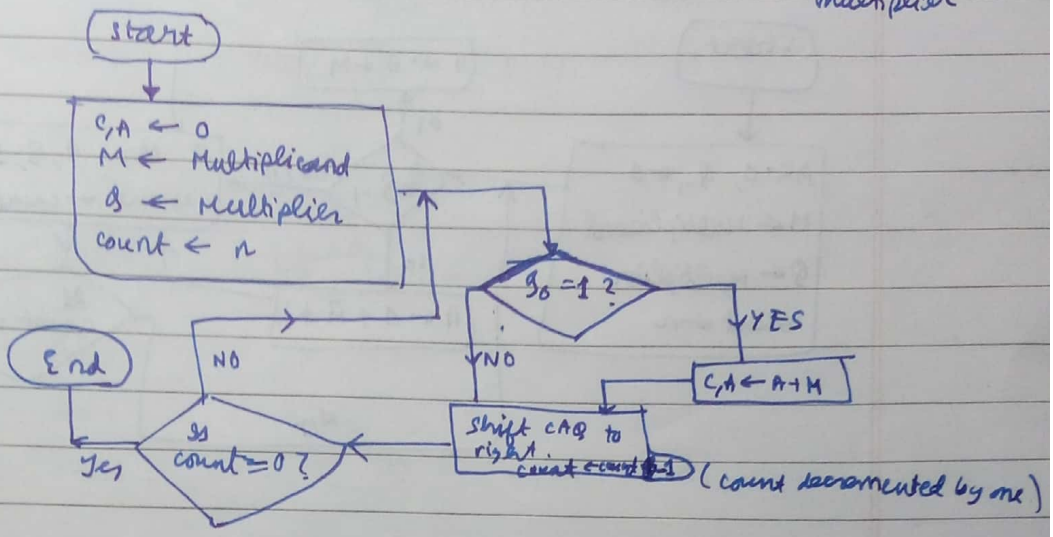
* Addition and subtraction using 2's complement



MULTIPLICATION

Unsigned Integer

count: No. of bits in multiplier



Eg.

```
  1 1 0
× 1 0 1
-------
  1 1 0
0 0 0
1 1 0
-------
1 1 1 0
-------
```

At first, Q = 1

C,A ← A+M

0 0 0 1 1 0

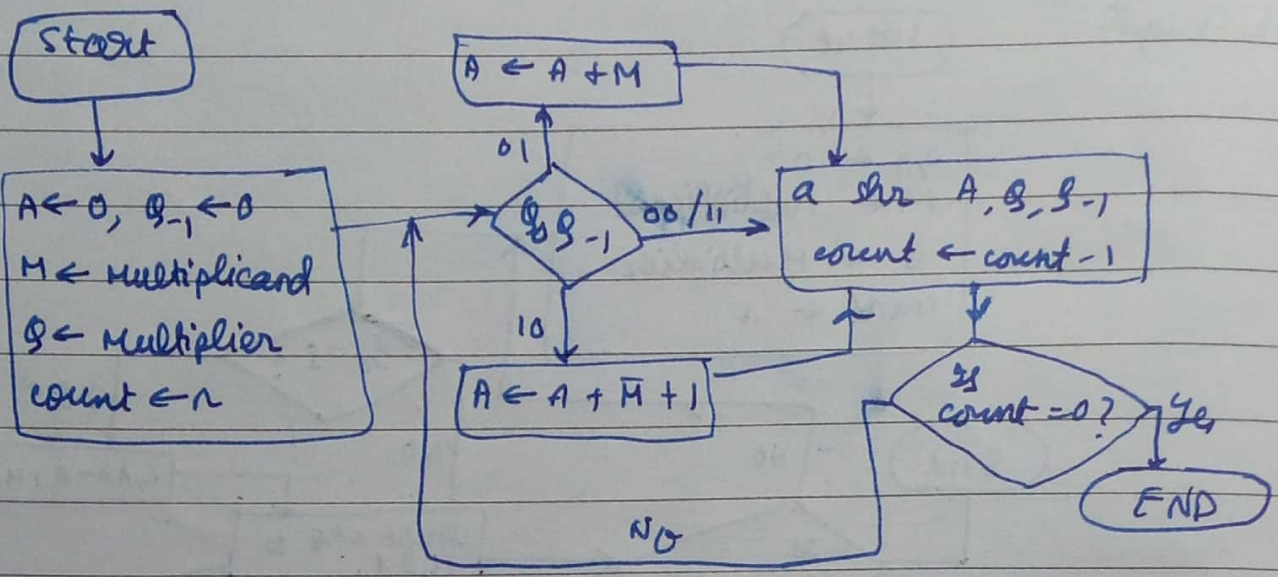
Eq	count	operation	C	A	Q	M
	4		0	0 0 0 0	1 1 0 1	1 0 1 1
		Q_0=1 ∴ Add M	0	1 0 1 1 / 1 0 1 1	1 1 0 1	
	3	shift CAQ	0	0 1 0 1	1 1 1 0	
	2	Q_0=0 shift CAQ	0	0 0 1 0	1 1 1 1	
		Q_0=1, Add M	0	1 0 1 1 / 1 1 0 1		
	1	shift CAQ	0	0 1 1 0	1 1 1 1	
	0	Q_0=1, Add M	1	0 0 0 1		
		shift Q	0	1 0 0 0	1 1 1 1	

Q = 11, M = 13

Q × M = 143

Signed integer multiplication (unsigned also)

Booth's algo



ashr: arithmetic shift right (keeps most significant bit same)

↓

1011

↳ 1101

shr: logical shift

CAQ

1011

↳ 0110

Example: $7 \times 13 = 91$ (A8)
(M) (Q)

M
× Q
AQ

Operation	(Partial product) A	(Q_n) Q	(Q_{n-1}) Q_{-1}	(Multiplicand) M	count
	0000	1101	0	0111	100
$Q_0Q_{-1} = 10$ sub M	0111				
	1001	1101			
ashr →	1100	1110	1		011
$Q_0Q_{-1} = 01$ add M	0111				
	0011	1110			
ashr →	0001	1111	0		010
$Q_0Q_{-1} = 10$ sub M	0111				
	1010	1111			
ashr →	1101	0111	1		001
ashr →	1110	1011	1		000

→ 2's complement of 21
= -21

Assignment - 1:

1) $(-7) \times 3$
2) $(-7) \times (-3)$

worst case: 10101010 } Possible multipliers for Booth's algo.
Best case: 11111111

Booth's encoding:

01110 00000 — 1011 0110

+1 00 -1 00 00 — -1 +1 0 -1 +1 0 -1 0 } encoded multiplier

+ → Addition op
− → sub op

Bit pair multiplication

- Booth's algo has 2 attractive features
  - Handles both +ve and -ve multipliers
  - Efficiency in terms of speed which depends on multiplier

- Bit pair multiplication reduces maximum number of operations and hence improves speed.

i+1	i	i-1	Booths	Bit pair
multiplier Bit pair		Multiplier bit on right Q_{-1}		
0	0	0	0 0 →	0 × M
0	0	1	0 +1 →	+1 × M
0	1	0	+1 -1 →	+1 × M
0	1	1	+1 0 →	+2 × M
1	0	0	-1 0 →	-2 × M
1	0	1	-1 +1 →	-1 × M
1	1	0	0 -1 →	-1 × M
1	1	1	0 0 →	0 × M

+100 -10060

↳ $-1 \times 2^4 + 1 \times 2^7$

+1 -1

↳ $+1 \times 2^1 - 1 \times 2^0 = 2 - 1 = +1$

eg. $13 \times (-6)$

multiplier = -6

$6 = 00110$

$-6 = 11001$

$+1$

11010

we add this bit to make no. of bits in multiplier even.

$[1\,6]\,0\,0$

$i-1$

$000\ 10010$

+ve no

Processor adds this → 2s left blank bits are filled with 0. If -ve, the left blank bits are filled with 1

11010 becomes $11101\ 0\ \ 0$ } Booth's

$00\ -1\ +1\ -1\ 0$

$0\ -1\ \ -2$ } Bit pair

Booths

$13 = 01101$

$-6 = 0\ -1\ +1\ -1\ 0$

$0 \rightarrow 00000\ 00\ 000$

$-13 \rightarrow 11111\ 00\ 11$

$+13 \rightarrow 00001\ +0\ 1$

$-13 \rightarrow 11100\ 11$

$0 \rightarrow 000000$

$111011\ 0\ 0\ 1\ 0$

11^{th} bit

sign bit = (-ve)

10 bits = 78

$\therefore$ Ans. = -78

Bit Pair

$13 = 01101$

$-6 = 0\ -1\ -2$

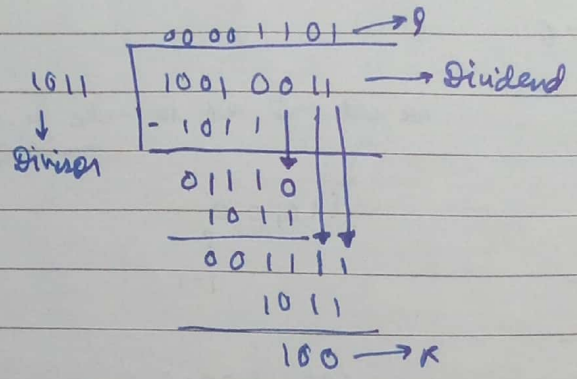
$11111\ 00\ 110 \rightarrow -26$

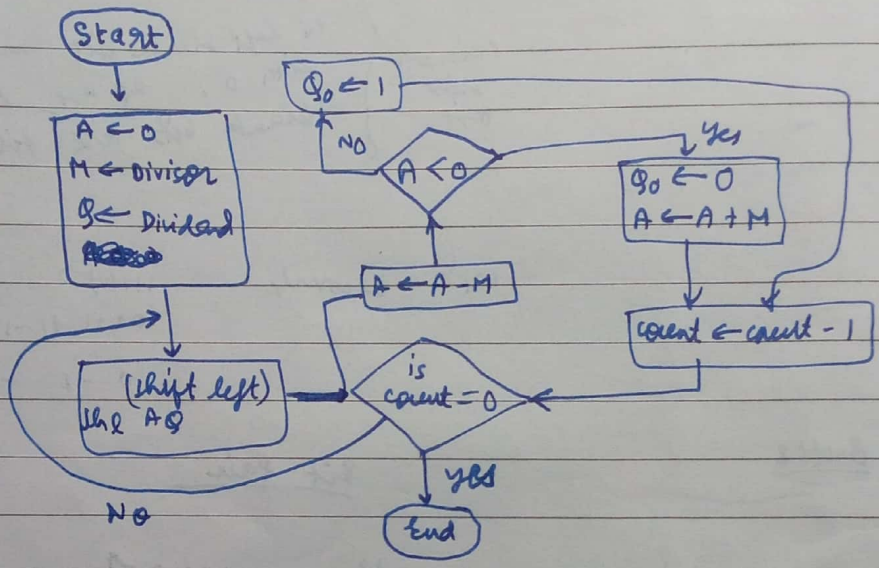
$1111\ 0\ 011 \rightarrow -13$

$\underline{600\ 000 \rightarrow 6}$

$111\ 01\ 10\ 0\ 10$

Division





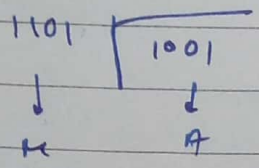
flowchart

↑
only for unsigned integer division.

Eg, $7 \div 3$

	A	Q	M	Count
Initial value	0000	0111	0011	100
Shl AQ	0000	1110		
Sub M	0011			
	1101	1110		
A<0 ∴ add M	0011	1110	. -	011
	0000	1100		
Shl AQ	0001			
Sub M	0011			
add M, $Q_0=0$	1110	1100		
	0011	1100	. — —··	010
	0001			
~~Shift AQ~~	0011			
Sub M	0011			
	0000	1000		
	0000	1001	—— —— -·-·	001
$Q_0 \leftarrow 1$	0001	0010		
~~Shift AQ~~	0011			
Sub M	0011			
	1110	0010		
$Q_0 \leftarrow 0$ Add M	0011	0010		
	0001	0010		

Division of signed integers



$A < M \Rightarrow A - M < 0$

∴ unsuccessful

∴ restore

① Divisor in register M

② Dividend in register AQ

③ shl AQ

④ If M, A have same sign, perform $A \leftarrow A - M$

else $A \leftarrow A + M$

⑤ Preceding operation is successful if sign of A is same before and after operation

⑥ If operation is successful,

$Q_0 \leftarrow 1$

If operation is unsuccessful

set $Q_0 \leftarrow 0$ & restore previous value of A

⑦ repeat till count = 0.

⑧ If sign of divisor & dividend are same, quotient is in register Q,

otherwise correct quotient is 2's complement of Q

Eg. $(-7) \div 3$

Q = 2's complement of 0010
R = 2's complement of 1111

Operation	A	Q	M	Count
	1111	1001	0011	4
Shl AQ	1111			
Add M	0011			
	0010	0010		
Restore	1111	0010		3
Shl AQ	1110	0100		
	0011			
	0001	0100		
Restore	1110	0100		2
Shl AQ	1100	1000		
add M	0011			
	1111	1000		
	1111	1001		1
$q_0 \in 1$	1111	0010		
Shl AQ	0011			
Add M	0010	0010		
	1111	0010		0
	-1	-2		
Letter				

* Combinational array multiplier.

- Advance in VLSI Technology have made it possible to build combinational circuits that perform n×q bit multiplication for relatively high value of operand.
- Advanced micro-devices multiplier chip produced in 1995 can multiply 2 32-bit number in less than 80 ns.

$Y_0\ Y_1\ Y_2$

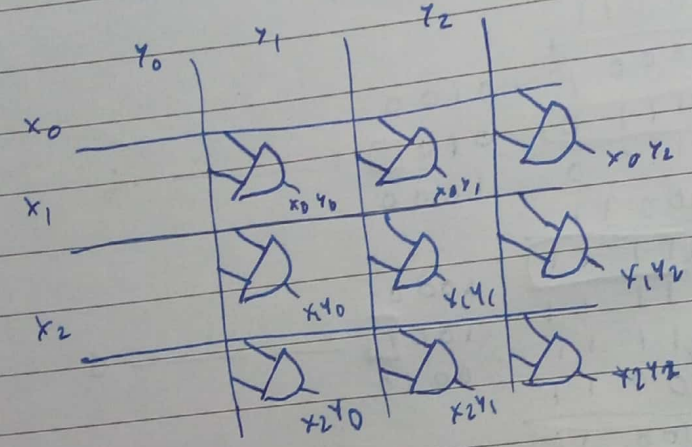
$X_0\ X_1\ X_2$

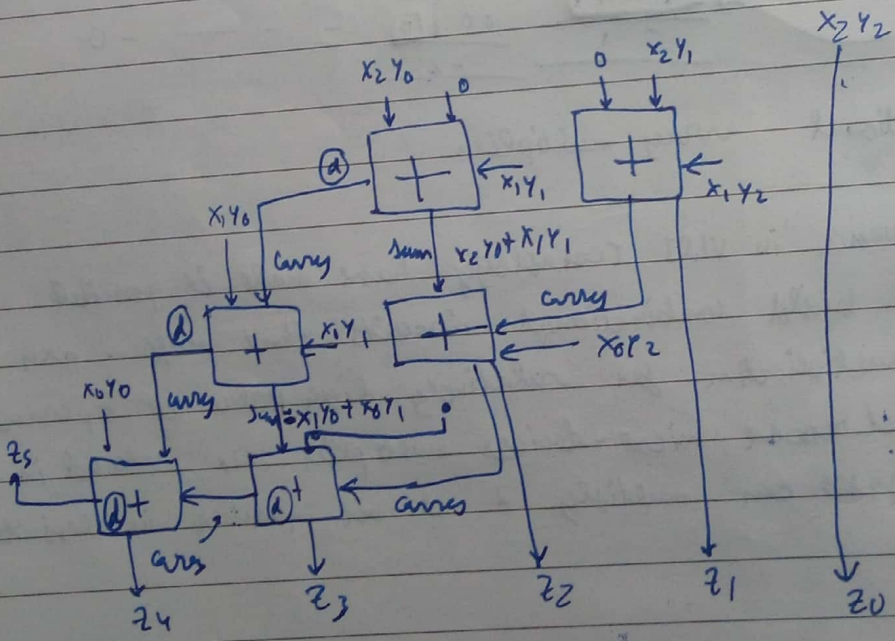
$X_2Y_0\ X_2Y_1\ X_2Y_2$ → $2^0 X_2Y$

$X_1Y_0\ X_1Y_1\ X_1Y_2$ → $2^1 X_1Y$

$X_0Y_0\ X_0Y_1\ X_0Y_2$ → $2^2 X_0Y$

$Z_5\ Z_4\ Z_3\ Z_2\ Z_1\ Z_0$ → $\sum_{j=0}^{2} 2^{n-1-j} X_jY$





[FA. has delay = d] → Digital Logic knowledge

Total delay of circuit above = d + d + d + d = 4d

In terms of n, ← delay of 1 FA.

Total delay = 2(n-1)d

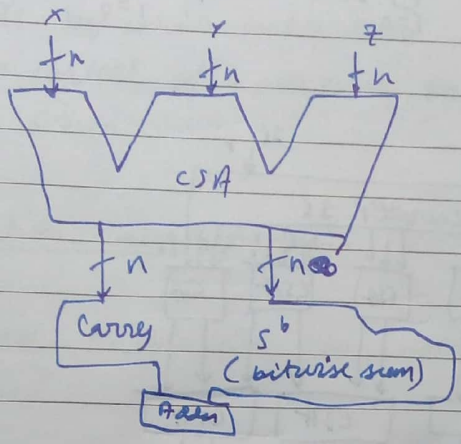
↓

no. of bits in multiplier

No. of bits in multiplier = n

⇒ No. of partial products = n

Carry save Adder (CSA)



$S^b = (0, S_{n-1}, \ldots, S_1, S_0)$

↳ leading bit always 0.

