Books :

Structured Computer Organization - Tanenbaum

Computer Organization - Hamacher

75% → Theory

25% → Lab

Theory :

Quiz	20%
Mid-term	30%
End-term	50%

- Computer is a concept of stored programs
- Program : set of instructions
- Needs memory (storage area) to store a program, processing unit is also required
- Needs I/P, O/P devices

1st Gen Vaccum Tubes

2nd Gen Transistor

3rd Gen IC

4th Gen VLSI → very large scale integration

↳ more no of transistors / area

The no. of transistors that you can put per unit area will double every 2 years → Moore's law

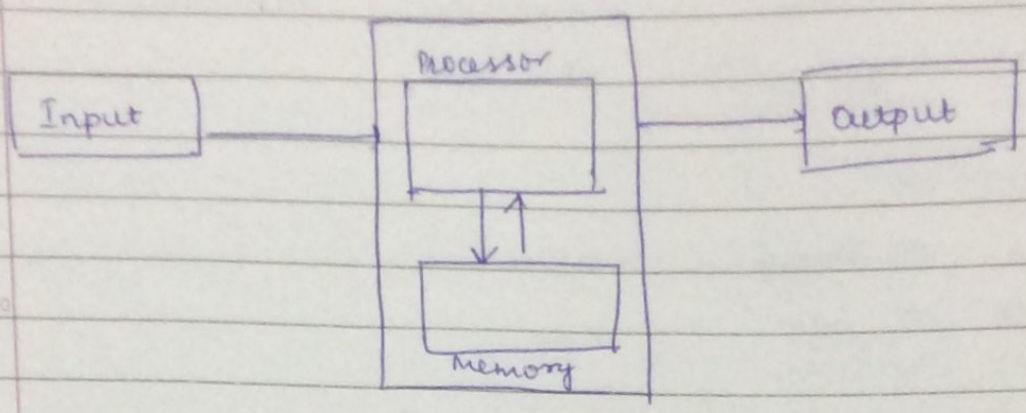
(It has slowed down now due to fabrication technologies)

5. Embedded Computers

When a program is being executed, it becomes a process.

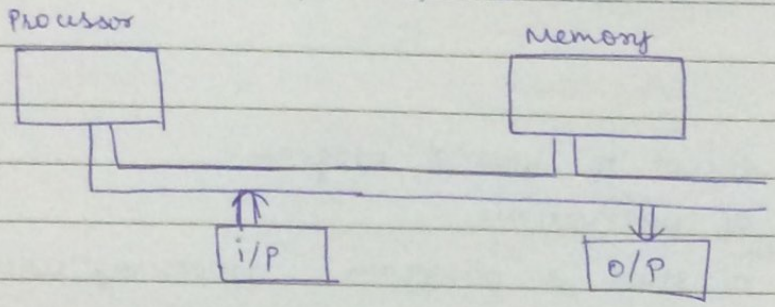
i/p devices : keyboard, mouse, joystick, webcam, sensors

o/p devices : Monitor, Printers, Speaker, Projectors



Von Neuman architecture

After evolution, now the architecture is :



Single Bus architecture:

Bus :- set of wires. Data travels over those wires, Over single wire, we can transmit 1 bit of data.

Multiprocessors :- Dualcore, Quadcore...

64 bit processor : can process 64 bits in one go.

Serial Bus :- Bits are going in a series.

ROM has boot
↓
non- volatile

int a = 8; 4 bytes is given to it

00000000 00000000 00000000 00001000

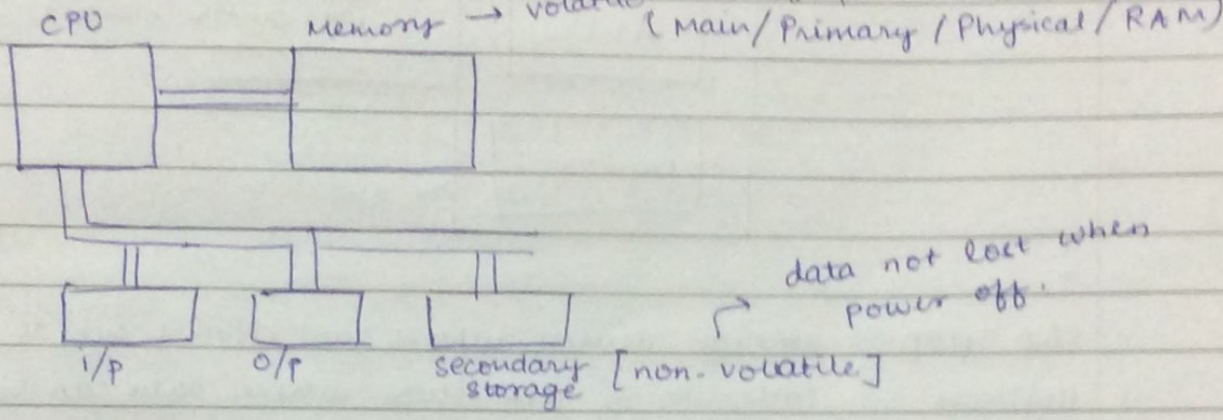
We have to send all 32 bits in one go. Hence, bus is required.

02-08-17

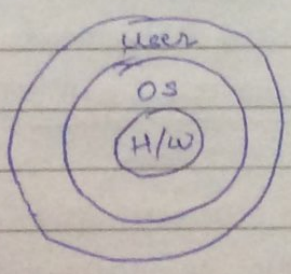
single bus can cause bottle neck and hence, make processor slower

↓
both i/o devices & memory want to communicate to CPU at same time.

We use double bus to avoid above kind of problems



- Programs are stored in Hard Disk (secondary storage) (part of i/p - o/p scenario)
- Whenever we run a program, it gets picked up from HD to RAM
- Operating system is the manager here.
  ↓
  assign specific spaces to all programs which are executed.
- OS : It is a software ⇒ It is also running ⇒ It should also be in RAM.
- On power on, ∃ boot program in ROM which will put OS in RAM.
- OS is interface b/w user & hardware.



One by one, instructions will go to CPU & will be executed.

Machinary language : Binary language

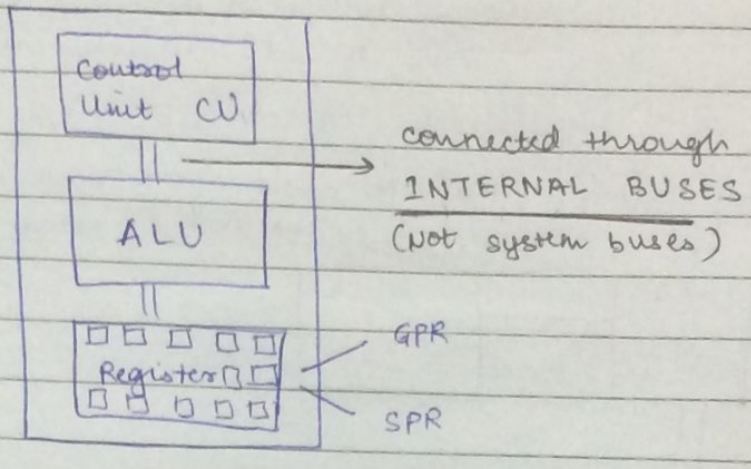
*Reuse digital circuits. → (chapter 3)

Camlin Page
Date / /

- Basic architecture for each machine is different
- High level language : c = a+b;
needs compiler → Assembly language : (use mnemonics) ADD R1, R2, R3
needs Assembler → Machine language : 0010 0001 0010 0011

* Assembly language is machine dependent, so is assembler.
- Modern day compilers directly convert HLL to Machine language
- Program is stored in RAM in machine language.

