1-1-19.

IDE : Integrated Dev. Environment

file a.out → description

objDump -D a.out | more

- Basic steps:
  - → write a file sort.c & save
  - → saved on a pessistant [non-volatile] storage
  - → The hierarchical arrangement of these files (stored) : File system

Eg of File systems :- NTFS, FAT 32, FAT16, EXT 2

→ Compiled. Task of the compiler, Read binary encoded data.

- Executable file formats :- ELF, COFF

(-S)

- On running objDump -D a.out | more ✱
  - → Details of the executable are displayed.
  - → with operations (col 3) & operands (col 4).

To communicate with the processor the only way is through ISA: Instruction set architecture

Col. 2: Represents the operation instruction in hexadecimal representation.

→ x86 : ISA (proposed by INTEL)

ARM, MIPS, ARM, RISC V. → open source. ISA processors can be built

Intel i5 is a processor such that the ISA corresponding to x86 can be understood with the same.

To design a processor:-

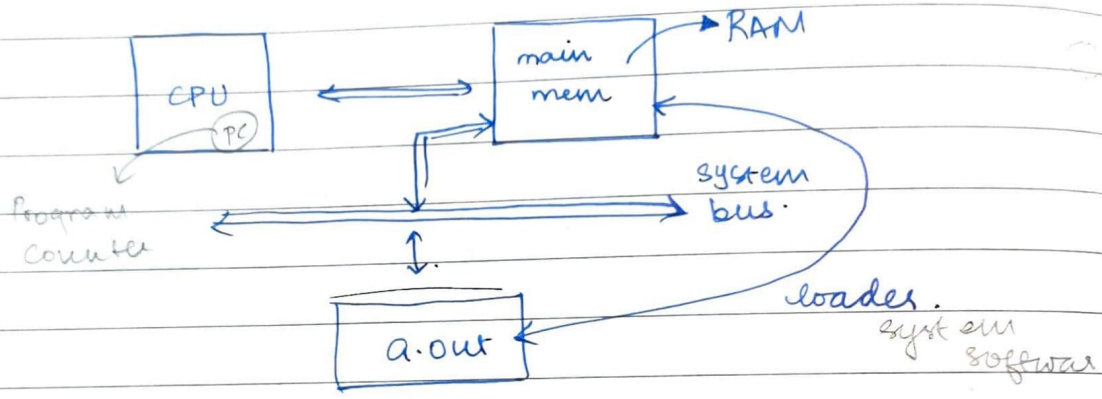
i) Speed
ii) Power
iii) Cost
iv) Size

ISA is what the chip understand through.

ISA
- Variable length encoding (used in CISC) x86
- Fixed length encoding (ARM)

→ Advantage of variable length :-
code length can be adjusted acc. to req.

→ Disadvantage:
Decoding becomes complicated (appropriate machinery & equipment is required).



Main-memory: linear array of bytes. (memory)
It is only Byte addressable.

1st col of ✱ it ~~assigns~~ is the address of the cell where that particular instruction sits.

→ Execution Model.

Step: 1>. Fetches instruction (PC)
PC: contains address of next command to b executed.

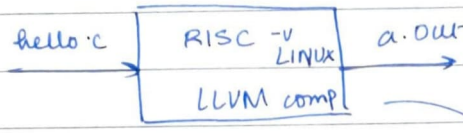
2>. Instruction decoded

3>. Execute the instruction.

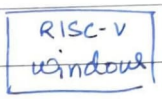
→ ltrace ./a.out.
↳ gives details of all library calls.

Hardware exposes its ISA for the OS.
OS exposes its system call interface for application software.

Library call → User space.
System call → Kernel space.

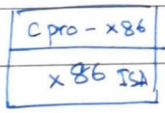


cannot work on.



Because the system call interface varies.

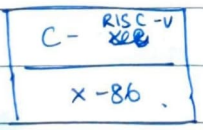
→ Native compilation :-



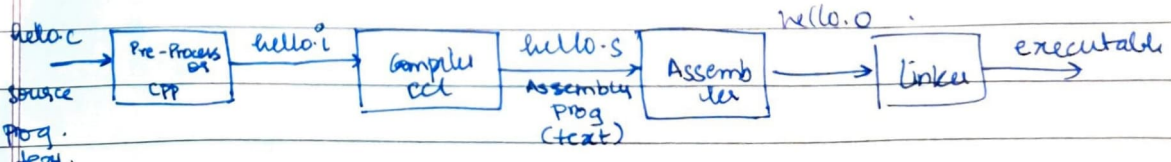
compilation & run on the same machine.

→ Cross compilation :-

Eg arduino:
Compiled on one machine run on another.
Eg: Developing a software for a sensor node.



• Software interface:



way assembly lang. is an intermediate.

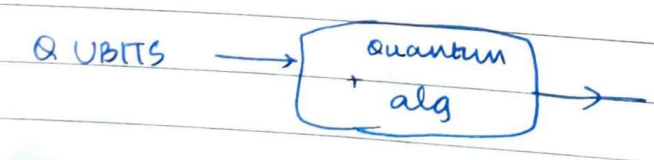
(1). for efficiency & performance.

(2). Most code for an OS can be written in a programming lang. however, 5% of the code will be tied to the processor, (its memory management etc.) ∴, it is processor dependent & you may need to write it in Assembly lang.

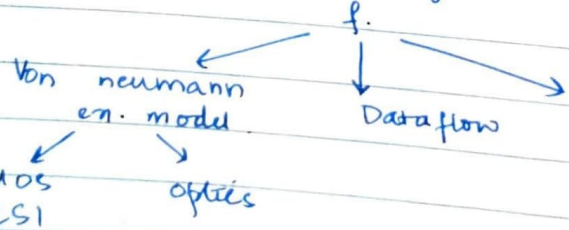
~ Solving a computational problem :-

Fig. Execution model.