$C = (C_n\ C_{n-1} \ldots C_1, 0)$

↳ always 0

$S_i = X_i \oplus Y_i \oplus Z_i$

$C_{i+1} = (X_i Y_i)$ or $(Y_i Z_i)$ or $(Z_i X_i)$

$S_0 = 1$ $\quad i = 0$

X = 0 0 1 0 1 1

Y = 0 1 0 1 0 1

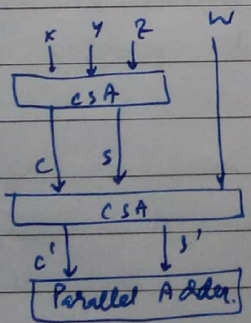
Z = 1 1 1 1 0 1

S^b = 0 1 0 0 0 1 1

C = 0 1 1 1 0 1 0

1 0 1 1 1 0 1

$X+Y+Z+W$

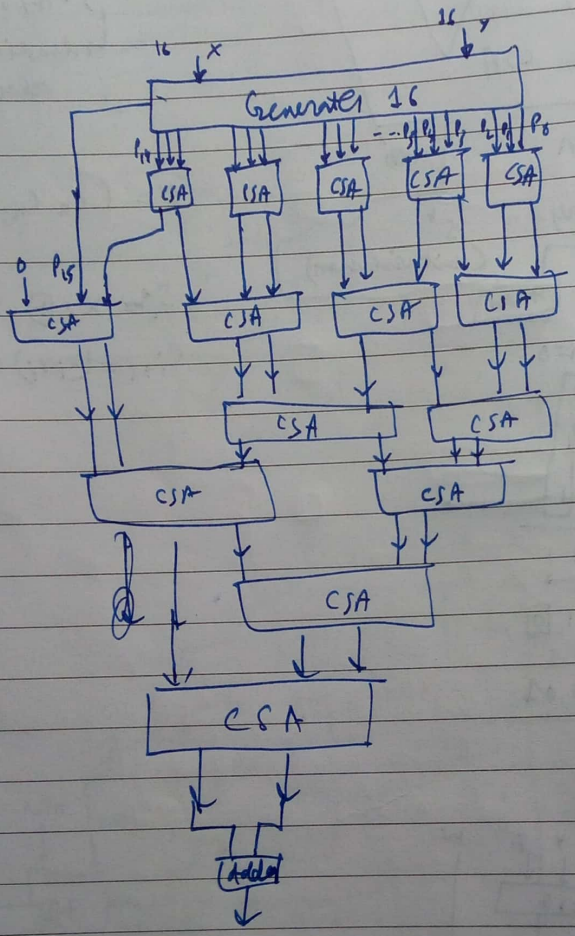


Multiplication using CSA

n - bit M in Y

n - bit Q in X

Product $P = \sum_{i=0}^{n-1} X^i 2^i Y = \sum_{i=0}^{n-1} P_i$ → Partial products



Hardwired CU
↑ → Microprogrammed CU

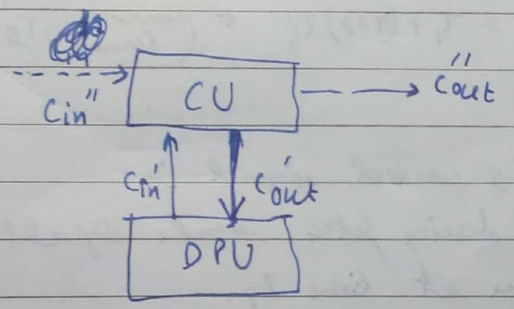
Control unit → Instruction Sequencing
→ Instruction interpretation

→ Instruction sequencing

Methods by which instructions are selected for execution, or how control of processor transfers from one instruction to other

→ Instruction interpretation

CU interprets instruction to determine control signal to be issued.



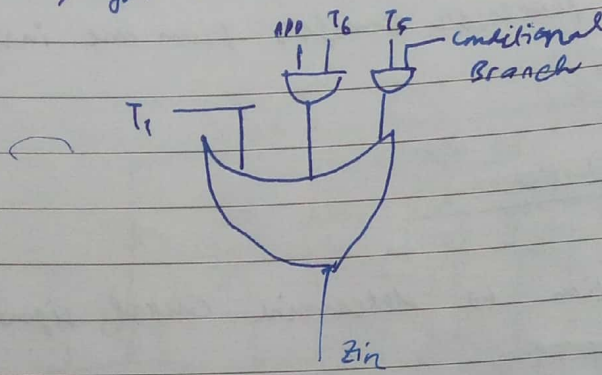
C_{out}' directly control operation of DPU.
C_{in}' indicate occurence of unusual condition such as errors in DPU.
C_{in}' allows data-dependent decisions to be made.

C_{in}'' receives from other CU (for eg. from a supervisor or controller). Typically include START, STOP, timing info.

C_{out}'' transmitted to other CU may indicate statues condition such as busy, operation condition, etc.

Hardwired CU

Eg. Let control signal Z_{in} to be activated at
T_1 for all instance.
T_6 for ADD instruction
T_5 for condition branch Instruction.

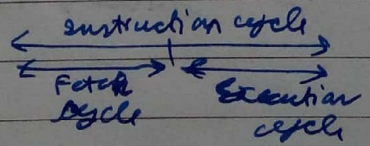


$$Z_{in} = T_1 + (ADD)T_6 + (\text{cond. branch})T_5$$

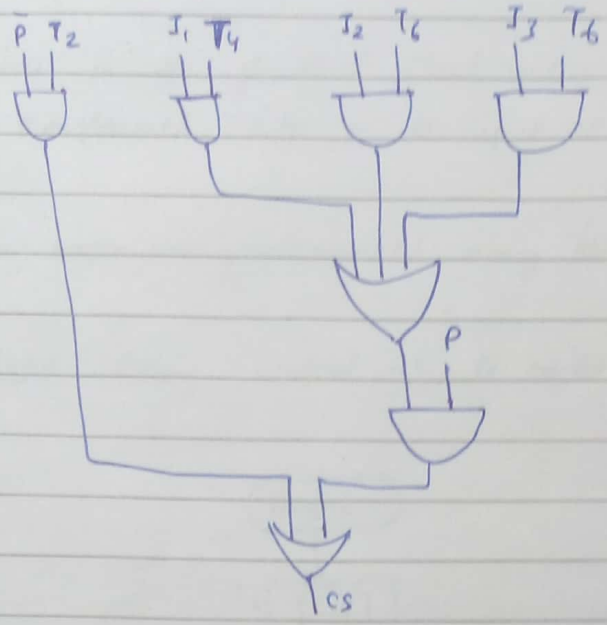
Eg. Suppose C_5 control signal is required during fetch cycle of all instruction at time T_7.

During execution of I_1 at T_4
... I_2 at T_6
... I_3 at T_6.

$p = 0$ for fetch
$p = 1$ for execute



$$C_5 = \bar{p}\,T_2 + p(I_1 T_4 + I_2 T_6 + I_3 T_6)$$



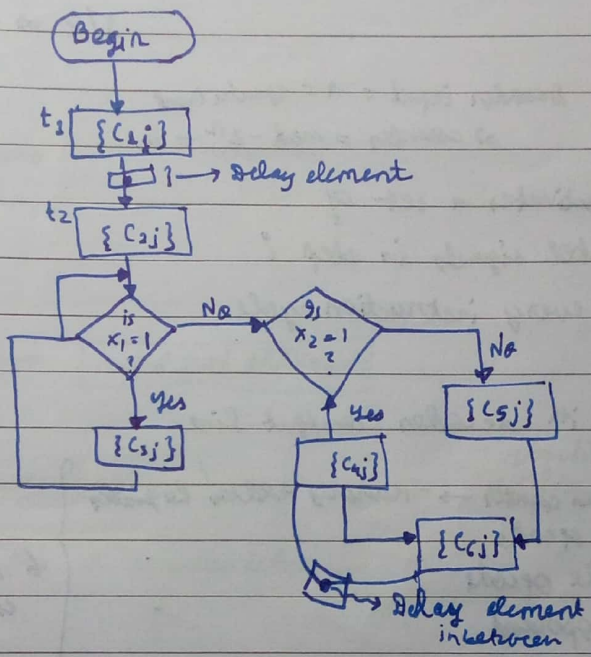
Hardware Implementation for previous example

Next, 3 methods, we will study.

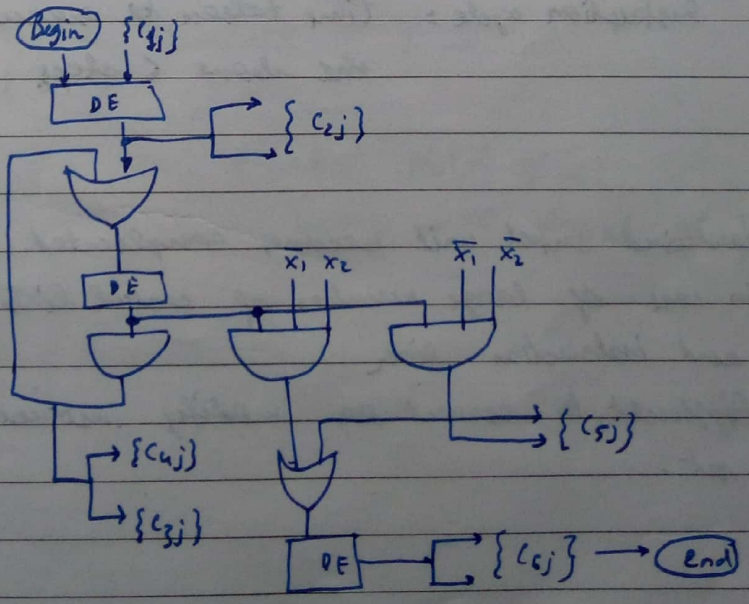
1. State Table Method
2. Delay Element Method
3. Sequence counter Method

Rules:

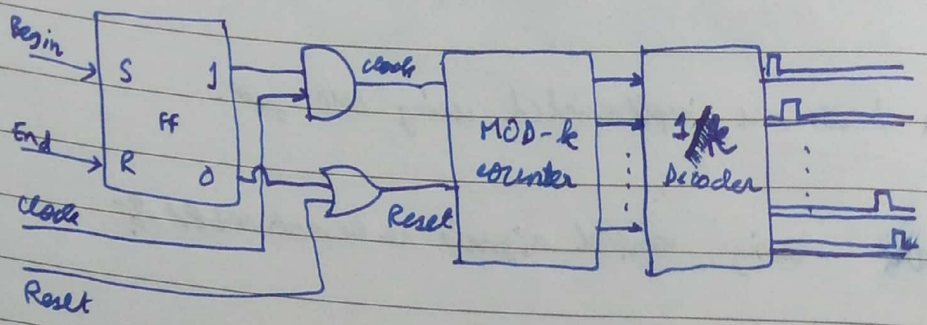
- If lines in the flowchart that merge into a common line are transformed into a fe-input OR gate.
- Decision boxes are implemented using AND gate.
- Flowchart showing control signals to be activated :-



- Circuit Diagram using Rules (to convert the flowchart above) :



Sequence counter Method:



1/k ⇒ 1 output and 1 input only are high at any time.

Decoder input = n = counter input
⇒ counter = mod-2^n = k

ϕ_i activates a set of control signals in step i of every instruction cycle.

ϕ_i : i^{th} decoder output line

$\{C_{1j}\}$ 1. Program counter → Memory Address Register
$\{C_{2j}\}$ 2. Read opcode
$\{C_{3j}\}$ 3. Decode opcode
$\{C_{4j}\}$ 4. Read operand
$\{C_{5j}\}$ 5. Execute I
$\{C_{6j}\}$ 6. Store result

} 6 steps to execute 1 instruction

Instruction cycle : Time taken to execute all the above 6 steps.

- Hardwired control unit becomes complicated in case of large numbers of control lines and instruction lines.
- Difficult to correct or modify instruction set.

Microprogrammed CU

I Micro-program (μp)

I_1 $\mu P_1 \rightarrow [\mu I_1, \mu I_2, \ldots, \mu I_m]$

I_2 μP_2

I_n μP_n

microprogram micro-instruction

microprogram

$\mu P_1 \xrightarrow{x} \mu I_1 \rightarrow$ [control field | address field]

$I_1 \rightarrow$ [opcode | addr] $\quad x$: starting address of μP_1

how this x is calculated

(for the previous past example, $\mu I_1, \mu I_2, \mu I_3, \ldots, \mu I_6$)

x μI_1 [control] [$x+1$]

$x+1$ μI_2 [control] [y]

y μI_3

↳ control field of μI_1 contains control signal of $\{c_{ij}\}$ group

Instruction = operation

↳ can be divided into a few μOperations

Load X

Load accumulator by content of memory location X.

$AC \leftarrow M[X]$

Instruction: Load R

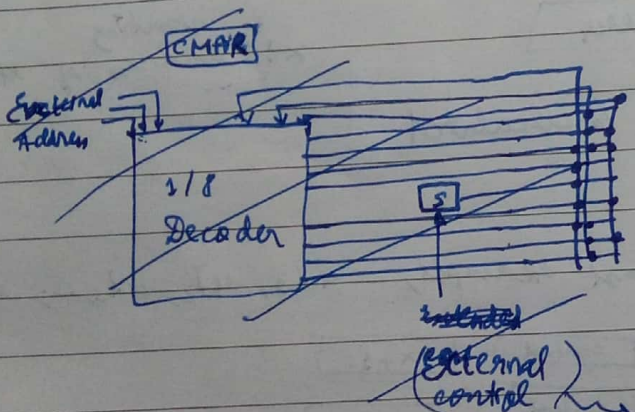
Micro-Operation: $R \leftarrow R_1$

$R \leftarrow R_2$

$R \leftarrow R_3$

$R \leftarrow R_4$

Wilkies Design



3 bits per address

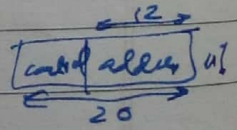
8 micro-instructions

$2^3 = 8$

∴ these 2 are related

eg: (if AC = 0)?

Redraw at home



with 12 bits,

No. of add $= 2^{12}$

∴ bits of control memory $= 2^{12} \times 20$

S → switch

→ Micro-Programmed control unit is slower than hardwired control.

μProgram counter goes from 001 to 001.---
Addres field of μInstruction points to next one (∵ can be found by adding one to initial)

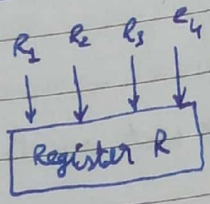
Then, don't need to provide
address field of μ-Instruction.
Just send external address from μProgram counter

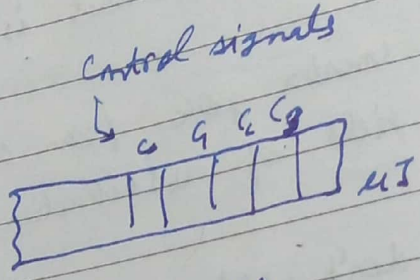
Instructions in RAM
μProgram in control memory } → 2 different memories involved

Read opcode of I_1,
Decode opcode of I_1,
From opcode, you get starting address of μP_1
μP_1 had 10 μI.
∴ Time for I_1 = Total time for fetch and execute of all the μI. for μP_1.

∴ This is why μP control unit is slower.

Horizontal MI format





* $R \leftarrow R_1$
$R \leftarrow R_2$
$R \leftarrow R_3$
$R \leftarrow R_4$

eg. $C_0 \rightarrow 1000$
$C_1 \rightarrow 0100$
$C_2 \rightarrow 0010$
$C_3 \rightarrow 0001$
No operation $\rightarrow 0000$

Horizontal = long format
n - control signals needed
⇒ size of control field = n

Vertical MI

K_1 K_2 K_3	
0 0 0	⇒ NOP
0 0 1	⇒ $R \leftarrow R_1$
0 1 0	⇒ $R \leftarrow R_2$
0 1 1	⇒ $R \leftarrow R_3$
1 0 0	⇒ $R \leftarrow R_4$

We can have atleast 1 bit per MI.

By Reading we get $K_1K_2K_3$
∴ Less bits than Horizontal.

But to decode $K_1K_2K_3$ into $C_0C_1C_2C_3$ we need extra decoder in case of vertical.

Height = No. of instructions
Width = Size of instruction

} Attempts have been made to reduce this.
↳ reduce no. of instructions reduces height

Size of control memory = Width × Height

- Examples of Independent & Dependent.

AC ← 0
AC ← AC - DR
PC ← AR
AC ← AC OR DR
PC ← PC + 1

In micro-programming level, independent μI are available & executed in parallel Hence one μI can specify signals that can execute multiple micro-operations

HLEP =

Parallelism in μI

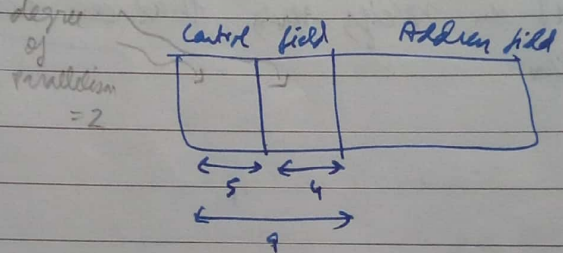
9) No. of uOperation = 46
Degree of parallelism = 2
Size of control-field = 9

$9 = 7+2$
$= 8+1$
$= 6+3$
$= 5+4$
} which one?