CPU :



- Flip-flop : storage device which can store 1 bit of data.
- Registers : collection of flip-flops where data can be stored. can store 1 ~~bit~~ word of data
- 32-bit machine : 32 bits can be processed in one go.
usually, the word size is 32 bit. We need to store 32 bits. Hence, size of register : 32 bits. (32 flip flops)

int a = 8 ; stored in register as :

00000000 00000000 00000000 00001000

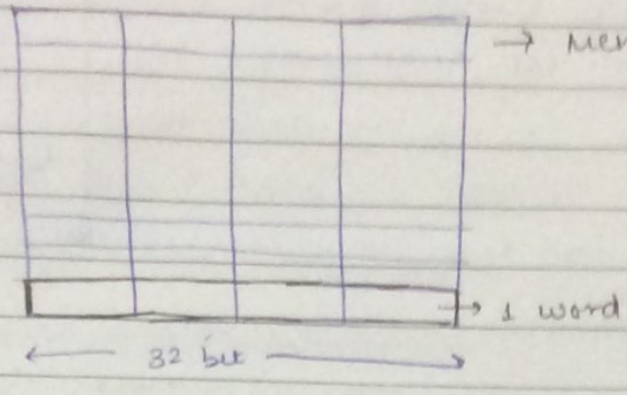
← word →

- Registers could be :
1) General Purpose (GPR)
2) Special Purpose

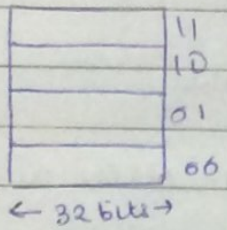
- ~~MAC~~ Upgraded machine will always run the programs that could be run on earlier one → has to be backward compatible

HD ~~contro~~ → magnetic disk

- Flip-flops are primary memory
- All RAM, ROM, HD ~~etc~~ have diff. manufacturing technologies



- To access a particular word in memory, we need its address
- Memory could be word addressable or byte addressable
  each word has an address
  ~~# of addresses = no of words~~
- suppose we have 4 words



need 2 bits to store addresses

- If there are 2^K addresses (2^K words), then K bits of memory will be needed to store the addresses

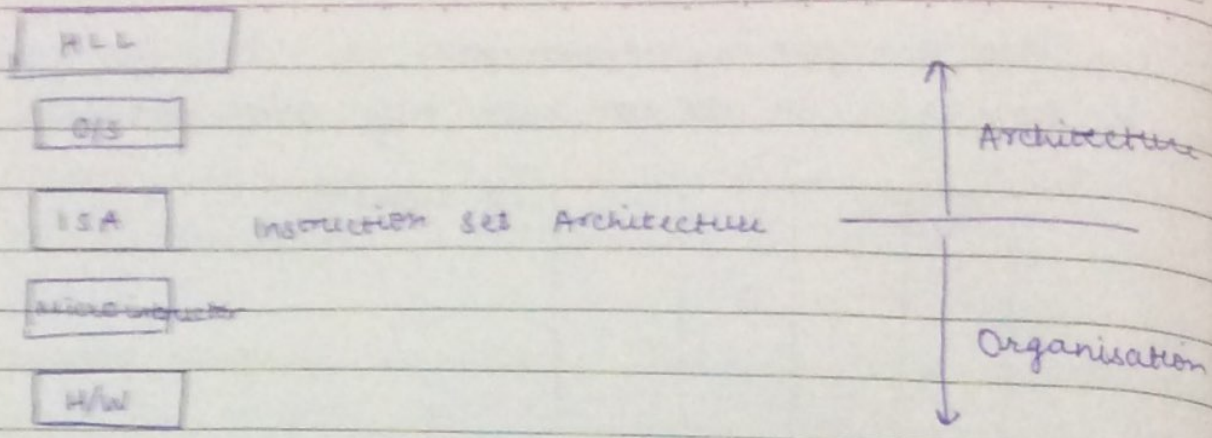
2^{20} : Mega
2^{30} : Giga
2^{10} (1K)

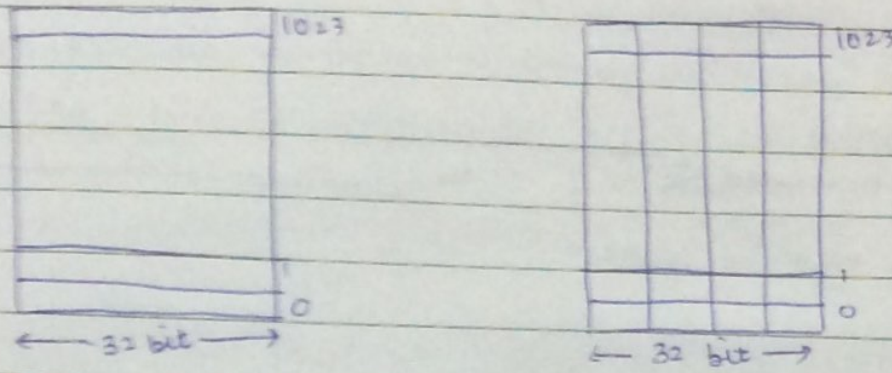
4-8-17

- Embedded processors / microcontrollers are used in washing machines
- It depends on system we design that how the processor should be, ...
- Your job as architect is to design.

HW : Decimal, Hexadecimal, Octal, Binary (conversions)

Camlin Page
Date / /





word addressable

byte addressable

→ In case of byte addressable memory, we can fetch a char. if we know its address (needs 1 byte)

→ 2 MB RAM $\Rightarrow 2 \times 2^{20}$ Bytes $= 2^{21}$ Bytes

Byte addressable ⇒ we will need 21 bits of addresses.

(By 32-bit machine)

4 words addressable : 4 bytes = 1 word

2^{21} bytes $= 2^{19}$ words

Word addressable ⇒ we will need 19 bits of addresses.

→ O.S is at bottom location of RAM (stored in starting)

→ When a program is executed, the starting add. of instructions will be loaded in a SPR (from RAM) called Program Counter

** always points to the next instruction to be executed

PC → Program Counter

MAR → Memory Address Register.

MDR → " " data "

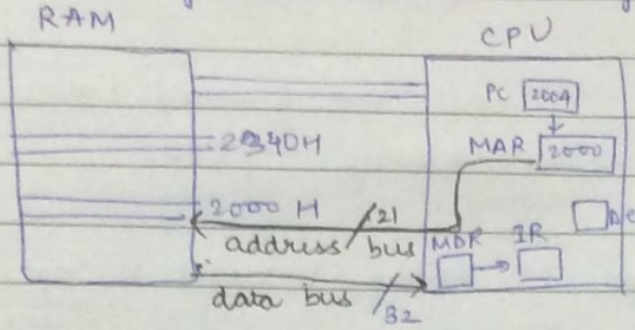
Multiple wires : ⟹ or ⟋⟶ (carrying similar type of data)

- The next instruction has to be fetched from the memory.
- PC gives this add. to MAR & at same time, PC will increment to point to next ~~instr~~ instruction ~~address~~. (next word)

~~It is word addressable~~

- Each bit doesn't have an address.
- It is word addressable ⇒ next address after 2000 H → 2001 H
  ⇓
  Each word has an address.
- If it had been byte addressable memory → next address after 2000H would be 2004 H (Assuming every instruction is of 1 word)

Byte addressable



we have written 21 bcoz 21 bit of address

2 MB RAM → 2^{21} Bytes ⇒ 21 bits needed for address

size needed of MAR = 21 bits.

Word addressable :→ size needed of MAR = 19 bits.

* Program is divided into 4 parts

Stack	→ store local variable
Heap	→ dynamic memory allocation
Data	→ store global / public variable
text	→ store set of instructions

- MAR is accessing the RAM so arrow dir^{n} MAR → RAM & read signal is also sent.

2000H : 21 bit : 0 0000 0010 0000 0000 0000

2 0 0 0

* The address bus should be unidirectional bcoz only Processor is sending add. to memory, not vice versa
  ⇓
  only add. is passed by this

IR → Instruction Registers

⇒ So, MAR will send add. to RAM. This content will go ~~from~~ to Processor through data bus.

→ Each word has 32 bits data stored & this complete data travels through data bus ⇒ 32 bits is required

→ MDR recieves the instruction.
So data bus is bidirectional.

* signals are all sent by the control Unit.

→ write, store : CPU → Memory / other device
Read, Load : Memory → CPU

→ It is assumed that 1st thing brought is an instruction So, it is taken to IR.

→ Purpose of IR : To hold the current instruction which is currently executed.

→ Decoder will decode & instruction will be executed. Then app signals will ~~to be generated~~ 2004 will be loaded in PC.

⇒ This cycle is known as Fetch - Decode - Execute Cycle.

Eg. ADD R1, R2, R3 (opcode: ADD; operands: R1, R2, R3)

R1, R2, R3 : GPR holding the variables

HLL : machine independent
AL : machine dependent

This ~~reason~~ can mean :

destination R1 ← R2 + R3 source ADD : operation to be performed
or R3 ← R1 + R2 ↳ opcode

This is a 3 address instruction (As it has 3 operands)

2-add. ADD R1, R2 : Add R1 & R2 & store in R1/R2 (machine dependent)
R1 ← R1 + R2
or R2 ← R1 + R2

⇒ only machines with 1-add have spl. registers called "Accumulator"

→ ~~Add~~ Add" is taking place in ALU.

Camlin Page
Date / /

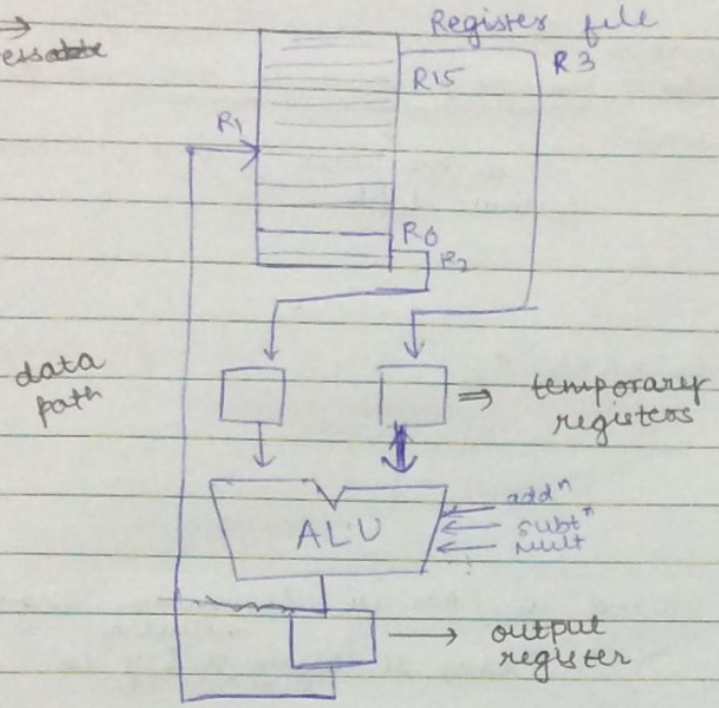
opl

1-add ADD R1 ; 1 a register : Accumulator which is taken as 2nd operand & result is stored over there only.

Acc ← Acc + R1

0-add ADD

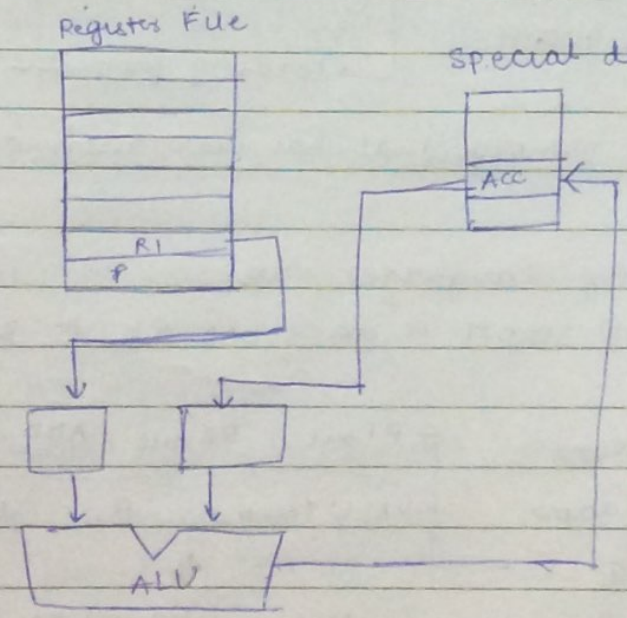
3-address~~add~~



ALU operates on 2 operands at one go.

Only 1 memory access when we have to fetch the instruction.

1-address~~add~~ :

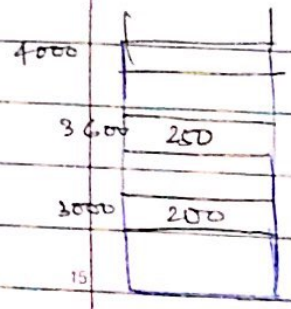


I/p & O/p registers of ALU are temporary registers

How to put values in accumulator :

```
{
  int x, y, z ;
    x = 250; y = 250 ;
    z = x + y ;
}
```

⇒ When we declare variables, memory space is assigned to 3 variable (4 bytes each) in the stack (local variables follows LIFO)



: stored in RAM

stack

"z = x + y" will be stored in RAM in Assembly language

brought value of x to R1 ← LOAD R1, 3000H
LOAD R2, 3600H
ADD R3, R2, R1
STORE R3, 4000H

have to bring values of x & y in processor

→ Loading from the memory to process

↓

memory that has been assigned to z.

11/08/17

Some assembly languages use :

LOAD R1, @ 3000H or LOAD R1, # 3000H

→ LOAD R1, 3000
LOAD R2, 3200
ADD R1, R2
STORE R1, 3350

R1 out, R2 out, ADD, ALU Temp in
ALU Temp out, R_{IN}, END

↓

Micro instructions : actually executed in hardware. Each micro-instruction takes 1 clock cycles (usually)
(The whole line is 1 micro instruction)

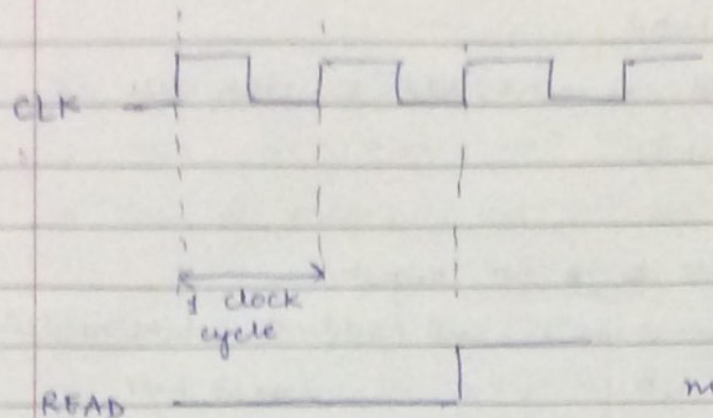
↓

everything in one clock cycle

Clock cycle

Camlin | Page
Date / /

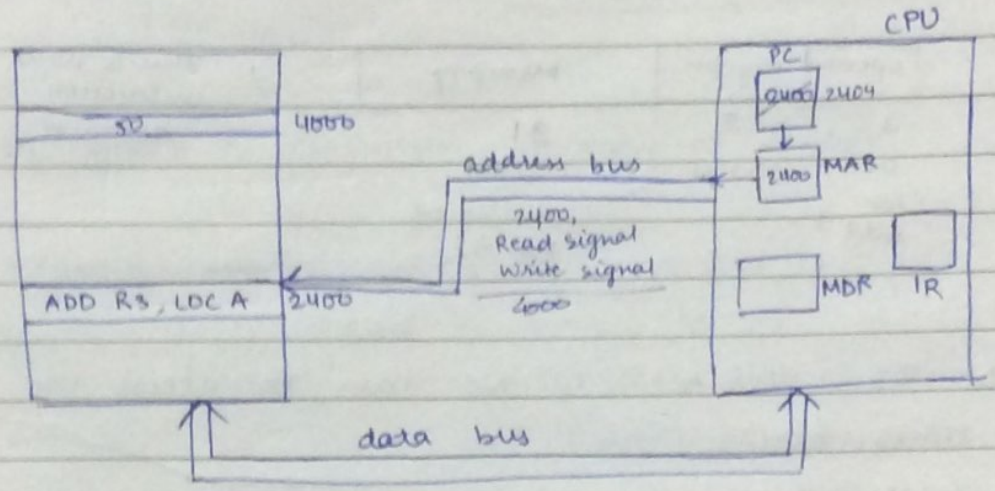
- Everything happens on a clock pulse.



(It depends on implementation by the system how many clock cycles will be taken)

If memory has to be accessed, a micro instruction will take more than 1 clock cycle.

ADD R3, LOC A LOC A → 3000



* The instruction ADD R3, LOC A will go through data bus to MDR.

After going to IR, it will get decoded and then, it ~~still~~ doesn't have LOCA data bcoz it is in the memory (while R3 data in the processor).