$9 = 5+4$

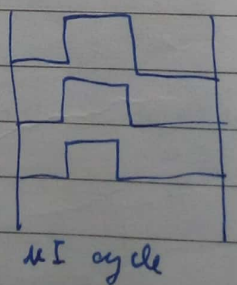
Because $2^5 + 2^4 = 48$
46 is closest to 48
and 2 operations are no operation

$\therefore 46+2 = 48 = 2^5 + 2^4$



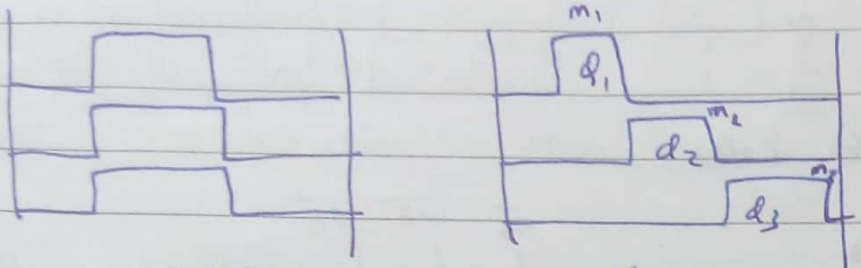
Micro-operation timing

m_1: $R_2 \leftarrow f(R_1) \rightarrow \mu I_1$
m_2: $R_3 \leftarrow f(R_2) \rightarrow \mu I_2$
m_3: $R_4 \leftarrow f(R_3) \rightarrow \mu I_3$



Monophase: Control signals required to execute micro-operation, remain active throughout μI cycle.

Date
Page



monophase → polyphase

no. of M.I reduces from 3 to 1

↓

Though M-Op are dependent, but we can execute them in 3-different phases in the same M I.

Back to micro-operation timing. (polyphase)

Micro-code compaction (minimizing H)

①

NAME	μ-Operation	Phase used	Resource used
m_1	RLEFT ← AC	ϕ_1	
m_2	RRIGHT ← 1	ϕ_1	
m_3	ROUT ← RLEFT + RRIGHT	ϕ_2	Adder
m_4	MAR ← RIGHTSHIFT(R_{OUT})	ϕ_3	Shifter
m_5	M(MAR) ← MDR	ϕ_1	Bus.
m_6	RLEFT ← 1	ϕ_1	
m_7	RRIGHT ← R_0	ϕ_1	

$MI_1 = \{m_1, m_2, m_3, m_4\}$

$MI_2 = \{m_5, m_6, m_7\}$

∴ no. of μ-instructions = 2 for polyphase

∴ height ↓ [H = 7 ⟶ H = 2]

②

	NAME	μ-Operation	Phase	Resource
H = 5	m_1	$A \leftarrow R_1$	ϕ_1	
	m_2	$C \leftarrow A + R_2$	ϕ_2	Adder
	m_3	$R_3 \leftarrow D$	ϕ_2	
	m_4	$R_6 \leftarrow R_6 + 1$	ϕ_2	Adder
	m_5	$R_7 \leftarrow R_3$ or R_6	ϕ_2	ALU

$\mu I_1 = \{m_1, m_2, m_3\}$

$\mu I_2 = \{m_4\}$

$\mu I_3 = \{m_5\}$

H = 3

$\mu I_1 = \{m_1, m_3, m_4\}$

$\mu I_2 = \{m_2, m_5\}$

H = 2

further reduce

Minimizing W

Objective is to derive a format of the micro-instruction control field such that the total number of bits in the control field is minimum.

∴ We want a control field encoding method that uses as few bits as possible

Compatibility class

c_i = control signal

I_j = Micro-instruction

$c_i \in I_j \rightarrow c_i$ is contained by I_j

control signals c_1 & c_2 are compatible if $c_1 \in I_j$ but $c_2 \notin I_j$

Maximal compatibility class

Defined as compatibility classes to which no control signal can be added without introducing a pair of incompatible control signals

Eg.

μI	control signals
I_1	a, b, e, g
I_2	a, c, e, h
I_3	a, d, f
I_4	b, c, f

a, b, c, d, e, f, g, h - 8 control signals
Target: To reduce "g" using some logic as below

Algo is enacted step-by-step next

Step-1: Determine the set of MCCs.

S_1 (a) ~~b c d e f g h~~

S_2 ~~bd, be, bh,~~ (cd) ~~de, dg, dh, ef, eg, fg, fh, gh~~

S_3 (bde) (bdh) (deg) (dgh) (efg) (fgh)

a, b	b, a	ca
a, c	b, e	cb
a, d	b, d	cd
a, e	b, c	ce
a, f	b, f	cf
a, g	b, g	cg
a, h	b, h	ch

may need to continue for other different examples → { : :

Process terminates when no new compatibility classes can be formed.

Step-2: Determination of all minimal set of MCCs that include each control signal. Each of these set is minimal MCC cover.

COVER TABLE →

c_i \ c_j / MCC	a	b	c	d	e	f	g	h
$C_1 = a$	×							
$C_2 = cd$			×	×				
$C_3 = bde$		×		×	×			
$C_4 = bdh$		×		×				×
$C_5 = deg$				×	×		×	
$C_6 = dgh$				×			×	×
$C_7 = efg$					×	●	×	×
$C_8 = fgh$						×	×	×

- Each row is a compatibility class obtained above
- Each column is a control signal

Certain rows and columns can be deleted from cover table to simplify determination of minimal MCC cover.

→ column c_j contains only 1 × occurring in row c_i, then c_i is said to be essential MCC and they must appear in every minimal MCC cover since it only covers c_j. All rows corresponding to essential MCCs can be

C_1, C_2 essential

deleted from cover table and all columns with × mark in essential rows can be deleted.

column a, c deleted

→ Column C_i is said to dominate column C_j if C_i contains a × mark in every row where C_j contains a × mark and C_i has more ×.

column d, g deleted

Dominating column can be deleted as every MC that covers C_j automatically covers C_i.

→ Row C_i dominate C_j if C_i contains a × in every column where C_j contains × and C_i containing more × than C_j.

Rows 2, dgh deleted

Dominated row C_j can be deleted since C_i covers all control signals that are covered by C_j.

Minimal cover for this table is

* { C_1, C_2, C_4, C_7 }
  a c bdh efg

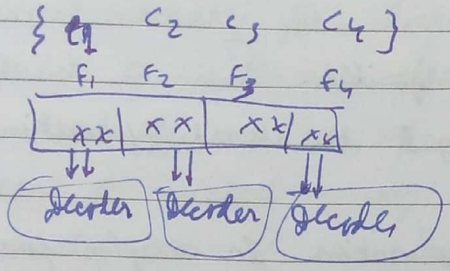
* { C_1, C_2, C_3, C_8 }
  a c bde fgh

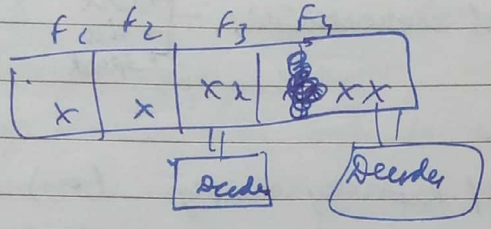
→ There are the 2 choices possible.

C_2 = cdf = cd

C_3 = bde = be

C_8 = efgh



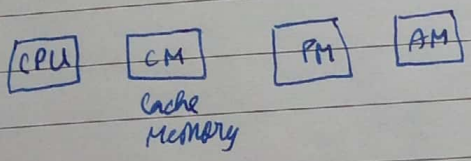


MEMORY SUBSYSTEM

= storage devices + Algorithm
to control or manage the stored information.

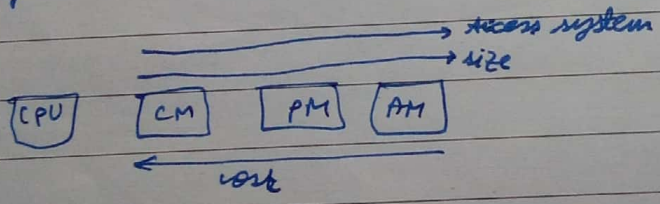
- Distributes stored information in complex fashion over a variety of different memory units with different physical characteristics. → cost, size, speed

- Eg. → Register
→ Physical / Primary memory (PM)
→ Secondary / Auxiliary memory (AM)



Logical position of CM is inbetween CPU and PM.

Objective of memory system design.



- Memory system provides adequate storage capacity with an acceptable level of performance at a reasonable cost.

- Methods to fulfill objective:
1. Use of a number of different memory devices with different cost / performance ratio organized to achieve high average performance at a low cost per bit.

2. individual memory for a hierarchy of storage devices
3. Development of automatic storage allocation method to make most efficient use of most available memory space.
4. Development of virtual memory / auxiliary memory
   - Development of virtual memory concept to make ordinary user free from memory management and make programs largely independent of memory configuration.
5. Design of communication links to the memory system so that all processors connected to it can operate at or near their maximum rate.

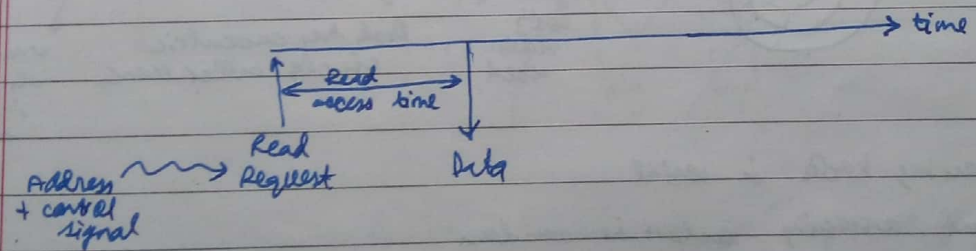
Memory device characteristics

- cost → cost of information storage cells and access circuitry.

  C = cost in Rs. C = cost per bit = $\frac{C}{S}$

  S = storage capacity in bits

- Access time (t_A)



(similarly for write access time)

- t_A should be low.

• Access mode

→ Random access mode

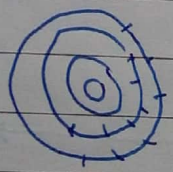
→ Serial access mode

- Random access mode
  = Each storage location can be accessed independently of other locations
  = costly, as every location has its own access mechanism

  eg. ferrite-core ...

- Serial access mode
  = Access mode shared across memory locations.
  = high t_A, low cost, t_A depends on location to be accessed
  eg. magnetic tape, audio cassette

→ Magnetic Disks



access [(side), (track no.), (sector no.)]

- side: both sides used
- track no.: Disk has concentric circles called tracks
- sector no.: tracks divided into sectors

Accessing track is serial

but accessing sector is random

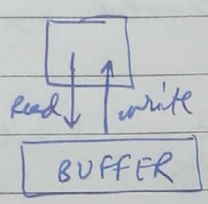
iff every track has its individual read-write head

Permanence of storage

- Destructive Read out (DRO)
- Dynamic Memory
- volatile

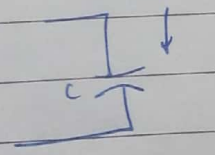
→ DRO

If Method of reading memory destroys stored information, then we store data in a buffer, and follow each read operation by a write operation.

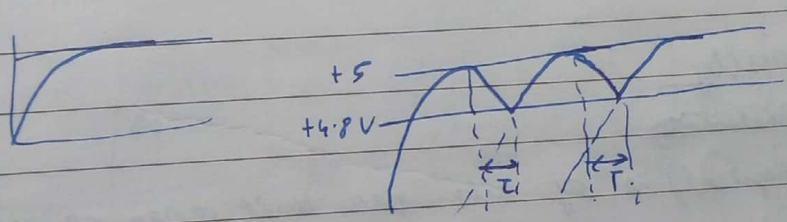


→ Dynamic Memory

Data determined by capacitor



when C is fully charged, content of memory location is logic 1.

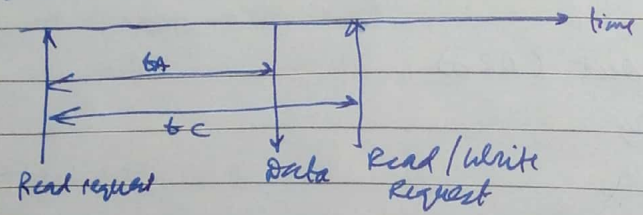


Every τ seconds, "refresh" operation occurs to keep C fully charged

→ volatile

Semiconductor memories are volatile
Magnetic memories are not volatile

Cycle time



$t_c \geq t_A$

t_c: cycle time

for RO, one read cycle includes one read followed by one write.

Data transfer rate (b_m) = $\frac{1}{t_c}$ (bits/sec or words/sec)

Bus width (w) = No. of bits that can be transferred simultaneously for that memory bus.

$$b_m = \frac{w}{t_c}$$

Storage density

~~[illegible]~~

→ (Physical) no. of bits per unit area or volume

→ Determines portability of memory.

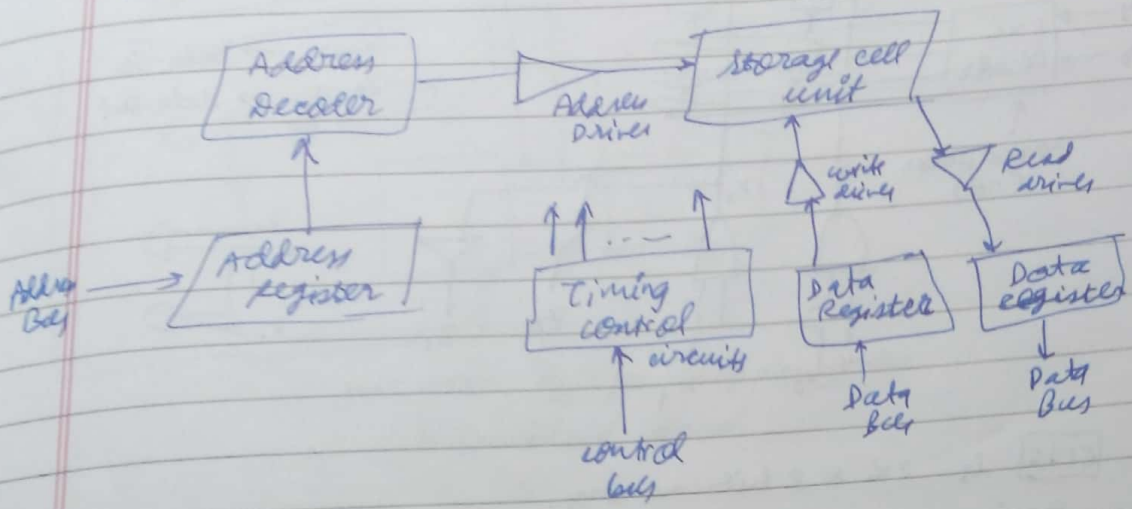
→ Large energy consumption combined with high storage density requires expensive cooling equipment.

→ Reliability - less reliable if parts are moving.

- less reliable if storage density is very high or data transfer rate is very high.