Now, we have to again access the memory to get ~~from~~ data from memory.

So, we will put the address 4000 in the MAR while PC doesn't loose the track of the program.

So, 50 will go to MDR and then to other Temporary register while R3 ~~data~~ will go to one Temp. register & after Op^n in ALU,

it will reach the destination.

- The decoding unit will decode that the add of LOC A is 4000 and send it to MAR
- PC will not pass the add. of next add. to MAR till the current instruction ends.

- 16 GPR processor ⇒ need 4 bits for register
  If it can perform only 2 op^{n}s sub / Add ⇒ 2 bits needed
  " " " 8 ⇒ 3 bits

3 bits : 8 operation's ALU
5 bits : 32 bit Registers
21 bit memory address ⇒ 2 MB

operand	Register	Memory
3	5	21
011 (for Add^n)	00011	4000 (address of LOC A)

: This is how instruction is decoded

12/8/17

* ADD R3, LOC A : We have to access the memory 2 times here.
1) To fetch the instruction
2) To fetch operand from LOC A from the memory.

→ The registers inside processor are connected by internal bus very fast

→ CPU is connected to memory by system bus which slows the processor as it has to communicate outside. (very slow)

→ MUL A, B : Easy instruction for user

→ Processor speed : depends on clock cycle.

Here, hardware is doing ~~add~~ repeated add" internally (let your hardware be of that kind, not direct mult)

So, MULT A,B will be complex instruction.
- this instruction will have lot many microinstruction
- ease of programming but lot of time is wasted.

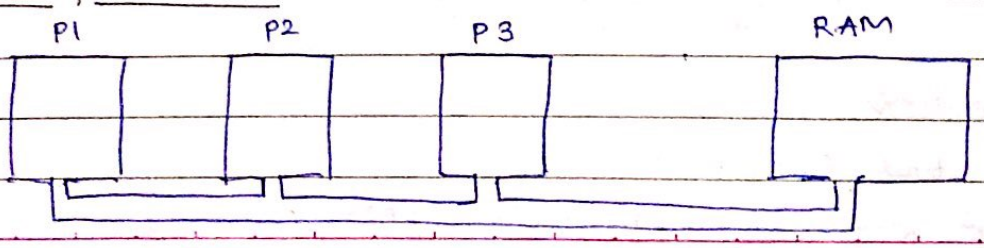
- Complex Instruction set computers CISC
  - This affects the performance. So, introduced
  Reduced Instruction set computers RISC
  But we can't simplify jump to RISC from CISC & start from the start.
  So, every machine is always backward compatible.

RISC :
- Usually fixed length instruc^n's, easy to decode.

To maximize the efficiency of processor :
- Set of instructions should be simple so that it is easy to decode
- Minimize memory access
  Less use of memory because it takes lots of time to access the memory. So, more GPR should be present.
- More use of Cache (an internal or intermediate memory).
- Have instructions that can be directly executed (1 microinstruct') on the hardware.

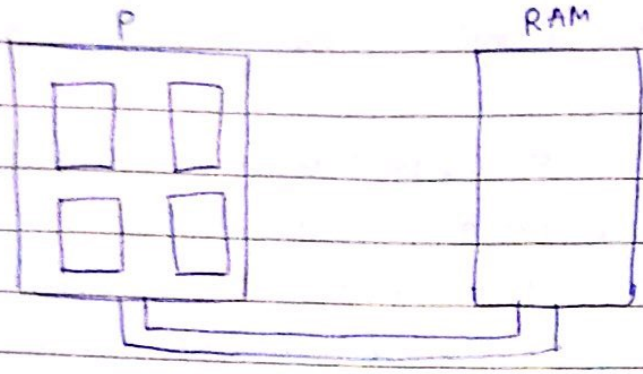
→ Now days we are using hybrid. Old architectures to run old program and new to run new programs.

Multiprocessors :-



— (1)

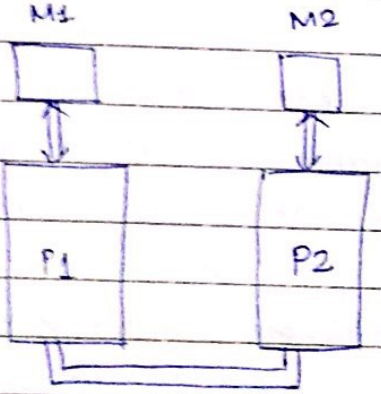
- P1, P2 and P3 are interconnected with each other as there might be a possibility that they need to communicate (1 programs is running on all processors which has many computations).



—— (2)

Single processor : Quad core (has 4 cores)
(multicore)

(1) : Only 1 processor can communicate with the memory at a time in this system.



Both processors have their own memories

Multi-computer system
(in networks)

→ A processor connected with a memory is a computer.

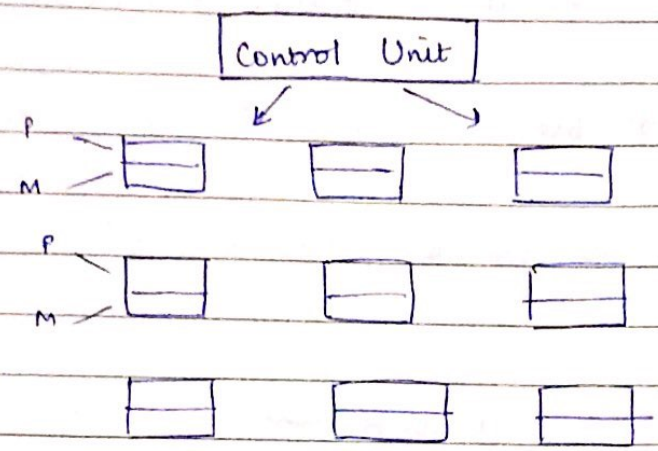
* There could be possibility of having a shared memory also. (in case of multi computer system)

Workstation :

High End Computer
(High frequency, high memory RAM, coprocessor)

- In a network of computers, 1 particular computer handles all the computers in the network → main frame / server.

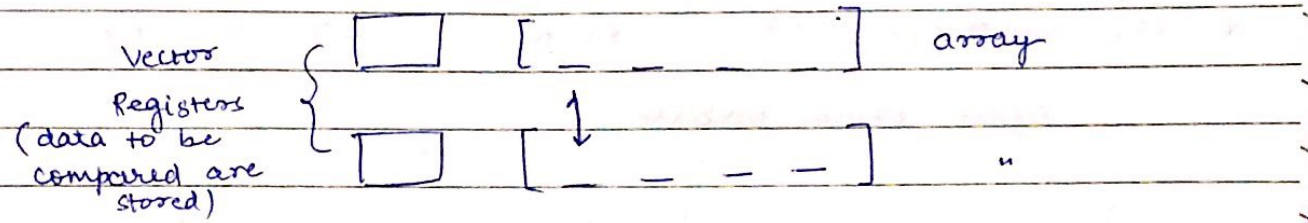
Array Processors : same kind of computation is to be performed on all the data.



grid of processors (small) each having its own small local memory. CU being sent to all the processors

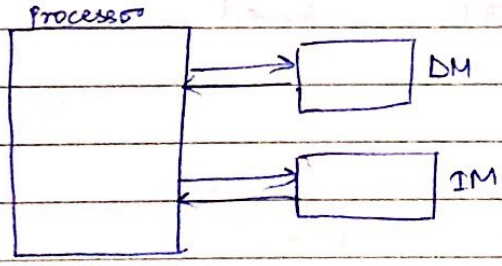
specific app^n's only (not in our daily computers)

Vector Processors : When oper^n's are performed on a pair of data.



→ Till now, both instructions & data are stored in the same memory. (only 1 can be executed at one go)

Split Memory :



Harvard Architecture

Here, we can access both DM & IM at the same time.

Boolean Algebra :

→ -8 : 1000 0

3 Ways to represent negative no :

1) Signed bit magnitude :

MSB : represent sign

4 bit : Range : -7 to +7

↓ for sign

n bit : Range : $-(2^{n-1}-1)$ to $(2^{n-1}+1)$

Problem : have 2 representations for 0 (+0 & -0)

2) 1's complement :

		1's compliment	
+0	0000	1000	-7
+1	0001	1001	-6
		⋮	
	0110		
+7	0111	1111	-0

We can't represent 8 as 1000 (used to represent -7)

Range : -7 to +7

Again same problem.

3) 2's complement : (2's complement : 1 in MSB)

+0	0000	1000	: -8
+1	0001	~~1010~~	~~10-1~~
	⋮		
+7	0111	1111	: -1

-0 : 1111 ⇒ Range : -8 to 7

+1

10000 : same representation for 0

- Range : $-(2^{n-1})$ to $+(2^{n-1}-1)$

→ signed no : ∄ positive no as well as negative no.

→ Unsigned no : ∄ only positive no.

4 bits : No. of unsigned no available : 16

1111 : 15 Range : 0 to 15

⇒ 0 to $2^n - 1$

8 bits : 0 to $2^8 - 1$

0 to 255

signed bit magnitude :

Range : −127 to +127

2's complement : −128 to +127

Unsigned

→ Floating point representation :

$x_3 x_2 x_1 x_0 \cdot x_{-1} x_{-2}$

$x_0 \times 2^0 + x_1 \times 2^1 + x_2 \times 2^2 + x_3 \times 2^3 . x_1 \times 2^{-1} + x_2 \times 2^{-2} + \ldots$

8.5

unsigned representation : ~~$8 \times 2^0 \; 5 \times 2^{-1}$~~ $\frac{5}{2}$ =

1000 . 1

→ −9 4 bit rep" using 2's compliment : Not possible

→ 1001 . signed no. value : −7

↓

→ $-(1 \times 2^3) + 0 \times 2^2 + 0 \times 2^1 + (1 + 2^0)$ (add − sign in front of MSB in case of signed no)

$= -7$

If add 1 in front:

11001 00111

Note: Extending the sign bit (whatever value is given in MSB) doesn't change the no. This is called as sign extension.
{signed no.}

→ Half adder : I/p don't have carry, diff. variable i/p

→ Full adder : process carry bit also.
1 → not taken as i/p

```
  1100          1100
  1101          1101
 -----         -----
 10001         11101
```
(Half adder) (Full adder)
(I/p has no carry)

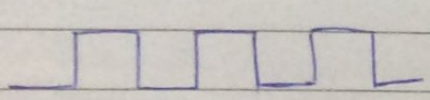
→ 1 Full adder was taking care of 1 bit only

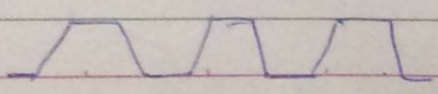
⇒ Shifter : moves the bits of a nibble one position to the left

18/8/17

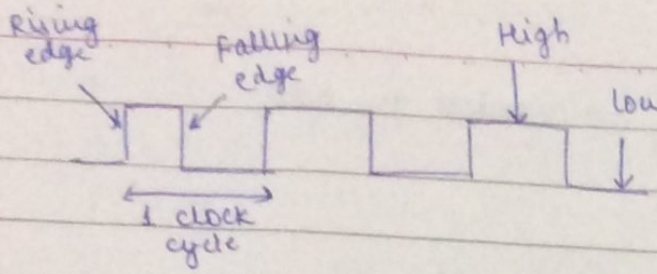
Sequential Circuit: Have some memory & remember their previous states. (Usually have flip-flop to remember their previous state)

- A "clock" is a spcl. circuit that sends electrical pulses through a circuit.
- Clocks produce electrical waveforms


(Ideal clock pulse)

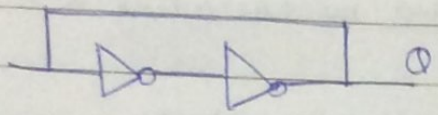

(In practical)

set of logic gates to implement boolean func" : Combinational Circuit



Edge triggered : change state on rising edge, falling edge.
level triggered : change state at highest or lowest level

→ To retain their state values, these circuits rely on feedback.



If it is 0, it will always remain 0. If 1, always 1.

– S-R flip flop : 1-1 : not defined.
S → Set
R → Reset

– D flip-flop : used in memory bcoz ~~don't~~ doesn't change o/p

Slide 60 : 3 flip-flops in each word ⇒ 3 bits
→ Need 2 bits (2 select lines S_0 & S_1) to store address of 4 words.
I/p lines : Sending data into the memory
O/p lines : Taking data out of memory

Slide 2 : 7 + 6 : 13 (signed bit)

```
  0111
  0110
 ------
  1101 ✗   → can't represent 13 in 4 bits in case of add" of signed bits
```

```
  0010
 ------
  0011 : -3
```

32 bits of x
32 bits of y
} ⇒ 32 FA needed to add

↳ have 4-bit ~~add~~ F.A.

32 bit of x
32 bit of y
} ⇒ 8 FA needed to add x and y.

K n-bit adders :

- K n-bit no. can by be added by cascading K n-bit adders.
- Carries ripple or propogate through blocks, **Blocked Ripple Carry Adder**

n-bit subtractor :

$X - Y \Rightarrow X +$ 2's compliment of Y

$\Rightarrow X +$ 1's compliment of $Y + 1$

$$X - Y = X + \bar{Y} + 1$$

n-bit adder / subtractor

```
  1 1 0 1 0 0 1 0
  0 1 0 1 1 1 0 1
  0 1 0 1 1 1 0 1      control line : 0
  1 0 1 0 0 0 1 0                     1
```

Add^n ⇒

2's compliment →

↳ sub^n : add 1, so $c_0 = 1$ (control line = 1)

Overflow :

- Overflows can only occur when signs of both the operands is same
- Overflows can occur if the sign of result is diff. from sign of operands. (7 + 6 = 13)

0 0 1

x_{n-1}

→ 0 1 1 1

y_{n-1} → 0 1 1 0

1 1 0 1

s_{n-1}

Overflow = $x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$

$0 + 1 = 1 \Rightarrow$ Overflow has occurred.

Overflow = $c_n \oplus c_{n-1}$

$0 \oplus 1 = 1 \Rightarrow$ Overflow has occurred.

Computing the adder time : Ripple adder

* For n-bit adder, s_{n-1} is available after $(2n-1)$ gate delays
c_n is available after $2n$ gate delays

21/8/17

→ $s_i = x_i \oplus y_i \oplus c_i$

$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

or $c_{i+1} = x_i y_i + (x_i + y_i) c_i$

» $c_{i+1} = G_i + P_i c_i$ — ①

G_i: AND op^n for x & y (generate func^n)

P_i: OR op^n for x & y (propogate func^n)

G_i & P_i don't depend upon c_{in}

From ①,

$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 c_0$

↓ only this term depend on c_0

This is called carry lookahead adder

Multiplication of unsigned no.

Multiplicand → 1 1 0 1 (13)

Multiplier → × 1 0 1 1 (11)

1 1 0 1 → called partial product

1 1 0 1 — ②

0 0 0 0

1 1 0 1

1 0 0 0 1 1 1 1 (143)

→ Product of 2 n-bit no. is at most a 2n-bit no.

→ We have to ensure that the register that stores the o/p is of double size

→ Unsigned multiplication can be viewed as add^n of shifted versions of the multiplicand.

→ Register stores ② also as 1101, (prev. also 1101) but at time of add^n, it gets shifted by one place.

→ Now, multiplier bit is 0, so it will directly shift without performing any add^n.

(Start from LSB) ↳ If multiplier bit = 1, initialize product = 0, add multiplicand to product & shift it right (not shift partial product left)

If multiplier bit = 0, simply shift the product to the right.

→ Register : | 1 | 1 | 0 | 1 |

| 1 | 0 | 1 | 1 |

after this bit has been checked, shift right so that again LSB has to be checked

| | 1 | 1 | 0 | 1 |

1101

→ We have to run the loop as many times as many no. of bits are in the multiplier.

↳ 32-bit machine : 2 int. mult ⇒ 32 bit multiplier
Product : 64 bits ⇒ 2 registers will be used to store product.

Signed Multiplication :-

```
      1 0 0 1 1      (-13)
    X 0 1 0 1 1      (+11)
```

Sign ext^n
```
1 1 1 1 1 1 0 0 1 1
1 1 1 1 1 1 0 0 1
0 0 0 0 0 0 0 0 0
```

Booth Algorithm

30 = 32 − 2 (as power of 2)

0011110 = 0100000 − 0000010

M x (30) = M (32 − 2)

```
                0 1 0 1 1 0 1
(add) for 32 ←      0 +1 0 0 0 -1 0 → for 2 (subtract)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 0 0 1 1 ⇒ 2's complement of the multiplicand
```

~~0 1 0 1 0 0 1 1~~

next bit & previous bit   0 & 0 : 0
1 & 0 : −1
0 & −1 : 1
1 & 1 : 0

0	1
+1	

1	0
−1	

→ In general, it reduces the no. of add^{n}s required.