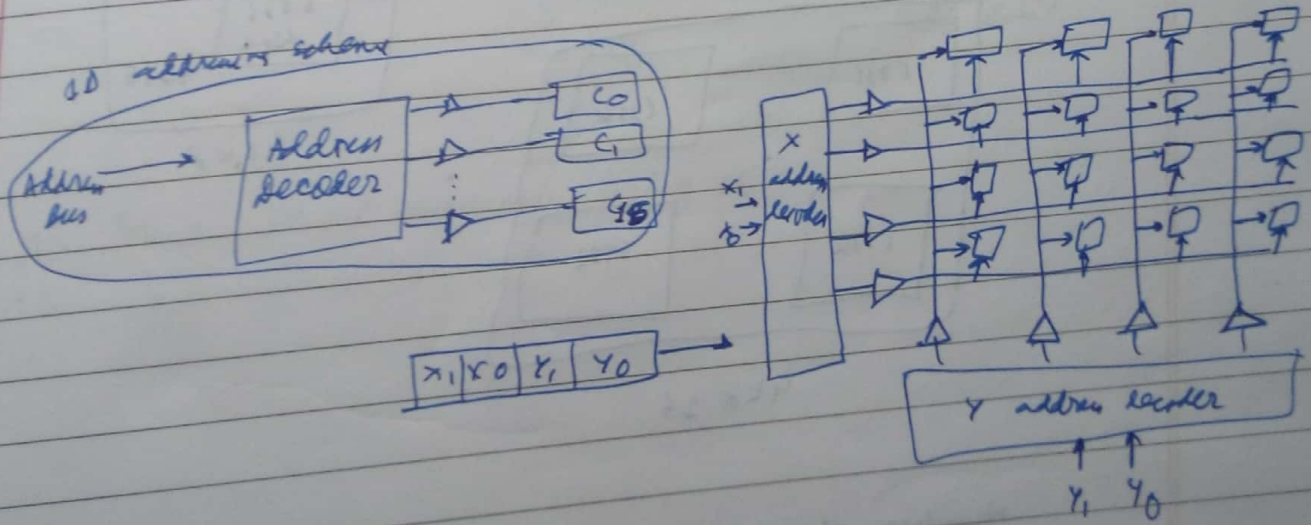
Date
Page

Random access memory

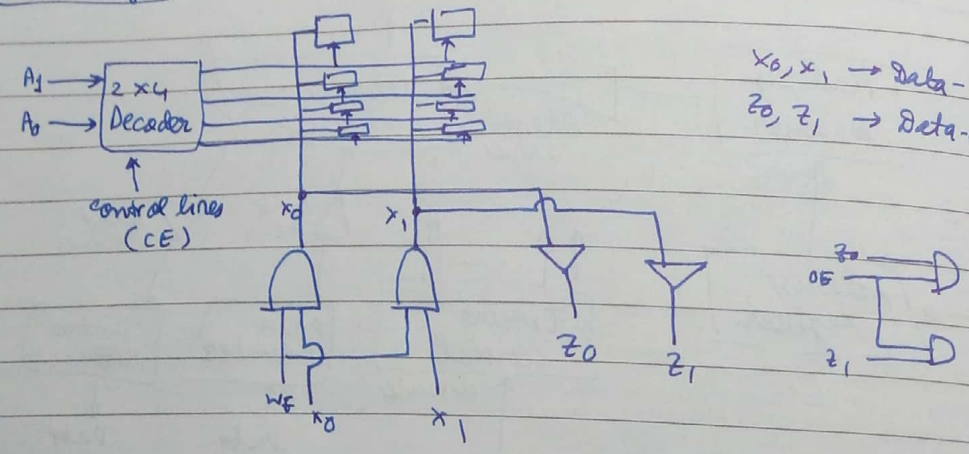


Drivers maybe → Amplifier
→ Transducer

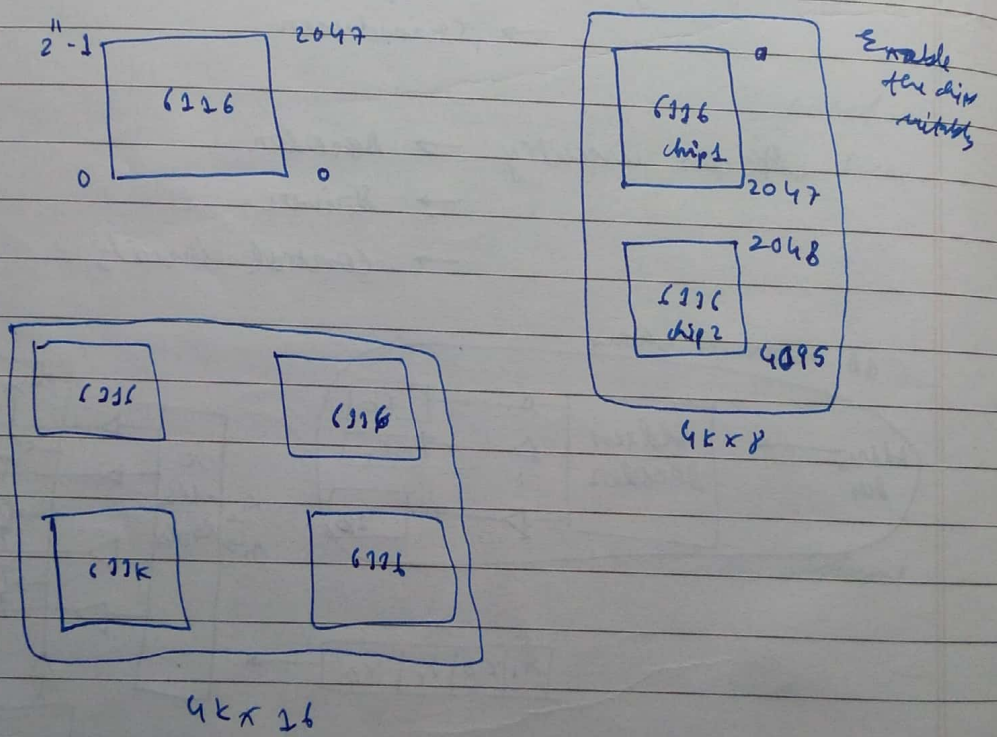
Access circuitry → Decoder
→ Driver
→ control circuits



RAM design



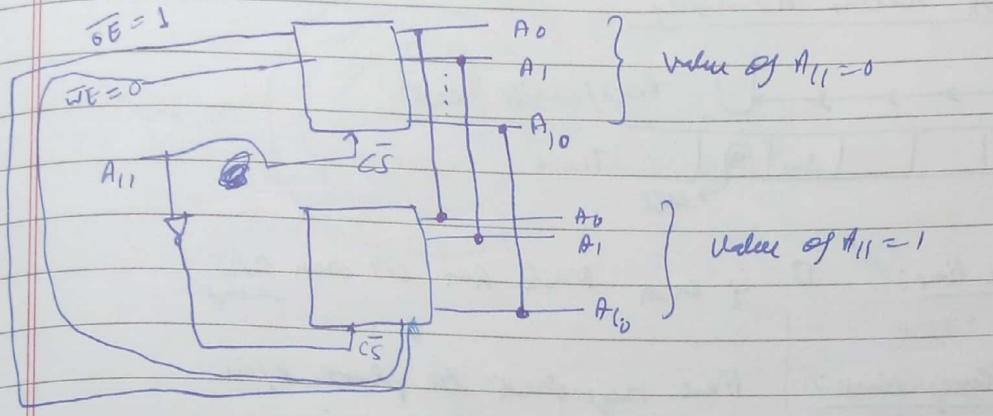
Q) 6116 is 2K x 8 bit memory.
Design 4K x 8 bit memory using 6116.



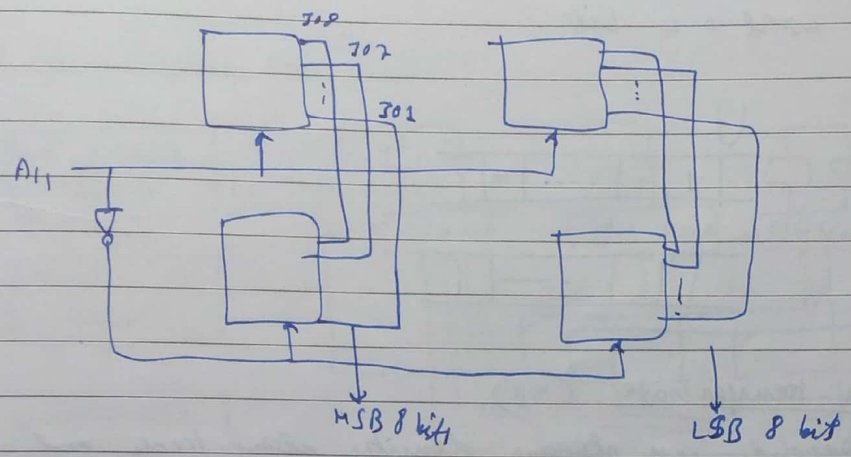
Number of rows in these designs

$$= \frac{\text{No. of desired memory location}}{\text{No. of available memory location}} \quad \left(\text{eg. } \frac{4K}{2K} = 2\right)$$

Number of columns $= \frac{\text{No. of desired bits per location}}{\text{No. of available bits per location}} \quad \left(\text{eg. } \frac{8}{8} = 1\right)$

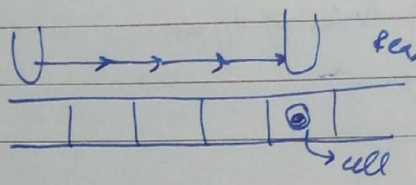


We use A_{11} for supplying the chip-select signal



H.W. 9) Design 32 x 8 memory using 16 x 4.

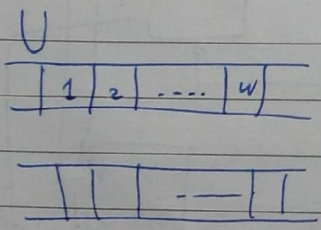
Serial Access Memory



Seek time: 0 if each track has its own R/W head

Latency time: time required to place R/W head over the required cell in a track

1 word = w bits



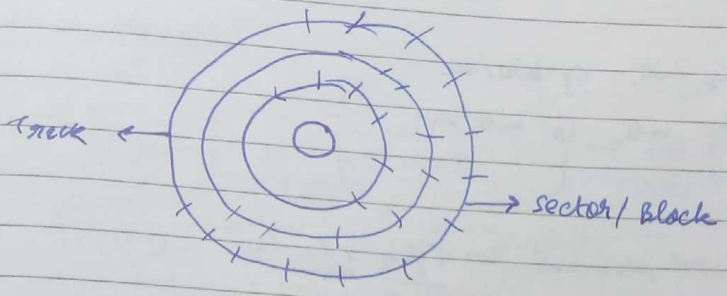
Data transfer rate (t_B)

Depends upon storage density along track and speed at which stored information is moving.

T = storage density (bits/cm)
V = velocity (cm/sec)
TV = Data transfer rate (bits/sec)
t_B = Time required to access a block of information.
N = track capacity (words)
r = Rate of revolution (revs/sec)
n = Number of words per block
rN = data rate of memory (words/sec)
$\frac{n}{rN}$ = time required to transfer 'n' words

t_s = seek time
Average latency time = $\frac{1}{2r}$

1 rev = 1 complete track

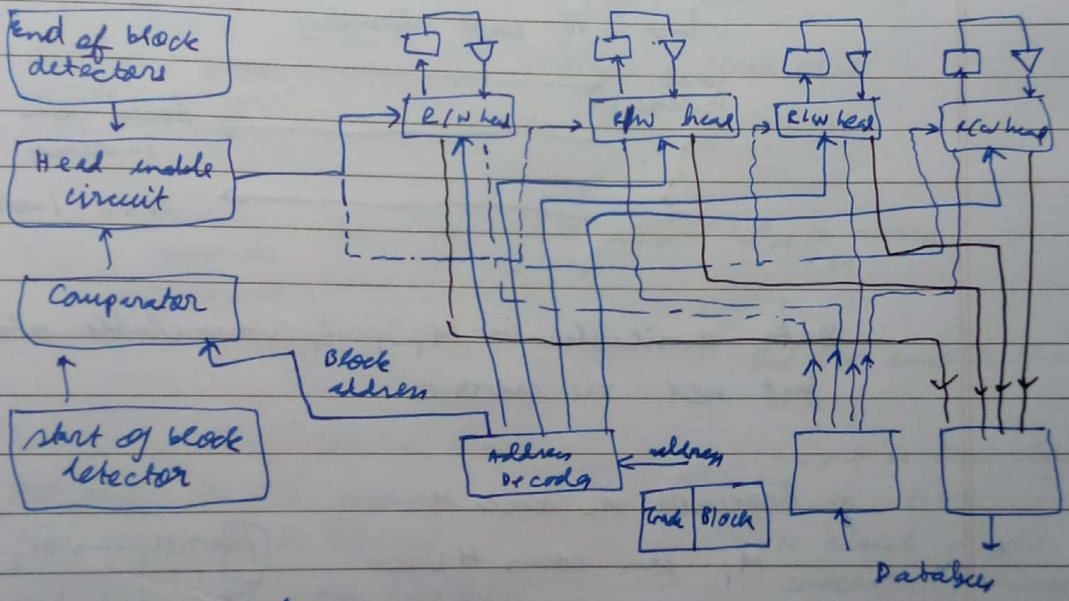


$$t_B = t_s + \frac{1}{2r} + \frac{n}{rN}$$

t_s → seek time
$\frac{1}{2r}$ → latency time
$\frac{n}{rN}$ → Data transfer time

NOTE:
Data transfer time ≠ Data transfer rate

Serial Access Memory Unit



Reasons of using virtual memory:

→ To free programmer from the need to carry out storage allocation and to permit efficient sharing of memory space among different users.

→ To make programs independent of the configuration and capacity of memory systems, used during their execution.

→ High access rate and low cost can be achieved with a memory hierarchy.

MM → S_1 bits, C_1 bits/sec

VM → S_2 bits, C_2 bits/sec

Average cost per bit = $\frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$

If $S_2 >> S_1$,

Average cost per bit $\cong C_2$

Memory hierarchy

$M_1, M_2, M_3, \ldots, M_{N-1}, M_N$

↳ N-level hierarchy

cost increases ←

Access time increases →

Size increases →

Data available in M_{i-1} is available also in M_i but not the reverse.

Processor can access M_1,

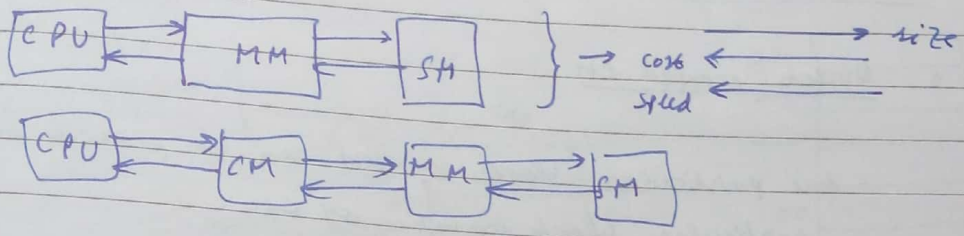
M_1 can access M_2

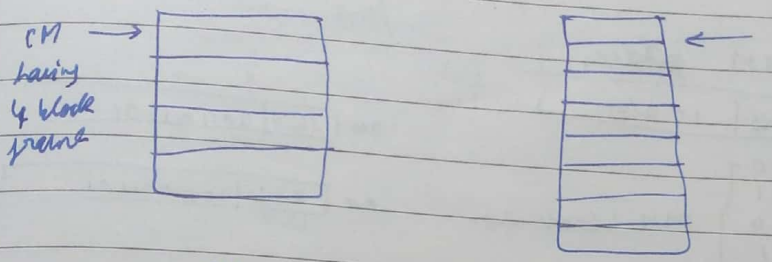
⋮

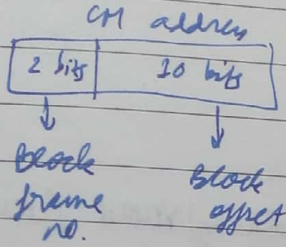
so on

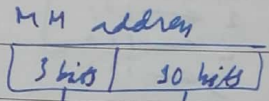
P → M_1 → M_2 → . . . → M_N

Cache Memory









Processor generates → converted to CM address → helps to access CM.

↓

But CM smaller than MM

∴ might be that data is in MM

∴ CM search might be successful / unsuccessful

↓ hit ↓ miss.

Processor generates 100 MM address.

Hit occurs for 80 accesses.

Miss occurs for 20 accesses.

H = Hit ratio = $\frac{80}{100}$ Miss Ratio = $\frac{20}{100}$

No. of times CM accessed = 100

t_c = access time of CM

t_m = access time of MM

t_A = average access time

Access time = $100\, t_c + 20\, t_m$

$t_A = t_c + \frac{t_m}{5}$

Access time = (No. of times CM accessed) × t_c + (No. of times MM accessed) × t_m

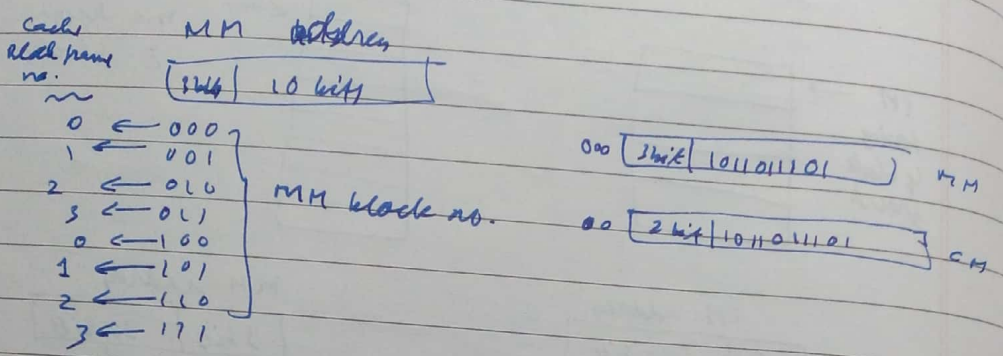
t_A = $\frac{\text{Access time}}{(\text{No. of times CM accessed})}$

$\therefore \boxed{t_A = t_c + (1-H)\, t_m}$

Address Mapping

1. Direct mapped CM

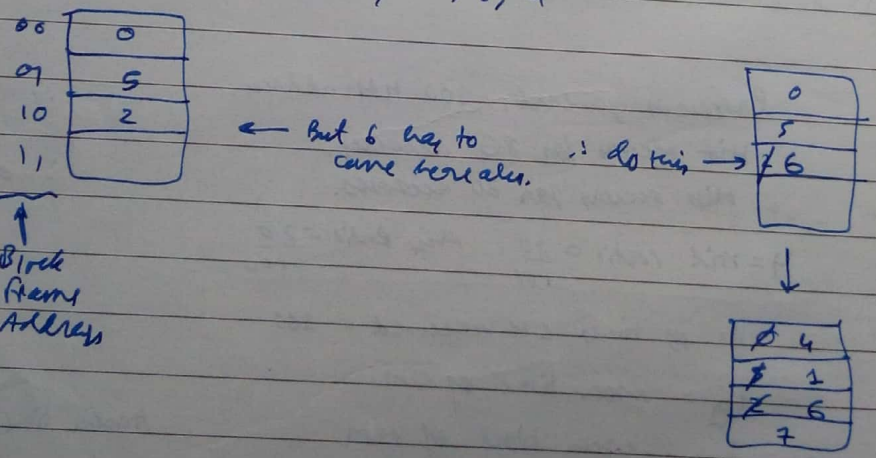
- One particular block of MM can be placed in one particular block frame of CM.