0 1 1 0 1 (+13) ⇒ 0 1 +1 0 1
1 1 0 1 0 (-6) X 0 -1 +1 -1 0

29/8/17

Bit - Pair Recoding of Multiplier

0.101 ⇒ 0.625

1 1010 0110 1A654H

Eg. 5 × (-6) using Booth algo.

-6 = 1010
1111010
⇒ 0000-1 1 -1 0

M : 0101

1010
1011

CY	A (Partial Product)	Q	
0	0000	-1 +1 -1 0	
0	0000 + 1011 (Add 2's complement)	0 -1 +1 -1 (2's complement)	Shift (1)
0	1011	0 -1 +1 -1	
	while shifting, take care of sign extension → 1101 +0101	1 0 -1 +1	Shift (2)
1	0010 → 1001 + 1011	1 0 -1 +1	
		0 1 0 -1	Shift (3)
1	0100 → 1010	0 1 0 -1	
		0 0 1 0	Shift (4)

⇒ NO 10100010

→ 01011101
+1

01011110
6 5 4 3 2 1 0

16 + 8 + 4 + 2

25/8/17 Bit - Pair Recoding of Multiplier

0 0 -1 +1 -1 0

Think as this x_1 x_0 → $x_1 \times 2^{n+1}$
$2 \cdot 2^n$ 2^n $x_0 \times 2^n$

value of x_1 will be 2 times more than x_0

0 0 -1 +1 -1 0
0×2+0×1=0 -1×2+1=-1 -1×2+0=-2

Slide 38 : First Booth, then bit pair recoding.

Eg: 1 0 0
-1 0
-2

1 1 1
0 0
0

Eg: 13 × -6

Using bit - pair recoding :

0 1 1 0 1
0 -1 -2 → add 2's compliment 2 times

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

01101
10010
+ 1
(-M) 10011
10011
100110

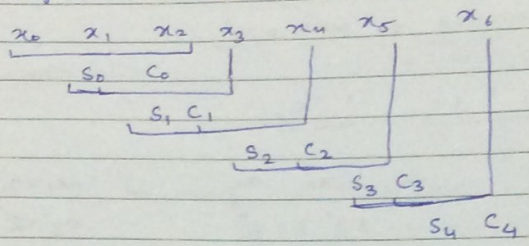
Camlin Page
Date / /

```
        0  1  1  0  1
        0  -1 -2
     1  0  0  1  1  0
     0  0  1  1
  0  0  0
1 1 1 0 1 1 0 0 1 0
```

→ Here, we are getting 2 shifts in partial product (we are passing up 2 bits)

Carry Save Addition of Summands

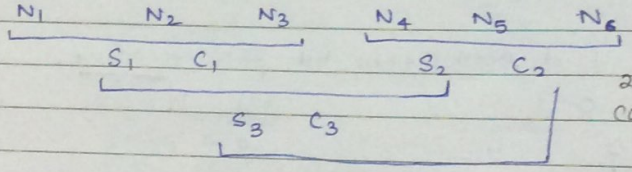
adding 3 no one by one



We can use here a look ahead adder

Slide no 44 :

Partial Products :



2 levels of carry save add.

→ not level of carry save add as we don't have 3 no. here.

Manual division :

```
      21
13 | 274 |
     26
     14
    -13
      1
```

How computer will do it

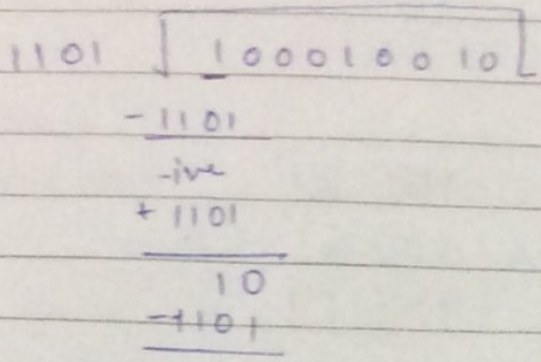
```
      0 21
13 | 274 |
    -13
    -11
   +13↓
     27
    -26↓
     14
    -13
      1
```

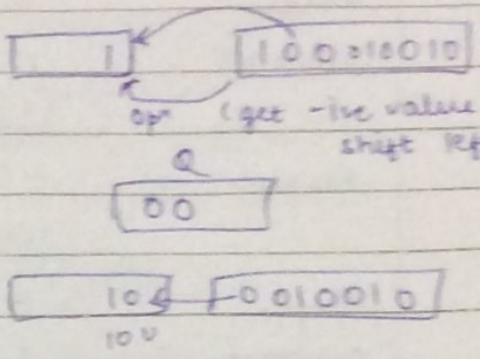
To restore no. (2) →

: since we are getting -ive no., Q=0. Now, we have to restore the no. ("2"). ~~so add the divisor~~ to the remainder.

~~Similarly~~

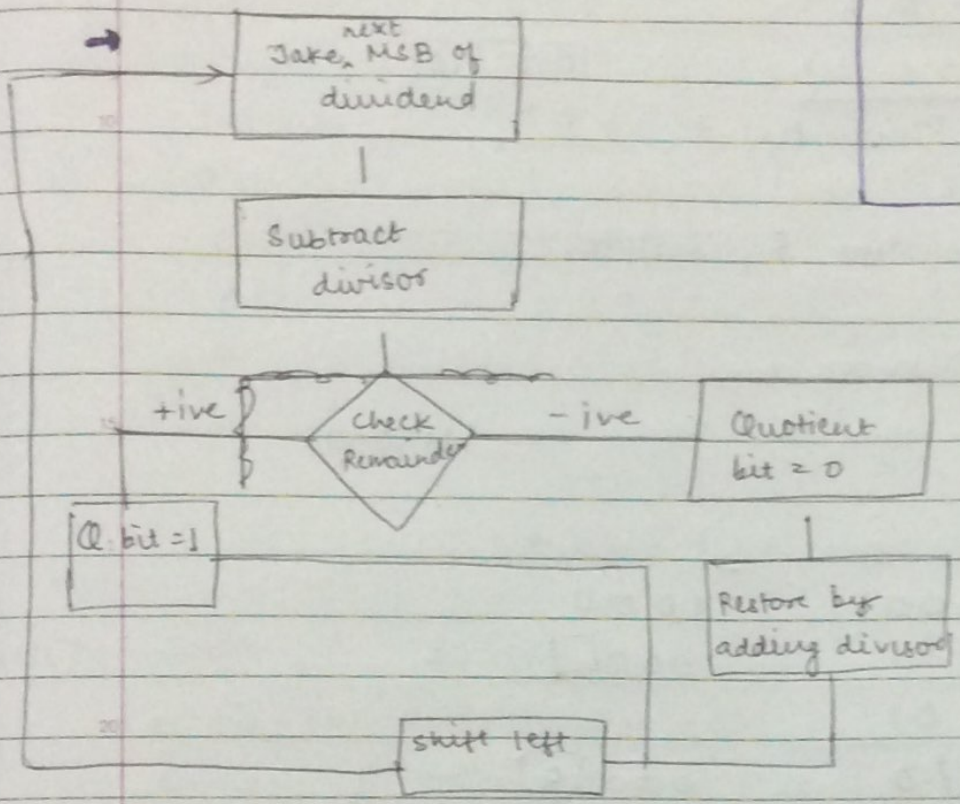
The machine will perform division of binary no in the same way





whenever R > 0 : put 1 in Q
shift left, add 1 more bit

R < 0 : put 0 in Q,
restore the R
shift left, add 1 bit more bit



Example : 11 | 1000 (Q)

A

1 0 0 0

8 : using 4 bits but 1 bit is taken extra in case of overflow / underflo

7 ÷ 4

0111 ÷ 0100

01000 : 5 bi

	A	Q
Initially :	0 0 0 0	1 1 1
Shift	0 0 0 1	1 1 □
Subtract	1 1 0 0	
Set q_0	1 1 0 1	
Restore	1 0 0	1 1 0
	0 0 0 1	1 1 0

0 1 0 0

2's comp. :

1 0 1 1

+ 1

1 1 0 0

Camlin Page
Date / /

Shift 0 0 1 1 1 0 □
Subtract 1 1 0 0
set q_0 1 1 1 1
Restore +0 1 0 0 1 0 0
0 0 1 1 1 0 0
Shift 0 1 1 1 0 0 □
Subtract 1 1 0 0
Set q_0 0 0 1 1 0 0 1
0 0 1 1 0 0 1
Remainder : 3

Problem : If negative R, restore.