Block frame address = (Block address) Mod (No. of blocks in CM frames)

CPU generates request for the following blocks:

0, 2, 5, 6, 7, 1, 4



Initially CM empty

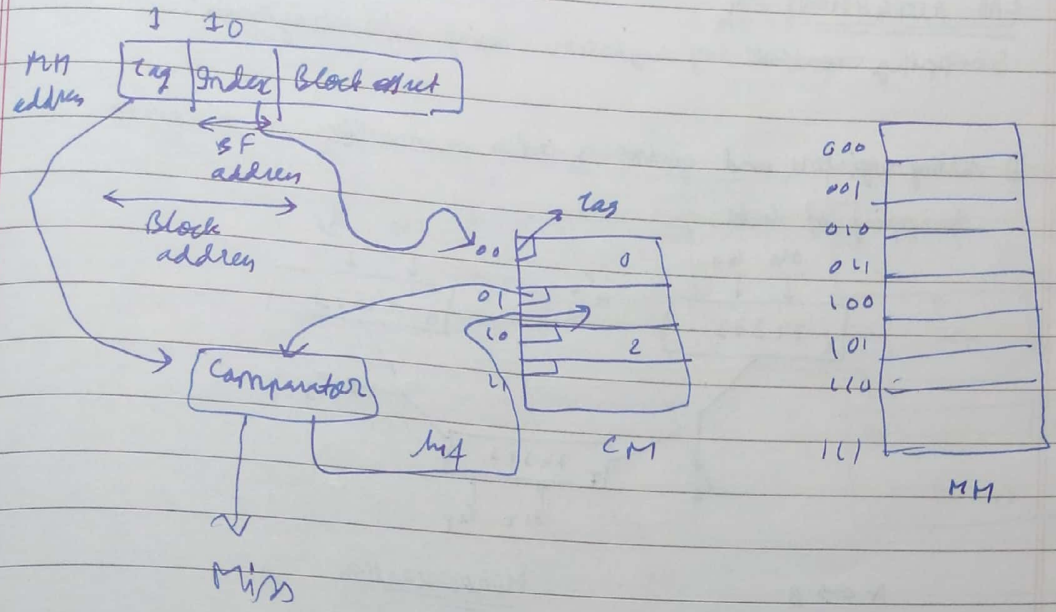
∴ 4 misses are compulsory. (for each BF addr)

Then 3 conflict misses also occur.

∴ total = 4 + 3 = 7 misses

∴ We get direct miss and conflicting miss.

∴ Direct mapping is bad.

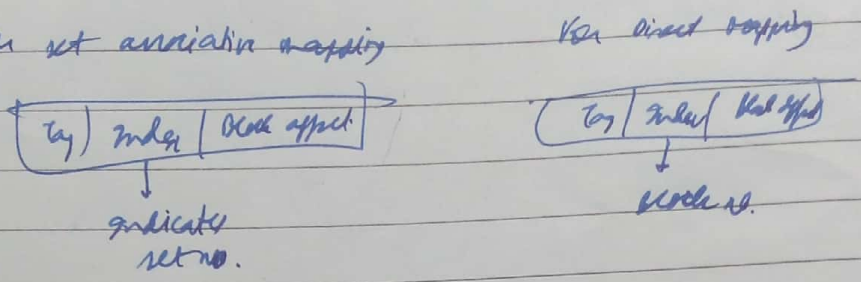

1 10
MM addrs
Tag
Index
Block offset
SF address
Block address
Tag
00
01
10
11
0
2
Comparator
Hit
CM
000
001
010
011
100
101
110
111
MM
Miss

2. Fully Associative Mapping

3. Set Associative Mapping (SAM)

Associativity = No of block frames in a set
[for direct mapping, associativity = 1]

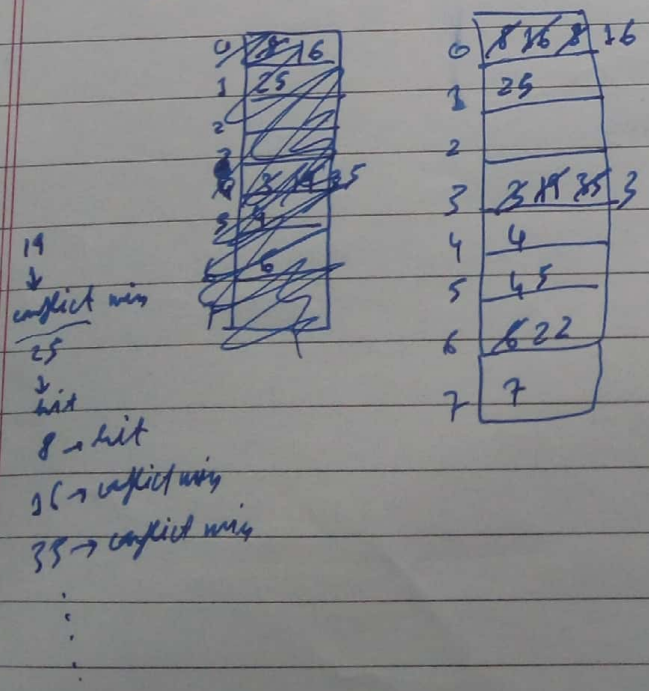
k-way set associative : A particular block can be placed in any of the k-block frames in a set.



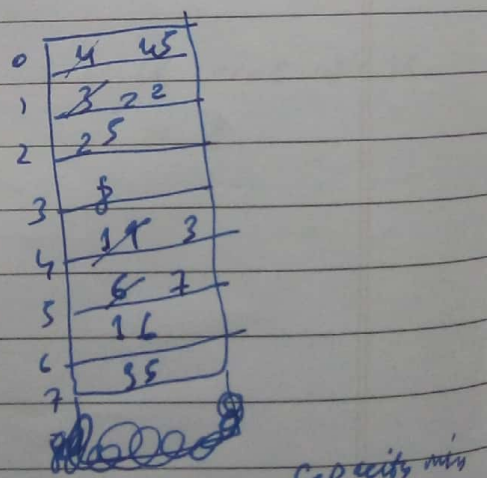
(Set No.) = (Block no.) mod (no. of sets in memory)

Eg. Sequence of memory block requests
4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 83, 16, 25, 7
CM contains 8 block frames.

For DM

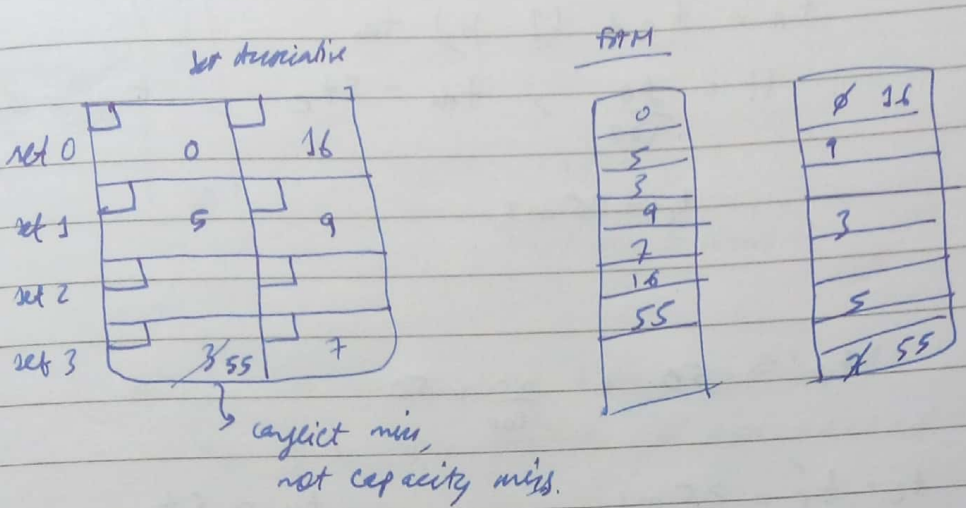


For FAM

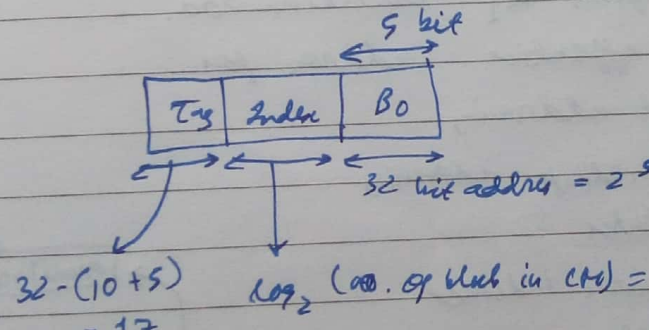


LRU: least recently used.

Eg. Two way set-associative memory with 4 sets and 8 cache block frame. 0-127 (128 blocks) in MM.
LRU used to replace cache block frame.
Sequence of memory block requests
→ 0, 5, 3, 9, 7, 0, 16, 55

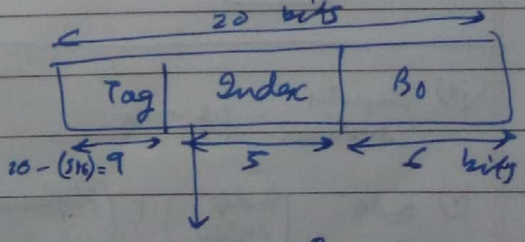


Eg. DM Cache of size 32 KB.
Block size = 32 bytes.
CPU generate 32 bit address.
Find no. of bits needed for cache indexing and tag bits



No. of CM block
$= \frac{32\ KB}{32\ byte} = 2^{10}$

Eg. 4-way set associative cache, consisting of 128 blocks.
CPU generate 20 bit address. Block size = 64 words.
Find no. of bits in tag, index and offset field.



No. of CM block = 128
CM block size = 64 word
MM addrs = 20 bit
4 way SPM.

No. of sets in CM $= \frac{128}{4} = 32$
Size of index = $\log_2$ (total no. of sets in CM)

Eg. A 2-level memory consisting of MM and CM.
CM is 5 times faster than MM. Hit ratio = 20%.
If average access time is increased by 20%
from 50ns, what is approx. change in hit ratio?

$$t_A = t_c + (1 - H)\, t_m$$

$$H = \frac{80}{100}, \quad t_m = 5t_c, \quad t_A = 50$$

$$\therefore t_c = 25\,ms$$

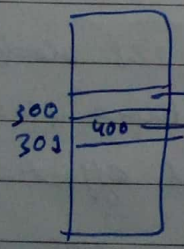
$$t_A' = 50 + \frac{20}{100} \times 50$$

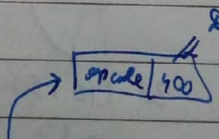
$$t_c = t_c' = 25\,ms \qquad t_m = 5t_c$$

$$H = ?\ (\text{solve})$$

Eg. An instruction is stored at location 300 with its address field at location 301. Address field value is 400. processor register R_1 contains 200. Evaluate effective address for

(i) Direct addressing
(ii) Immediate addressing
(iii) Relative
(iv) Register indirect
(v) Index





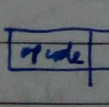
Effective address (EA) = address of operand.

(i) Direct : EA = 400

(ii) Immediate : EA = 301 → opcode | 400 → operand = 400

(iii) Relative :

Read from 300 → PC = 300
Read from 301 → PC = 301
∴ Now → PC = 302
Displacement = 400
EA = 302 + 400 = 702

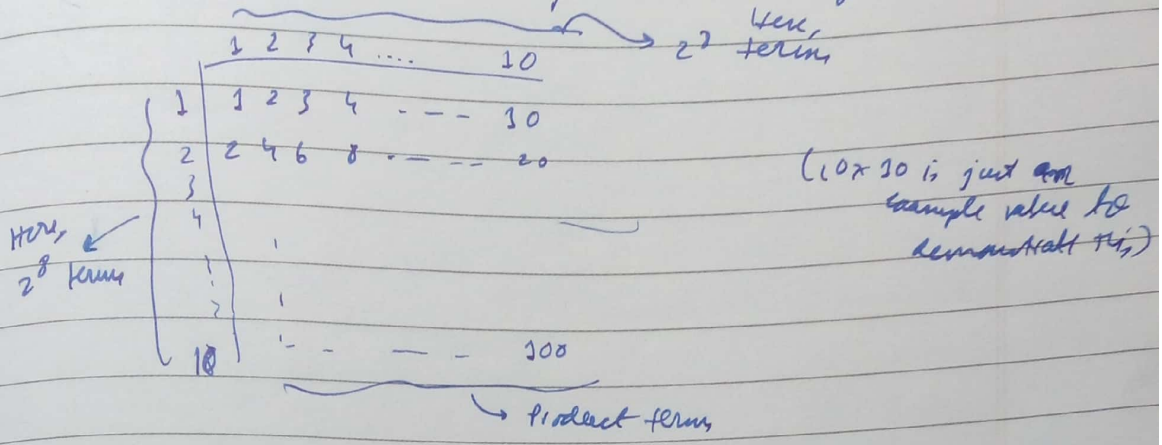


Displacement is set and base register (if not specified, we add program counter)

(iv) Register Indirect : EA = 200

(v) Index : (Index register content) + (Displacement) = EA
200 + 400 = 600

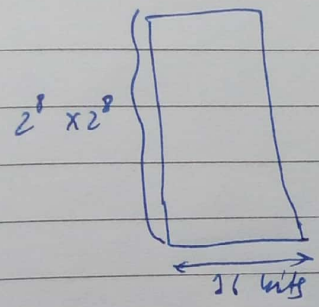
Eg. What should be size of ROM to store the multiplication table of 2 8-bit unsigned integer?



(10×10 is just an example value to demonstrate this.)

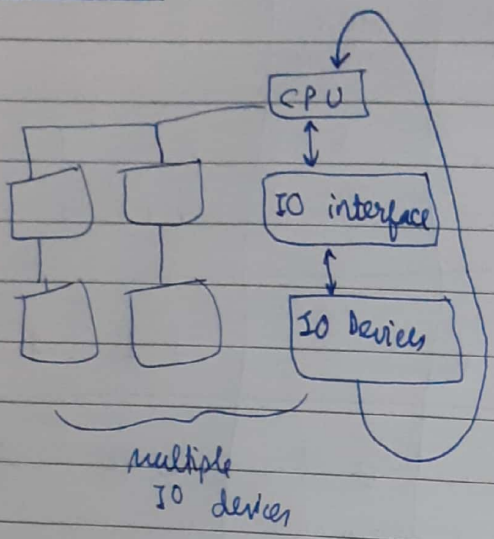
Here, # of product terms = $2^8 \times 2^8$

No. of memory location in ROM needed = $2^8 \times 2^8$

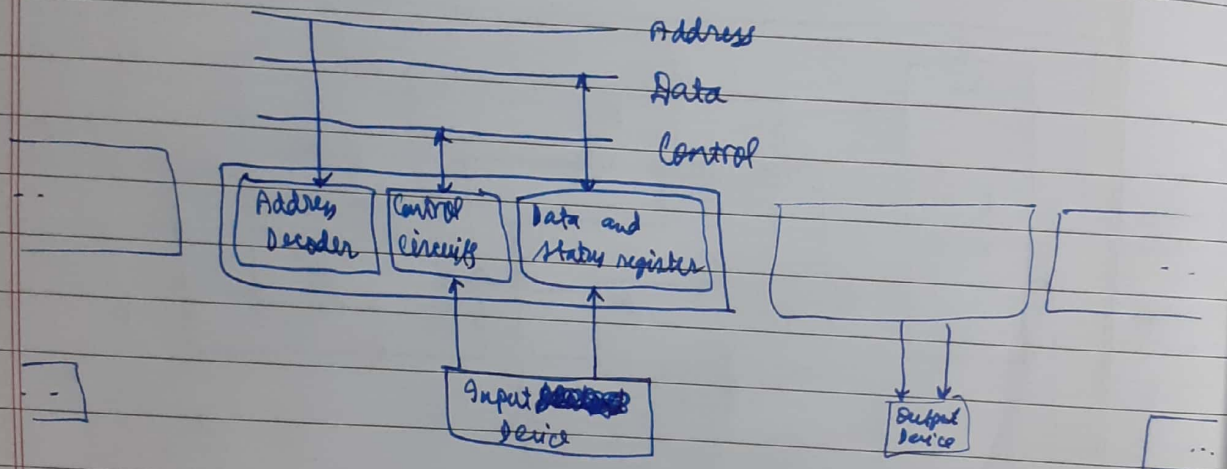


∴ ROM size = $2^8 \times 2^8 \times 16$

Peripherals (I/O devices)



Each IO device has its own address.



Status register

↓

status → whether transfer of data b/w IO device and processor is successful or not.

eg. of control signals

I/O R

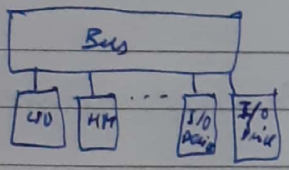
I/O W

Bus

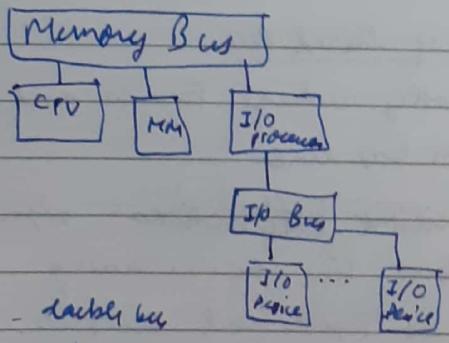
* Bus architecture
  - Bus interconnects several processor-level components. (MM, I/O processor, CPU, I/O Devices)

* Bus control
  control 2 things
  - Time of transferring data via bus
  - selection process by which a device can gain control of the bus.
    → Devices are divided into 2 categories:

Bus Master	Bus slave
- CPU - I/O processor	- MM - I/O devices
· Main bus control unit - supervise the use of the bus by other units	· Responds to command issued by bus master



- single bus
- no role of I/O processor



- cache bus
- I/O processor.

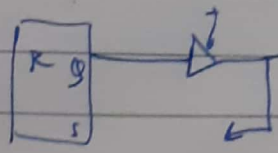
* Bus communication

Synchronous	Asynchronous
- Sending and receiving units work at same speed. This is possible if sender and receiver are close-by, as they can then use same clock pulse.	- "Handshaking signals" - Analog Digital conversion
- For larger separation, different clocks have to work at same frequency	- Reading and receiving and exchange handshaking signals, before starting exact communication.
- speed of operation = speed of clock	- speed of operation = speed of slowest device
	- Used for long-distance communication
	- More complex bus control circuit

* Shared Bus

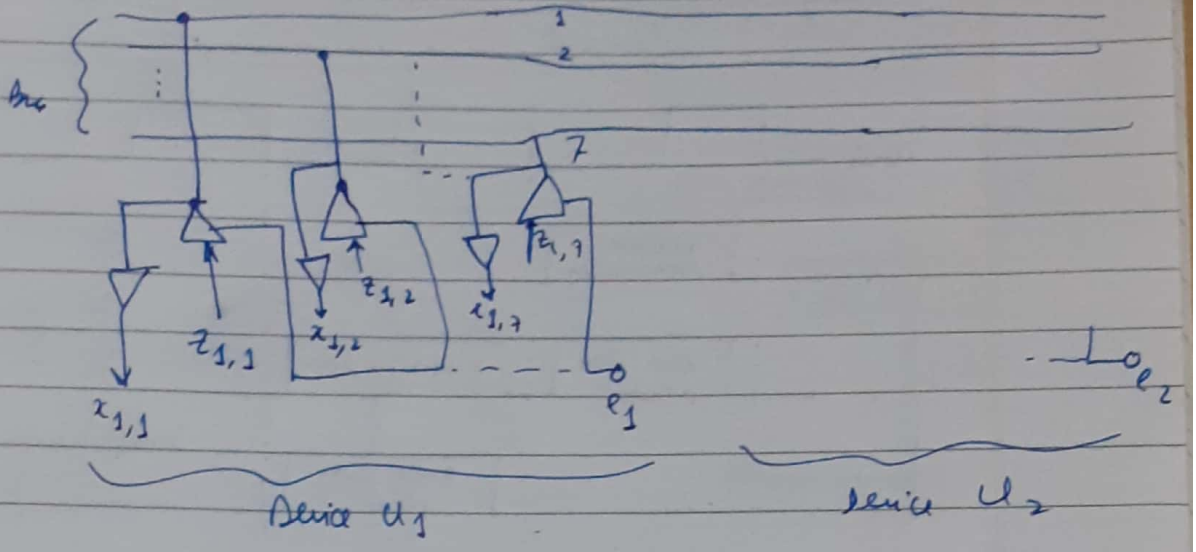
- Advantage of shared Bus.
  - → Greatly reduce Fan-in and Fan-out constraints on bus lines
  - → Facilitate bidirectional signal transfer over a bus line by allowing same line to be used as input and output.

eg.



Requirement ↓

Tristate used here provides similar function as shared bus.



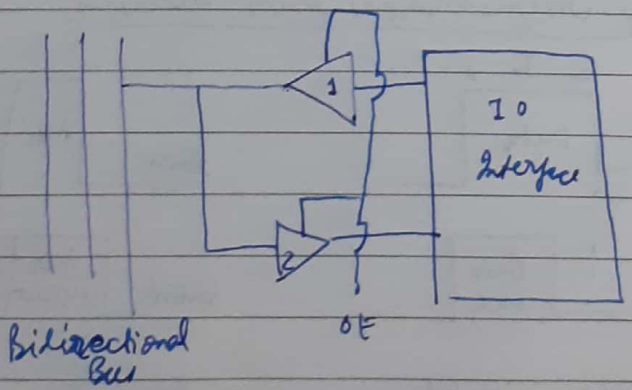
$[e_1 = 1 \quad e_2 = 0]$: $U_1 \rightarrow$ write
$U_2 \rightarrow$ read from buffer

$[e_1 = 0$ (Read), $e_2 = 1$ (write)$]$

$[e_1 = 1, \ e_2 = 1]$: Not allowed → can't write simultaneously

$[e_1 = 0, \ e_2 = 0]$: Allowed → can read simultaneously.

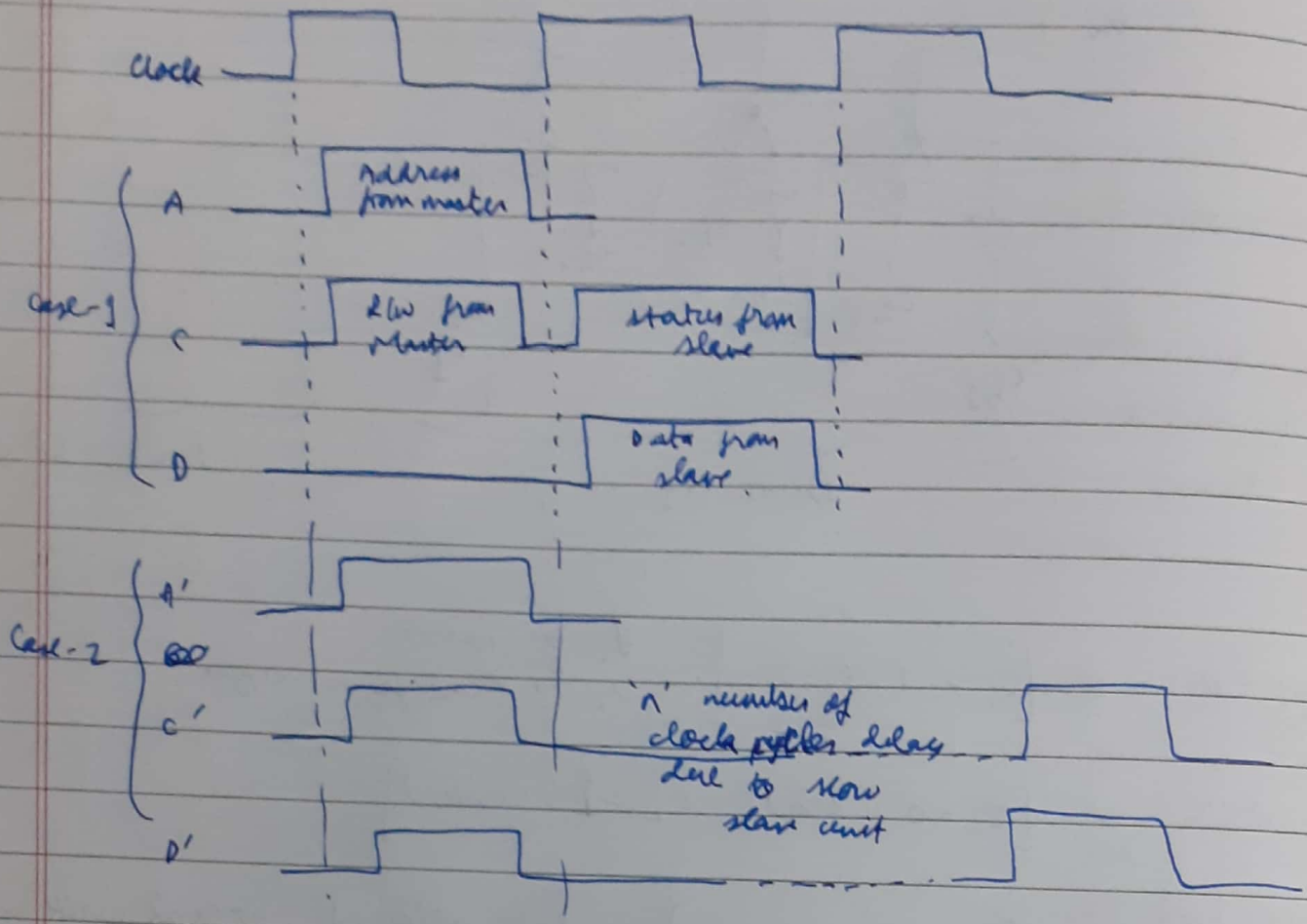
* Transceiver



1 = Driver (OE = 1)
2 = receiver (OE = 0)

Driver drives the bus while maintaining its output level within specified limit

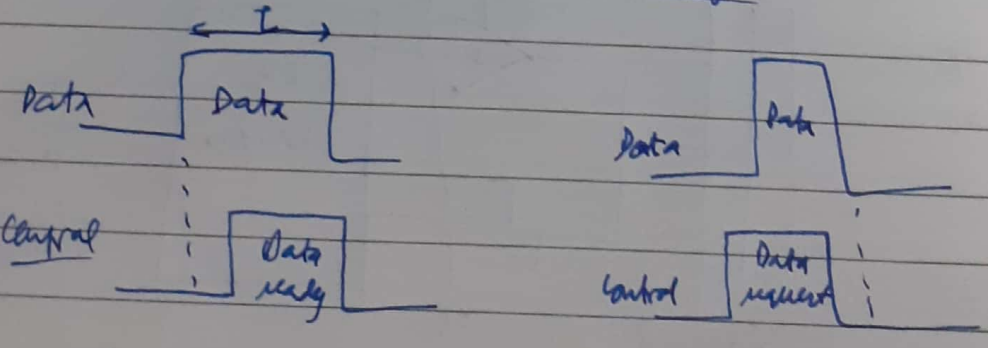
* Timing diagram for synchronous data transfer



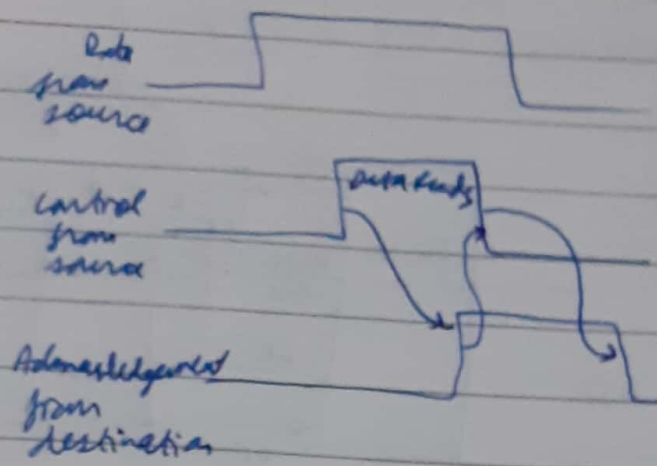
* Asynchronous Data transfer

- Used by devices having widely different data rates.

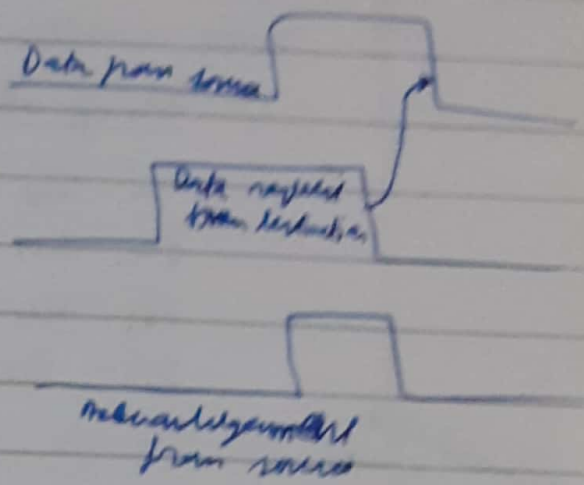
→ One-way Asynchronous Data Transfer.



→ 2 - way Asynchronous Data Transfer



Source initiated data transfer



Destination initiated data transfer.

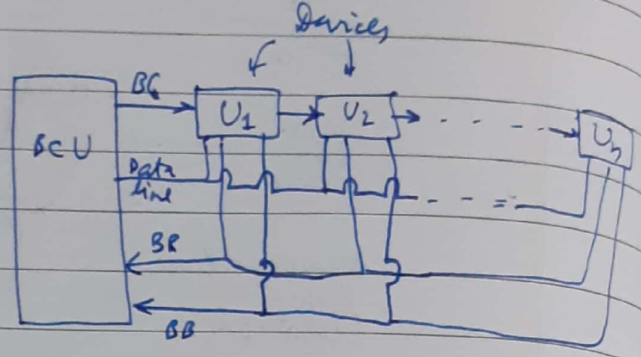
Bus arbitration

Arbitration is the process where
↳ Next device to became bus master is selected

- Three arbitration schemes

1) Daisy chaining

BR: Bus Request
BG: Bus Grant
BB: Bus Busy
BCU: Bus Control Unit

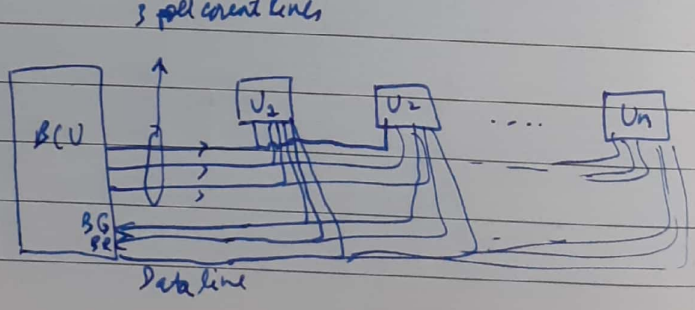


* Control lines = 2 min

Advantages:
- Needs very simple arbitration algorithm
- Can be used with essentially unlimited number of bus units.
- It has 2 control lines (other than BB)

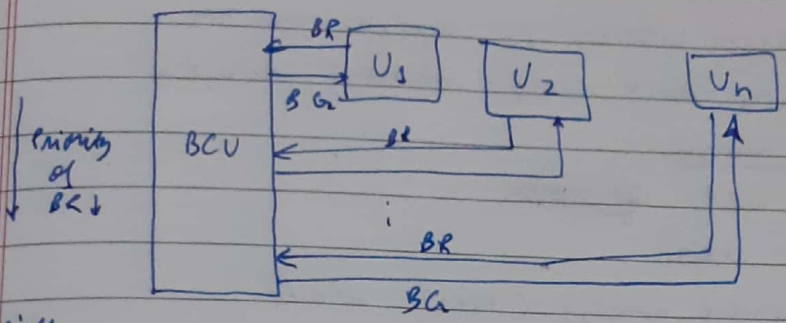
Difficulty:
- Failure of 1 bus unit affects the entire scheme.
- Priority is decided in term of location of device w.r.t BCU.
- Slow. (BG for U_n has to pass from $U_1 \to U_2 \to U_3 \to \dots \to U_n$)

2) Polling



"Polling sequence" → eg. 5, 4, 3, 1, 2, 6
↳ 5 has most priority

3) Independent Requesting

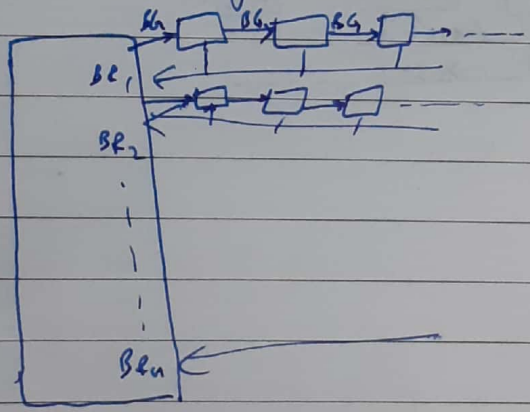


control line = 2n

No. of control lines = 2 × (No. of devices)

4) Hybrid system

- combination of 1 and 3



m - BR lines
n - devices
$m < n$

priority: $BR_1 > BR_2 > BR_3 > \dots$
∴ higher priority devices should be connected with higher priority BR. lines.

BG lines are arranged as in daisy chaining.

I/O operation

↓

Data transfer between IO devices and MM.

↳ IO device can't access memory directly

Mode of Data transfer:

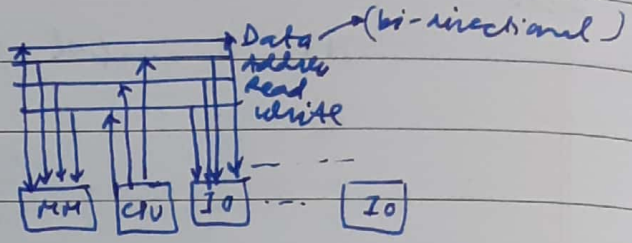
→ Programmed mode

→ Interrupt mode

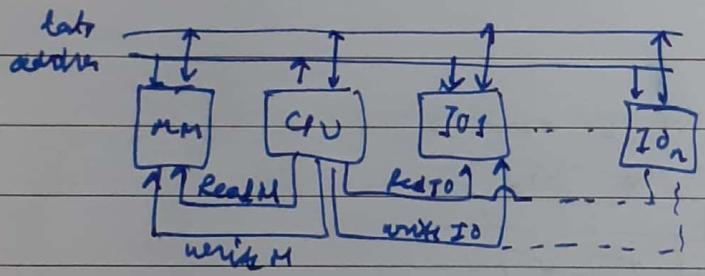
→ Direct memory access (DMA) mode

Programmed Mode

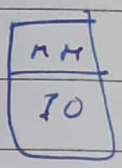
→ Memory mapped IO



→ IO mapped IO



Address space is shared [∵ MM size ↓]



- If address from MM part, memory is referred otherwise, IO is referred.

After this, absent for 1 class (6 Pages)

classmate
Date
Page

Date
Page

RAID

- Redundant array of inexpensive Disks
  - ↳ storage system based on multiple disks
  - → 7 different configurations (or RAID levels)
    - ↳ RAID-0 to RAID-6
      - ↳ improving speed.