Non Restoring Division

1000 ÷ 11 → 11100 (1)

Initially 00000 1000
Shift 00001 000□
sign bit of A: 0 → subtract
11101
1 ⇒ q_0=0 11110 000 0

Shift 11100 00 0 □
A: 1 add
00011
1 ⇒ q_0=0 11111 00 0 0

Shift 11100 0 0 0 □
A: 1 add
00011
0 ⇒ q_0=1 00001 0 0 0 1

Shift 00010 0 0 1 □
Subtract
11101
11111

13 ÷ 3
Shift
subtract
Shift
add
shi
sub
0
Sl
A
30/8/17
→ 32
R
→

Camlin Page
Date / /

13 ÷ 3

	00000	1101
Shift	00001	101 □
subtract	11101	
	11110	101 0
Shift	11101	01 0 1
add	00011	
	00000	01 0 1
Shift	00000	1 0 1 □
subtract	11101	
0	11101	1 0 1 0
Shift	11011	↑
Add	00011	
	11110	0 1 0 0

00011

00001

30/8/17

Fractions

→ 32 bit no. : 0. _

Range : 0.0000 ... 0 to 0.111 ... 1

$1 - 2^{-32} \approx$ 0.9999 ... 98

significant digits

$x = \underbrace{m_1 \cdot m_2 m_3}_{\text{mantissa}} \times \underbrace{b}_{\text{base}}^{\pm e}$ ⟶ exponent

$0 \le v(b) \le 1 - 2^{-n}$

→ In binary system, base = 2

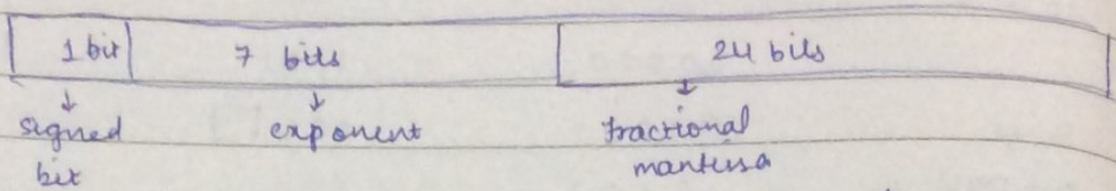
We already know the base ∴ we don't have to store it. (∵ implied base)

(Only exponent & mantissa)

⇓ can be -ive, +ive

→ use 2's complement

↳ signed bit is used



Float value representation (sample) ★

→ -0.75 $\quad 0.5 + 0.25 = 0.75$

-0.11 ; Binary conversion

To get it in scientific notation, we should shift it.

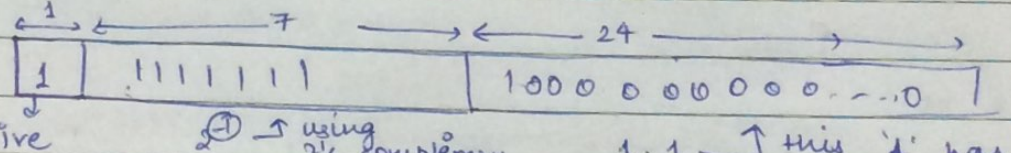
→ $-1.1 \times 2^{-1} \Rightarrow$ Normalised form

($-$: sign bit; 2^{-1} : implied base)

always we have to represent in such a way that this position has '1'. Hence, don't need to store this also.

We have to only store 1.1 → this '1' & exponent ↳ in 2's compliment form

01
10



-ive ; 2^{-1} → using 2's complement ; 1.1 → this '1' has

To check which no it is:

→ 1 ⇒ $-$

→ 1 1 1 1 1 ↳ ⇒ 2's compliment ⇒ -1 ⇒ $mm \times 2^{-1}$

→ 1 0 ..... 0 $\quad 1.100\ldots \times 2^{-1}$

* we can't use all 0's and all 1's in fractional mantissa, they are reserved

* If 3 bits, smallest non-zero no. : $0\ 0\ 1 \equiv 0.125$

next smallest " " : increment of $0.125 \equiv 0.25$

next " " : $0.25 + 0.125 = 0.375$ ...

$1.0011\ldots \times 10^{63} \longrightarrow 0.10011\ldots \times 10^{64}$

Camlin Page
Date / /

But we don't want negative exponent.
So we do an Excess-p coding

~~max~~ Range : -64 to $+63$: 7 bits alloted to exponent
⇒ We will add $+64$ to all exponents

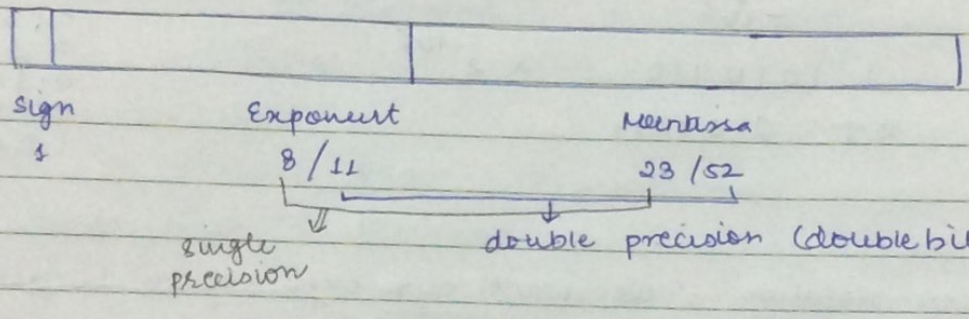
If 8 bit : -128 to $+127$ ⇒ add $+127$

so 2^{-1} $\quad -1+63 = 62$ ⇒ we will put 62 over there

benefit : • we can compare 2 floats easily
• we don't have to take 2's compliment

→ make lowest exponent (-64) 0 by adding 64.

IEEE 754 : We have to add $+127$ (8-bit)

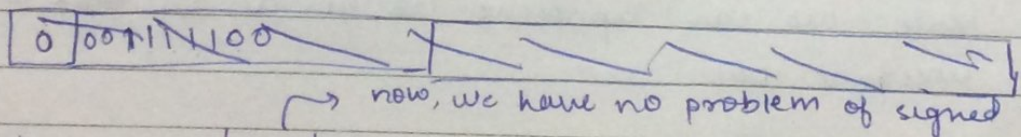


In case of 8-bit, we have to add $+128$ (least no. : -128)

→ 21.25
16+5

~~10~~101.01 $\quad +4+127 = 131$

⇒ $1.010101 \times 2^{+4}$ $\quad$ ~~100+128 = 132~~



→ now, we have no problem of signed bit

2	125	0
2	62	0
2	31	1
2	15	1
2	7	1
2	3	1
		1

2	132	0
2	66	0
2	33	1
2	16	0
2	8	0
2	4	0
	2	0
		1

2	131	1
2	65	1
2	32	0
2	16	0
2	8	0
2	4	0
2	2	0
	1	1

10000011

0	10000011	010101.---.

Ques:

1	000101 01	1010 110.---. 0

16 8 4 2 1

-ive

16+5 = 21

21 - 127

-106

$1.1010110\text{---} \times 2^{-106}$

2.09375

12
3125
× 5
15625

0.5
0.125
0.3125
0.15625
1.09375

1/09/17

Excess notation : we want our exponent to be always positive

→ 7 - bit exponent ⇒ -64 to +63

To get all the exponents, we should add 64

⇒ 0 to 127 : Excess 64 coding

hence, we can represent 128 no as all are unsigned no

While denormalizing, subtracting 64 from the exponent.