RAID-1 (Mirroring)

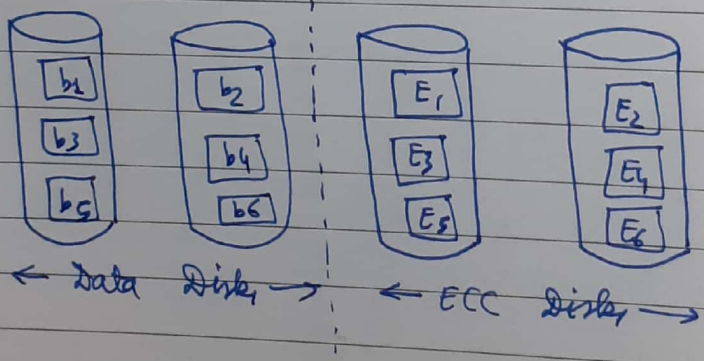
N = number of data disks

N = number of redundant disks

2N

RAID-2

- strips the bits of a file across the disks.

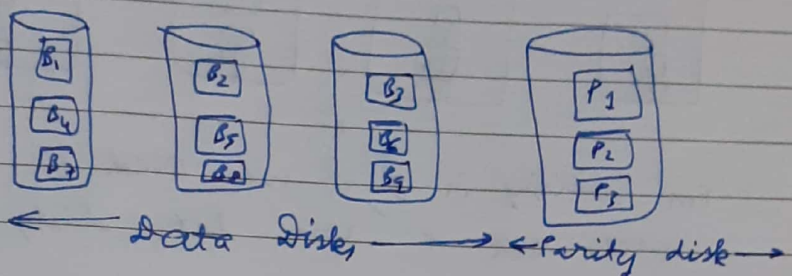


ECC: Error-correcting Code

- uses lot of disk
- need to maintain synchronization of all these disks, which is expensive.
- random reading is fair, good for sequential reading and fair for sequential writing.
- single-bit error can be detected and corrected.
- multiple errors can be detected, but cannot be corrected.

RAID-3

- Strips the bytes of a file across the disks.

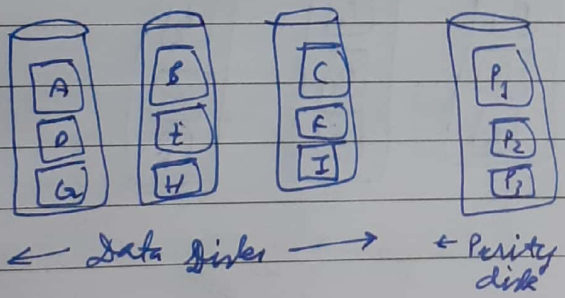


N+1 disks,

- high read speed
- low write speed
- If 2 disks are damaged, data cannot be retrieved simultaneously

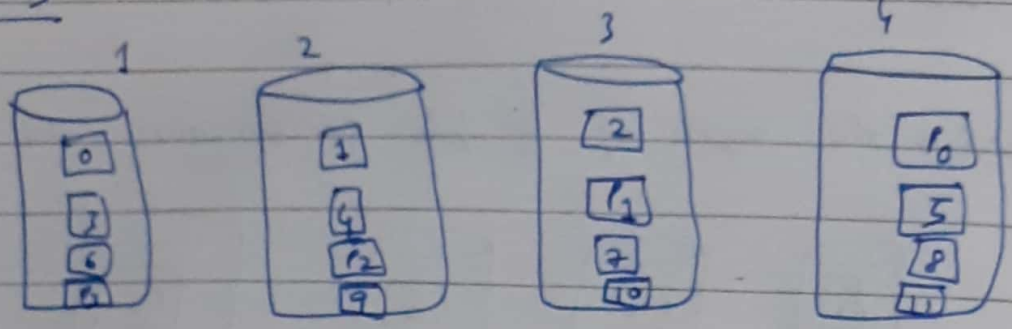
RAID-4

- Uses Block-level stripping



- Needs hardware support for parity calculation
- Write operation is slower due to parity calculation
- Good for sequential read/write data.
- Good random read, bad random write.
- For each I/O write, calculation of parity block is needed.

RAID - 5

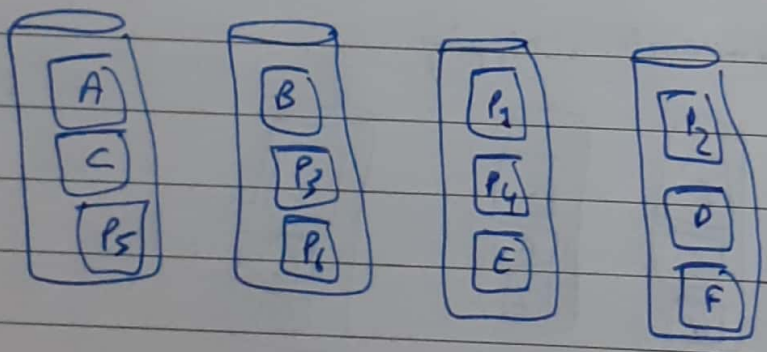


- For writing in 3, we need to access 1st & 3rd data disks.

- For writing in 8, we need to access 4 and 2.

- All the disks can be accessed simultaneously.
- But block 1 and 3 cannot be accessed simultaneously.

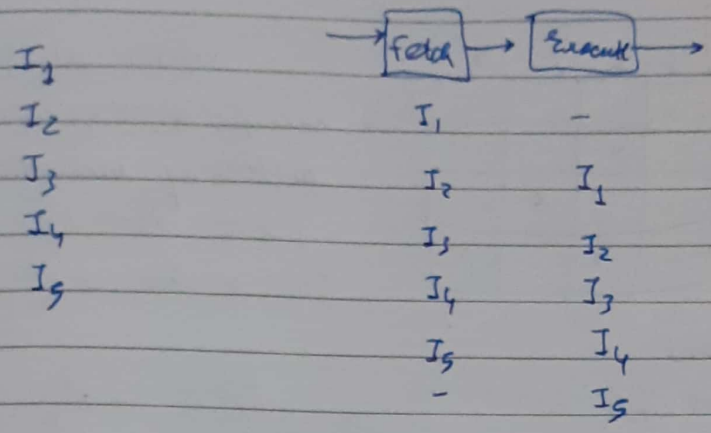
RAID - 6

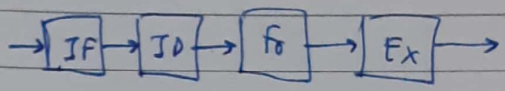


-

-

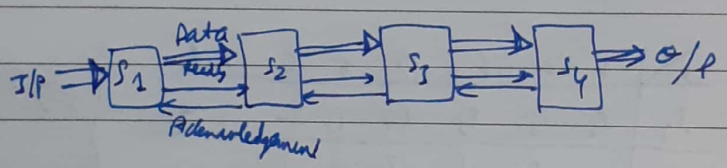
Pipelining



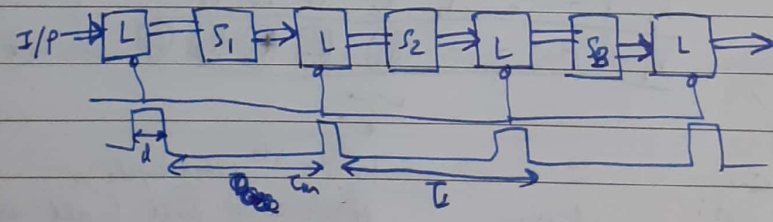


*Linear pipeline processor

- cascade of stages which are linearly connected to perform a fixed function over a string of data flowing from one end to the other.



Asynchronous model (4-segment pipeline)



Synchronous model (3-segment pipeline)

L → clocked master-slave flip-flop.

S_1, S_2, S_3 → combinational circuits

f = clock frequency = $\frac{1}{\tau}$

d = latch delay

τ = clock time

$\tau = d + \tau_m$

$\tau_m = \max \{ t_1, t_2, t_3 \}$

t_1: time taken by S_1 for its task

t_2: time taken by S_2

t_3: time taken by S_3

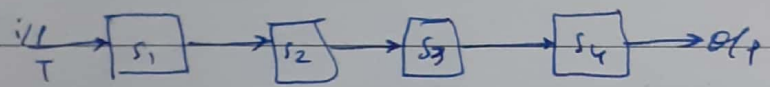
* Reservation table for a 4-segment pipeline:

S \ cc	1	2	3	4
S_1	×			
S_2		×		
S_3			×	
S_4				×

→ specifies the utilization pattern of successive stages

S = segment
cc = clock cycle

Non-pipeline:



non-overlapping segments in pipeline:

∴ 4 clock cycles needed for task T.

1 task, 4 clock cycle

S \ cc	1	2	3	4	5	6	7	8
S_1	T_1	T_2	T_3	T_4	T_5	–	–	
S_2	–	T_1	T_2	T_3	T_4	T_5	–	
S_3	–	–	T_1	T_2	T_3	T_4	T_5	–
S_4	–	–	–	T_1	T_2	T_3	T_4	T_5

⇒ Space-time Diagram

→ When a task reaches segment S_4, it is completed

in the 4th clock cycle, pipeline is full (all segments have some task)

Assuming there are no data dependencies

pipeline concept: 5 tasks, 4 segment pipeline,

8 clock cycle

No. of clock cycles needed = $k + (n-1)$

k → no. of segments
n → no. of tasks

$4 + (5-1) = 8$
↳ $4 + 1 + 1 + 1 + 1 = 8$

* Speedup of a k-segment pipeline (S_k)

clock time = τ

- Non-pipeline processor:
  - 1 task → $k\tau$
  - n tasks → $nk\tau$
- Pipeline-processor
  - n-tasks → $\{k + (n-1)\}\tau$

- $$S_k = \frac{[nk]\tau}{[k + (n-1)]\tau}$$
  - $S_{k\,max} = k$, for $n \to \infty$
  - $S_{k\,min} = 1$, for $n = 1$

* Efficiency of a k-segment pipeline (E_k)

- $$E_k = \frac{S_k}{k} = \frac{n}{k+n-1}$$
  - $E_{k\,max} = 1$, when $n \to \infty$
  - $E_{k\,min} = \frac{1}{k}$, when $n = 1$

* Throughput of a k-segment pipeline (H_k)

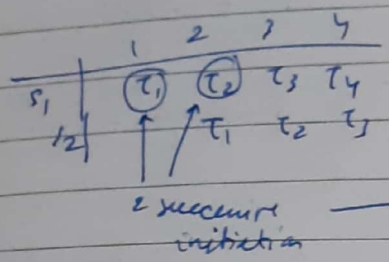
↳ output per unit time

- $[k + (n-1)]\tau$ time is required for evaluating n-tasks
- $$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{E_k}{\tau} = E_k \cdot f$$

Relation b/w H_k, E_k and S_k:

$$H_k = E_k \cdot f = \frac{S_k}{k\tau}$$

* Pipeline latency
↳ delay that occurs b/w 2 successive initiation in a pipeline



→ ∴ latency = 1 clock cycle

→ For static pipeline, pipeline latency = 1

* Pipeline Performance depends on k.

- If we increase ~~the number of segments~~, we feel that time needed ↓
- But k can't be increased to ∞.

- t = total time required for a ~~non~~pipeline sequential program to execute ~~the same~~ program on a k stage pipeline, with an equal flow of delay d.

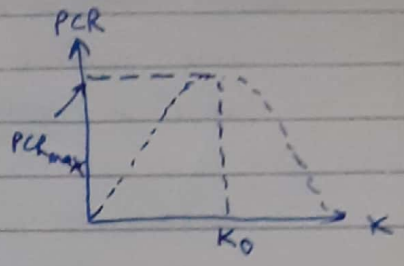
- clock period = $\left(\frac{t}{k}\right) + d$
  - $\frac{t}{k}$ ↳ time taken to execute a task per segment
  - d ↳ latch delay

- Throughput = $\frac{1}{\left(\frac{t}{k} + d\right)}$

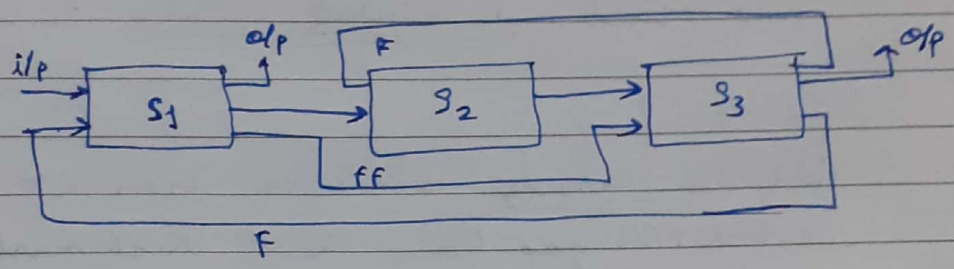
- Pipeline cost = $c + kh$
  - c ↓ cost of logic stages
  - kh → cost of each latch

- performance cost ratio = PCR = $\frac{\text{Throughput}}{\text{cost}} = \frac{1}{\left(\frac{t}{k}+d\right)(ct+th)}$

K_0 = optimal cost = $\sqrt{\frac{tc}{dh}}$



Nonlinear pipeline



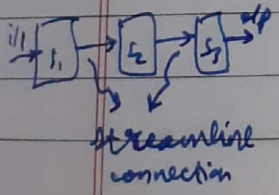
Dynamic pipeline

F : Feedback
f : Feedforward

S \ CC	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X	.			
S_3			X		X		X	

RT for function x

Static:



S \ CC	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

RT for function y

- Static pipeline can evaluate a single function, whereas a dynamic pipeline can evaluate multiple functions using the same pipeline

- In a static pipeline, it is easy to partition a given function into a sequence of linearly ordered sub-functions which is not possible in dynamic pipeline due to presence of f and ff connections

* Latency Analysis:

- Any attempt by 2 or more initiations to use the same pipeline stage at the same time would cause a collision.

- Latency for which collision occurs is called forbidden latency

- Latency for which there is no collision is called permissible latency.

- Forbidden latency calculation from RT:
  → Time gap b/w the 2 check mark in the same row in the reservation table

$FL_X = 5, 2, 7, 4$
$PL_X = 1, 3, 6, 8$
} for function X (see previous page)

$FL_Y = 2, 4$

Diagram to show that if latency = 2, collision would occur for function X.

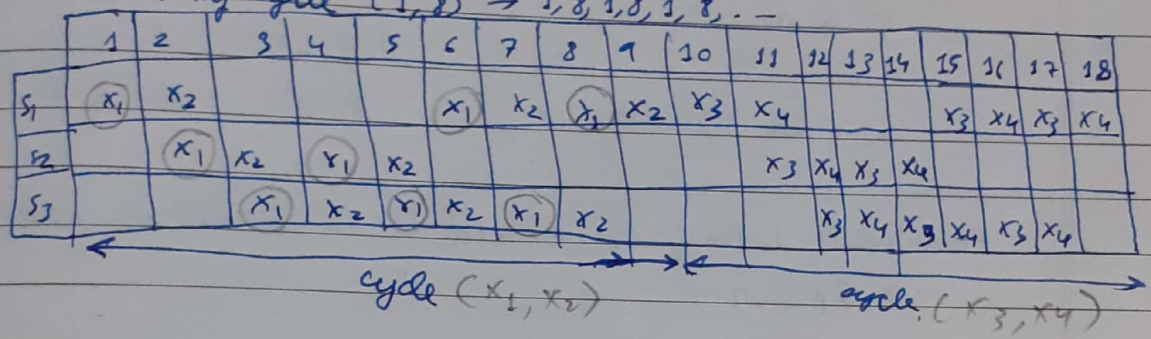
CC / S	1	2	3	4	5	6	7	8	9	10
S_1	X_1		X_2		X_3	X_1		X_1X_2		X_2
S_2		X_1		X_1X_2		X_2X_3				
S_3			X_1		X_1X_3		X_1X_2 X_3		X_2	

* **Latency sequence:**
  - A sequence of permissible non-forbidden latencies b/w successive task initiation

* **Latency cycle:**
  - Latency sequence which repeats the same subsequence (cycle) indefinitely.

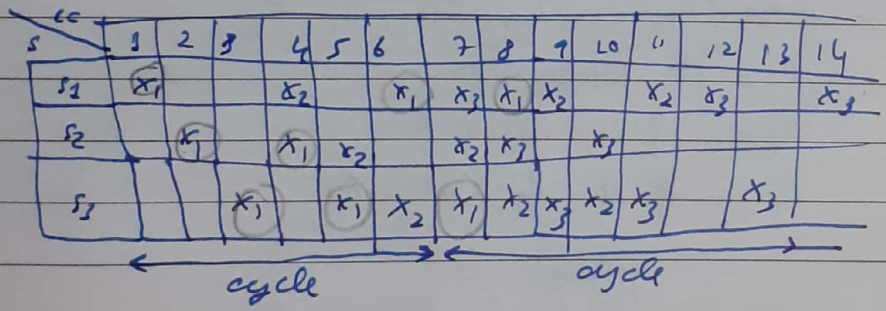
For function X

Latency cycle (1, 8) → 1, 8, 1, 8, 1, 8, . .



Average latency = $\frac{1+8}{2} = 4.5$

Latency cycle (3) ⟶ 3, 3, 3, ...



Constant latency = 3

Collision-free scheduling

When scheduling events in a pipeline, the main objective is to obtain the shortest average latency b/w initiations without causing collisions.

Collision-vector

For a reservation table having 'n' columns
Max Forbidden latency = m
$m \leq n-1$

eg. for function x, $n = 8$, $m \leq 7$

Permissible latency (p) should be as small as possible
$1 \leq p \leq m-1$

eg for function x, $1 \leq p \leq 6$

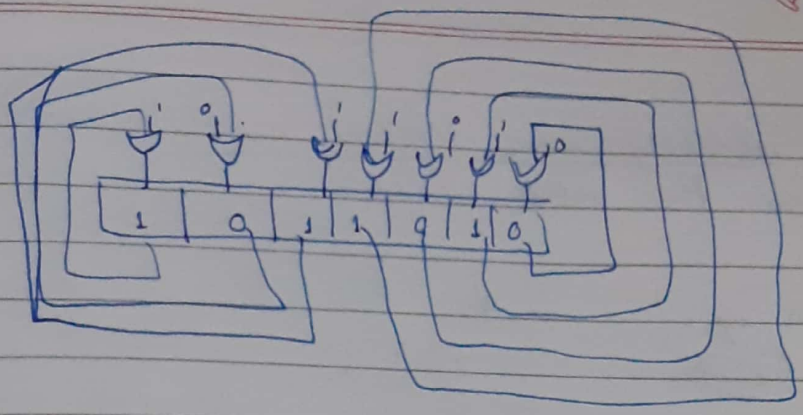
$p = 1 \rightarrow$ static pipeling

collision vector (CV) = $C_m C_{m-1} \ldots C_1$
C_i = i^{th} bit in CV
$C_i = 1$ if i is a FL
$C_i = 0$ if i is not a FL.

eg. for X,

C_7	C_6	C_5	C_4	C_3	C_2	C_1
1	0	1	1	0	1	0

for Y, CV = 1010



1 unsafe
0 safe

0 emerges for PL
∴ safe

1 emerges for PL
∴ unsafe.

1 0 1 0 1 0

→ 0 1 1 1 0 1 → 0 } 1st shift

→ 0 0 1 0 1 1 1 0 → 1 } 2nd shift

1 0 1 1 0 1 0
+ 0 1 0 1 1 0 1
1 1 1 1 1 1 1

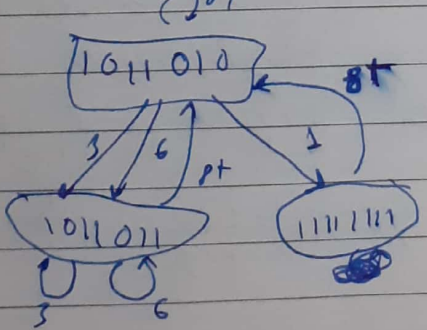
→ in register

1 0 1 1 0 1 0

↳ right 3 times to set right

0 0 0 1 0 1 1

1 0 1 1 0 1 0
0 0 0 1 0 1 1
1 0 1 1 0 1 1



state diagram for function X

↓

users all possible state transitions that avoid collision.

Min latency = 3.
(average)

Only infinitely many latency cycles, are on trace from the state diagram.

For eg. : (1,8)
(1,8,6,8)
(3,8)
(6,8)
(3,8)
(3,6,3), ...

Simple cycle is a latency cycle in which each state appears only once.

↳ eg. (1,8), (3), (6), (3,8), (8), (6,8)
4.5 3 6 5.5 8 7

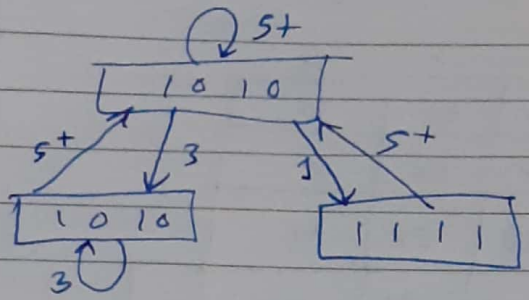
Not simple cycle
↳ eg. (1,8,6,8) → traverse 10 1, 8, 6, 10 twice

Simple cycle → Average latency (eg. 4.5, 5.5, 7)
→ Constant latency (eg. 3, 6, 8)

Greedy cycle = simple cycle with average latencies lower than other simple cycles.
eg. (1,8) and (3) are greedy
4.5 3
↳ least avg. ↳ least constant

Minimum average latency = 3

4. State diagram for 4.



{1, 5} and {3} are greedy cycles

CPI
↳ clocks per instruction

MIPS
↳ million of instruction per second
→ helps to find out about speed of processor.

CPI

I_c = Number of instructions in a program.
T = CPU time to execute I_c no. of instruction
J = clock time

$$T = I_c \times CPI \times J$$

MIPS

C = total no. of clock cycles required to execute I_c no. of instructions

$$CPI = \frac{C}{I_c}$$

I_c: Total no. of instructions
T: Time to execute I_c no. of instructions.

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6}$$

$$= \frac{I_c}{I_c \times CPI \times \tau \times 10^6}$$

$$\text{Instructions per second} = \frac{I_c}{T}$$

$$= \frac{1}{CPI \times \tau \times 10^6}$$

$$= \frac{f}{CPI \times 10^6}$$

$$= \frac{f \, I_c}{C \times 10^6}$$

Pipeline in superscalar processor

- use multiple independent instruction pipeline having multiple stages.
- fetches multiple instructions at a time. Then attempts nearby to find independent instructions, that are independent of one another and can therefore be executed in parallel

clock cycle →	1	2	3	4	
	IF	ID	EX	W	
	IF	ID	EX	W	
	IF	ID	EX	W	
		IF	ID	EX	W
		IF	ID	EX	W
		IG	ID	EX	W

IF = Instruction fetch
ID = Instruction Decode
EX = Execute
W = register write-back

superscalar processor of degree m
m = no. of pipelines in the processor

Here m = 3.

Consider m = 2, N = no. of tasks = 4

cc →	1	2	3	4	5
S1	T_1	T_3			
S1	T_2	T_4			
S2		T_1	T_3		
S2		T_2	T_4		
S3			T_1	T_3	
S3			T_2	T_4	
S4				T_1	T_3
S4				T_2	T_4

5 clock cycles required

Assume:
All tasks are independent (for this semester)

S_i : segment of pipeline

for base processor, $m = 2$, 4 tasks

	1	2	3	4	5	6	7
S_1	T_1	T_2	T_3	T_4			
S_2		T_1	T_2	T_3	T_4		
S_3			T_1	T_2	T_3	T_4	
S_4				T_1	T_2	T_3	T_4

No. of clock cycles = 7

If k = no. of segments
N = no. of tasks
m = degree of processor.

No. of clock cycles
- for Base processor → $k + 1.(N-1)$
- for superpipeline processor → $k + \frac{1}{m}(N-m)$

Speedup

$T(1,1)$ → 1 instruction in 1 clock cycle
→ $CPI = 1$

$T(m,1)$ → m instruction in 1 clock cycle
→ $CPI = \frac{1}{m}$

$T(m,1) = k + \frac{1}{m}(N-m)$

$T(1,1) = k + (N-1)$

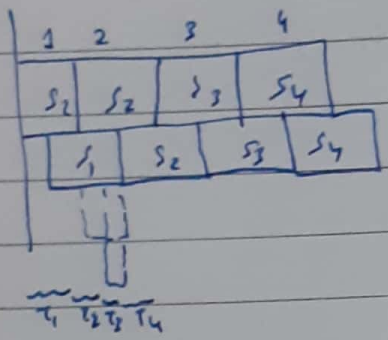
$S(m,1) = \frac{T(1,1)}{T(m,1)}$

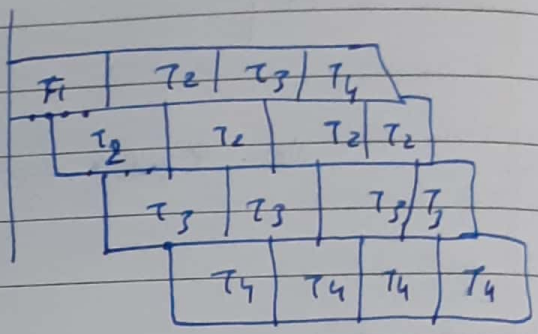
$$S(m,1) = \frac{k+N-1}{k+\frac{1}{m}(N-m)} = \frac{m(k+N-1)}{mk+N-m}$$

$N \to \infty$, $S(m,1) \to m$

Pipeline in superpipeline processor

Exploits the fact that many pipeline stage perform tasks that require less than half a clock cycle.





$CPI = \frac{1}{m}$

degree = m

$\Rightarrow \left(\frac{1}{m}\right)$ part of a clock cycle required for completing a task

$\tau(1, m)$ ⟶ 1 instruction is executed in $\frac{1}{m}$ clock cycle for superscalar processor of degree m.

$$\tau(1, m) = k + \frac{1}{m}(N-1)$$

$$S(1, m) = \frac{k + N - 1}{k + \frac{1}{m}(N-1)} = \frac{m(k + N - 1)}{mk + N - 1}$$

RISC

- Relatively few instructions & few addressing modes
- Memory access related to load and store instructions
- All operations done within CPU register
- Fixed length, easily decoded instruction format.
- Hardwired control (not microprogrammed)

CISC

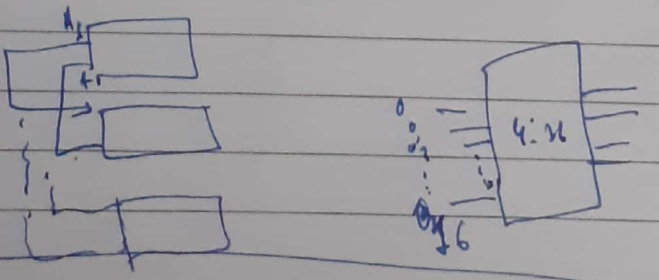
- Large no of instructions typically from 100 to 250.
- Some instructions that perform specialized tasks are used infrequently.
- Large variety of addressing modes
- Variable length instruction format.

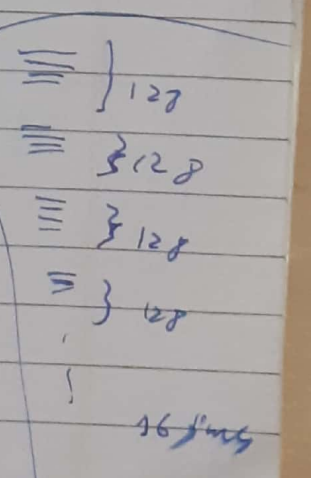
Q) How many 128 x 8 RAM chips are needed to design a memory of capacity 2048 bytes? What is the size of the address bus to access 2048 byte of memory? How many of these lines will be common to all chips? How many line must be decoded for chip select? Specify size of decoder.

Required memory → 2048 bytes = 2048 x 8
Available memory → 128 bytes = 128 x 8

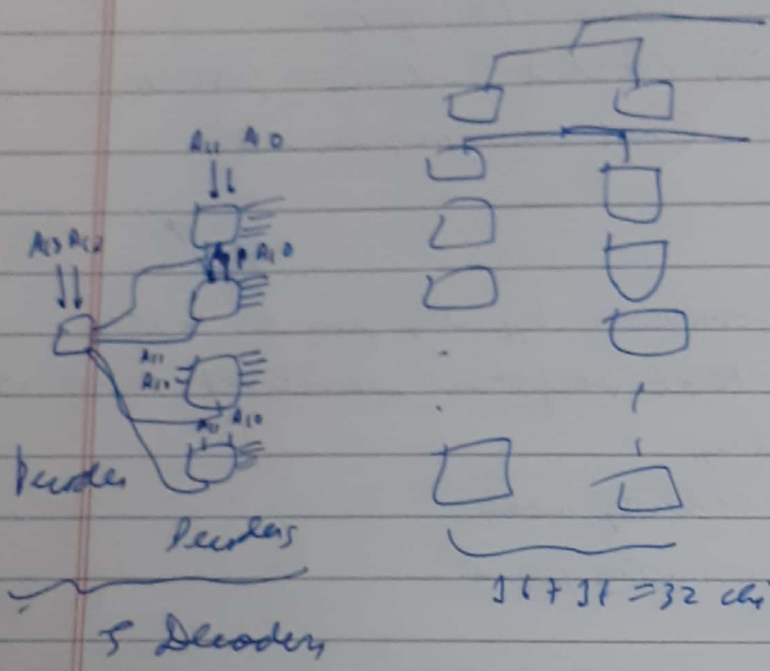
∴ No. of chips = 16
No. of line in address bus = 11

Decoder has chip-select role





Q) Each chip capacity 1K×8
How many 2×4 decoders are required to generate enable for constructing 16K×16 RAM from 1K×1 ?



$A_0 - A_9$: common to all

$A_{10} - A_{11}$: inputs to 4 decoder

$A_{12} - A_{13}$: enable decoder

A_{13}	A_{12}	A_{11}	A_{10}	A_9	...	A_0	
0	0	0	0	0		0	1024 address per chip 1
0	0	0	0	0		1	
0	0	0	0	1		0	
0	0	0	0	1		1	
0	0	0	1	0		0	1024 address per chip 2
0	0	0	1	1		1	

chip 16

8) A magnetic disk has the following specification.

No. of tracks = 1024

No of sector per track = 512

No. of bytes per sector = 512

Disc rotational speed = 7200 rpm

Seek time = 9 ms

Find data transfer rate and average access time

7200 → 60 secs

1 rotation → $\frac{1}{120}$ secs

1 rotation = 1 track access

(512 × 512) bytes = 1 track size.

∴ 512 × 512 bytes can be accessed in $\frac{1}{120}$ sec

No. of bytes accessed / sec = 512 × 512 × 120

Avg access time = seek time + latency time + Data transfer time

$= (9 \times 10^{-3})s + \left(\frac{1}{2} \cdot \frac{1}{120}\right) + \left(\begin{array}{l}\text{Time to access} \\ \text{data from 1} \\ \text{sector}\end{array}\right)$

(1 sector ⇒ 512 bytes)

Time required to access 512 bytes $= \frac{512}{512 \times 512 \times 120}$

∴ Avg access time $= \frac{9}{1000} + \frac{1}{240} + \frac{1}{512 \times 120}$

Q) Pipeline → speedup factor = 10
→ operating efficiency = 80%
No. of stages in the pipeline = ?

$$E_k = \frac{S_k}{K}$$

$$\frac{80}{100} = \frac{10}{K} \Rightarrow K = \left\lceil \frac{100}{8} \right\rceil = 13$$

Q) A non-pipeline system takes 50 ns to process a task.
Same task can be processed in a 6-segment pipeline with clock cycle = 10 ns.
Find speed up ratio of pipeline for 100 tasks.

$$\text{speed up} = \frac{50 \text{ ns}}{[\{6 + (100-1)\} \times 10] \text{ ns}} = \frac{5}{105}$$

Q) A processor takes 12 cycles to complete an instruction execution. The corresponding pipeline processor uses 6 stages with the execution times of 3, 2, 5, 4, 6 and 2 cycles respectively. What is the speedup that a very large no. of instructions are to be executed?

$$\text{speed up} = \frac{12}{\max\{2,3,4,5,6\}} = 2$$

Q) Pipeline with 2 stages - each stage takes 50 ns. (multiplication, addition). How much time is required to complete following code segment?

for i = 1 to 100 do A[i] = B[i] * C[i] + D[i]

$$\underbrace{\{k + (n-1)\}}_{\text{no. of clock cycles required}} \tau = \{2 + (100-1)\} 10 = 1010 \text{ ns}.$$

5) A pipeline has 4 stages with time delay 60, 50, 70, 80 ns

Interface latch delay = 10 ns
Find clock frequency of the pipeline?

cycle time = $[\max\{50, 60, 90, 80\} + 10]$ ns
= 100 ns

clock frequency = $\frac{1}{100}$ GHz = 10 MHz

6) A 4 stage pipeline has stage delay 150, 120, 160, 140 ns respectively. Registers that are used b/w the stages have a delay of 5 ns each.
Find total time taken to process 1000 data items on this pipeline.

For 1st data item

$150 + 5 + 120 + 5 + 160 + 5 = 585$ ns

$T_{m+d} = 165$ ns (5 ↗)

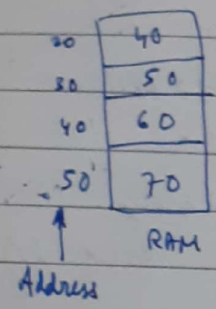
$\max\{150, 120, 160, 140\}$

For all the other items, 165 ns time required.

$585 + (1000 - 1) \times 165$ ns

every operand field of an instruction is associated with some piece of data. In order to execute the instruction, the processor needs the current value of data, which can be specified in several ways.

↑ Addressing mode

Q) 

Instruction: (i) Load AC
(i) load immediate 20
(ii) load direct 20
(iii) Load indirect 20
(iv) Load immediate 30
(v) Load direct 30
(vi) Load indirect 30
load instruction, indirect addressing mode, 30 operand

Find AC content after execution of instruction

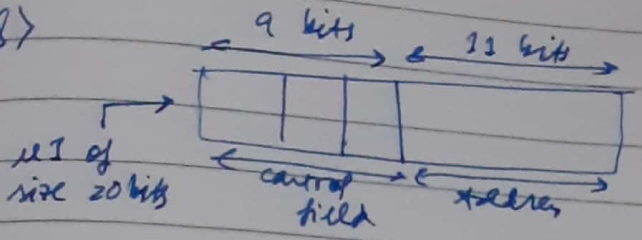
(ii) AC = 20 → 2 memory access → Read address 20 → read data at the address (40)
(iii) AC = 40
(iv) AC = 60

Q) In asynchronous serial communication controller uses a start-stop scheme for controlling the serial I/O of a system, it is programmed for a string of length, 7 bit, 1 odd parity bit and 1 stop bits.
Transmission rate = 1200 bits/sec
(i) What is the complete bitstream that is transmitted for the string 0110101 ?
(ii) How many such strings can be transmitted per second ?

(i) 1 0110101 1 → 10 bits long string
(parity ↑, ↑ stop (could be last bit))

(ii) $\frac{1200}{10} = 120$

Q>



Find the size of each control field and size of control memory

No. of M-op = 21

Degree of parallelism = 3

$2^1 + 2^1 + 2^1 - 3 = 21$

∴ size of each control field = 3

size of each location = 20 bits

no. of locations = 2^{11}

control memory size = $2^{11} \times 20$