$E' = E_{true} + p$

→ 4 bit for exponent ⇒ Excess 8-coding. (-8 to +7)
0 to +15

Suppose we decide to use excess 7-coding
→ -1 to +14 ⇒ 0 to +14
not +ive ✗ True exponent : -7 to +7
(only +ive mantissa)

excess 6-coding (add +6) -2 to +13 ⇒ 0 to 13
E_{true} : -6 to +7
only this much range I can represent.

IEEE Notation :-

Bits available for exponent : 8
→ Range : -128 to +127

IEEE has decided excess -127 coding
Range : -1 to +254
actual range : 0 to +254 (Not representing negative no.)

However, IEEE reserves 0 & 255 for some other purpose.
→ available range : +1 to +254
E_{True} (-127) | -126 to +127 | ⇒ True range available to us.

How can we represent 0, ∞, NAN (Not a no.) ??
↳ not a real no.

Sign	Exponent	Mantissa	
0	00000000	00.....0	Pos Zero
	('0' is reserved for this)		
1	00000000	00.....0	Neg Zero
	(Any machine using signed bit rep^n / 1's complement representation would need this)		
0	1111 1111	00.....0	Pos. Infinity
1	"	"	Neg. Infinity
	"	Non-zero no.	~~NAN~~ NaN

→ If exponent e is represented in excess-p notation :

$$a \cdot 10^{(x+p)} * b \cdot 10^{(y+p)} = (a \cdot b) 10^{(x+y+2p)}$$

Ideally, it should have "x+y". But acc. to IEEE, it should have been "x+y+p".

But here, we have "x+y+2p". So, we have to subtract p to avoid Bias.

Eg. 2.3516×10^3 $\quad$ 4.25896×10^2

$= 2351.6$ $\quad$ $= 425.896$

↓ $\quad$ ↓

___ . ___ $\quad$ ___ . ___

* First, convert in binary form.

Excess-p

___ . ___ $\xrightarrow{\text{Normalize}}$ $1. \_\_\_ \times 2^{-}$ $\longrightarrow$ S E+p M

___ . ___ $\quad$ $1. \_\_\_ \times 2^{-}$ $\quad$ S E+p M

* While shifting the mantissa, don't forget 1 befor the decimal which we remove at the time of representation.

Guard Bits :-

1.011

× 1.101

larger mantissa but, only can store 3 bits → loosing some digits → keeping some extra ~~digits~~ bits called Guard Bits

Floating Point Operations : FLOPS

Truncation / Rounding

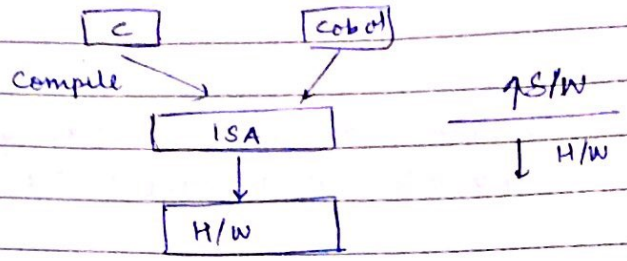
1) Straight chopping
2) Von Neumann Rounding
3) Rounding (used in IEEE)

If MSB of Guard bit : 1

⇒ 1 is added to LSB of the retained bits.

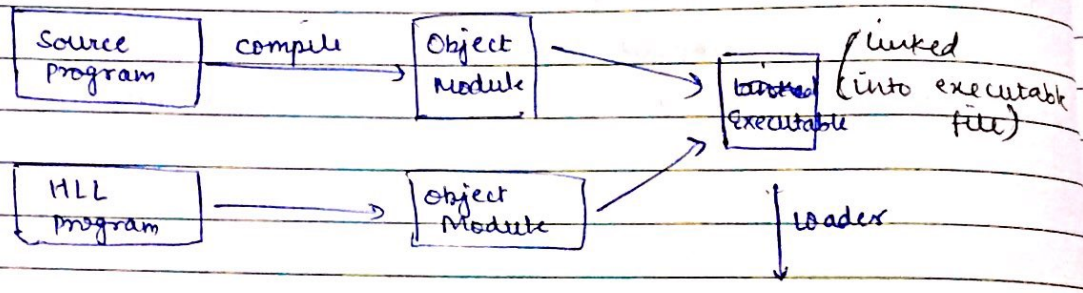
INSTRUCTION SET ~~STRUCTURE~~ ARCHITECTURE (ISA)

↳ Each machine has its own ISA, which actually runs on the machine.



We may have our program in any language, the compiler will convert it in ISA & then, compiled into object module

Object module :



After compiling, program is converted into Assembly language, & then into machine language.

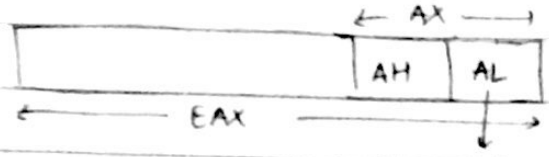
→ eg C = a+b

a → 4000
b → 4004
c → 4008

```
LOAD  R1, 4000
LOAD  R2, 4004
ADD   R1, R2
STORE R2, 4008
```

→ ISA : Interface b/w software & hardware

→ In Pentium 4, some registers are



32-bit

1 byte data

16 bit of data is used

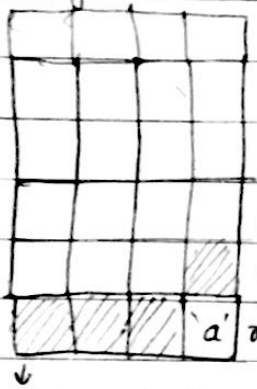
(earlier programs which use this much data are able to get executed)

Upgraded machine must be backward compatible.

① Memory Model :- → ISA

Sometimes, these documents are available for everyone to see, sometimes not (Intel doesn't)

(assuming 32-bit memory model)



Now, if we have to store an integer, if we start storing at 1, if it is word addressable, we have to access 2 words & get the int. However, here, in byte addressable, it will be non-aligned memory.

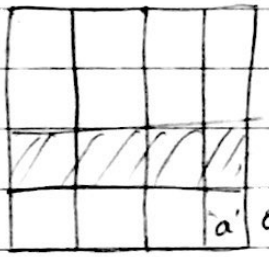
non-aligned memory

advantage : saving space

disadvantage : involve complexity in accessing & decoding the data

(Can use in case of smaller programs)

Other option :- always start at word boundary



→ aligned memory

→ First you have to choose this. Then, byte addressable / word addressable

→ 1 MB = 2^{20} bytes

byte addressable ⇒ 20 bits to store address

Word " ⇒ 18 " " "

→ Same memory, but no of bits in address has reduced.

Advantages ?? —— ①

Byte addressable memory :

→ accessibility improved (advantage) { disadvantage of word addressable memory }

If word addressable, we have to transfer whole word to CPU & then shift & get it to lowest bit. So, additional ~~work~~ instructions have to be executed.

—— ① : Some bits have to be given to address

Opcode Reg MLOC

↓ ↓ ↓ to store address, bits have to be given to this.

7 5 20 bits let total bits = 32

↳ If there are 5 bits to store address of 8 registers, max. no. of registers = 2^5 = 32 (R0 – R32)

↳ Only 7 bits for opcodes, max. opcodes possible = 128 (not sufficient)

test( )

0000000 → ADD Reg, Reg

0000001 → ADDI Reg, # N : adding value to register, storing in register

0010000 → ADDM Reg, LOCA : adding value directly from memory to register & store.

↓

{ Usually not allowed in the processors CISC instruction }

→ Opcode in all 3 cases will be diff. otherwise the decoding unit won't be able to perform properly

① ADD R3, R5

0000000 00011 00101 x : designing ISA

bits: 7 5 5 x

The decoding unit will find it is add^{n} op^{n} (1st 7 bits) which add 2 Reg. it will bring data ~~for~~ from R3 (next 5 bits) to IR & 2nd ~~Reg~~ data from R5 (next 5 bits) to 2nd IR & bring o/p in o/p register

② ADDI R3, # 1023 → Immediate (Immediately available in the instruction, don't have to look for it in registers)

↓

available : 20 bits

max no we can give : 2^{20}

But if we need to store no upto 2^{10} (1024)

000000 00011 1111111111 x

The decoding unit will get data from R3 (next 5 bits) to IR in ALU and directly send next 10 bits will get stored in IR in ALU

③ If o/p has to be sent back to memory o/p from o/p register in ALU to MDR & add will be stored in MAR & then data will be sent to memory through data bus