- Quantum execution model.



Physical Realization of ex. models.



otherwise computing is useless

Optimizing encoding & decoding processes is essential.

- Objects can be classified as:-

Primitive	Composite
- Integers.	- Rational no:s.
- Real no:s.	- Complex
- Alphabets.	- Matrix
	- Images, video.

Space allocated to different data types varies from machine to machine.
1 bit / 3-bit / 13 bit data types are unheard of because, it is not possible for the processor to represent multiple wide range of nos:

~ Unsigned rep

$b_n\ b_{n-1} \ldots b_2\ b_1\ b_0$

↓

represents mag. 2^{bn}.

~ Signed rep.

$b_n \ldots b_2\ b_1\ b_0$

if 1 → -ve no: } signed magnitude rep.

0 → +ve no.

	U	SM	1's	2's.
0 0 0	0	0	0	0
0 0 1	1	1	1	1
0 1 0	2	2	2	2
0 1 1	3	3	3	3
1 0 0	4	0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	0	-1.

For a w bit no:

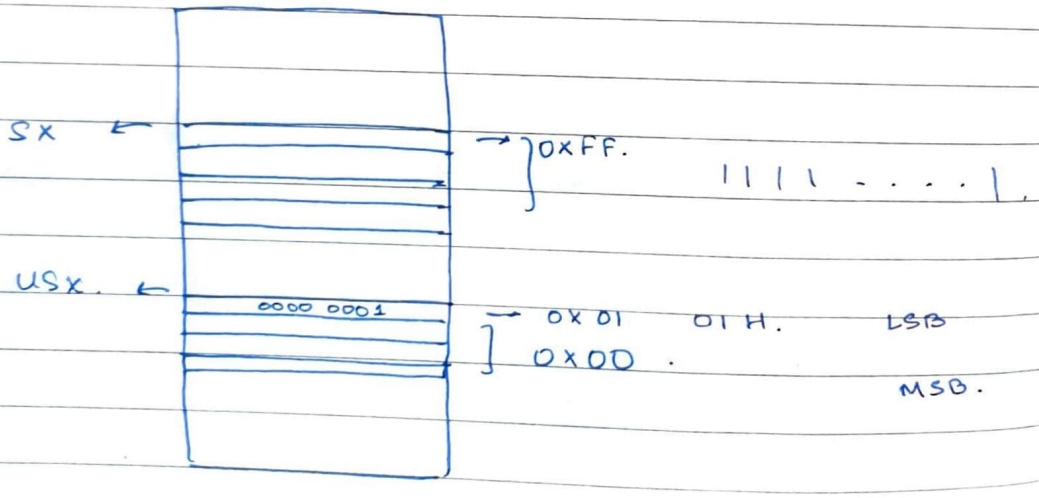
b_{w-1} b_{w-2} . . . b_1 b_0

- $-2^{w} - 1$: 2's comp.
- $-2^{w-1} - 1$: 1's comp.
- $2^{w} - 1$ unsigned.

∵ Rules of binary arithmetic are the same, the same circuit (eq.) is invoked for operating on signed or unsigned numbers

eg: $Sx = -1$.

$USx = 1$.

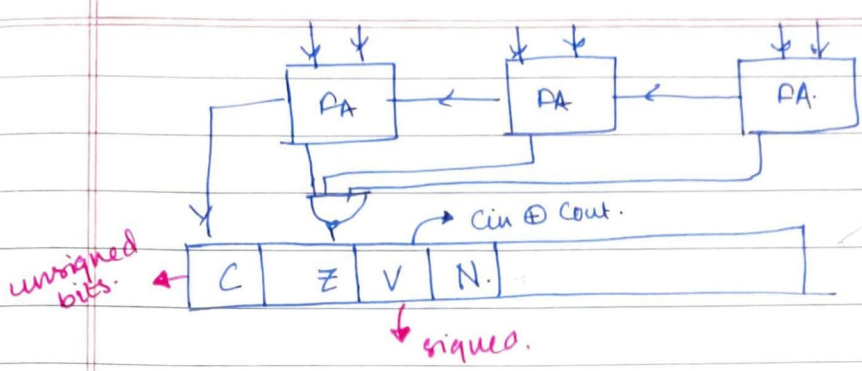


Overflow:

Overflow can be determined using a Carry bit that is the final output carry.

Z bit is 1 - when result of arithmetic op is 0. else it is 0.

For signed bit nos check → V.
For unsigned → C



Case:

$-1 \quad -0 \cdot 111$

$-3 \quad 110$

$(1)\ 001$

} This isn't an overflow ∵ closure property isn't violated.

signed rep

$3 \quad 011$

$2 \quad 010$

$5 \quad 101$

-8

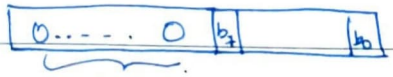
-3

} This is a case of overflow.

Rep: C - carry out [final].
Z - Zero bit (checks if result is 0.
V - Cin ⊕ Cout.

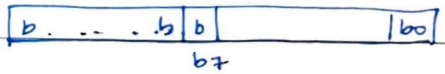
→ If the addends are from 2 different domains of bits [Eg: 16 bit no + 8 bit no].

For unsigned nos.
zero-extension solves the problem.
∴,



8 bits.

For signed nos:
signed extension solves the problem.



→ for 3-bit R & L shift

100 —LSR→ 010 . → 4/2 = 2

Logical shift right.

100 —ASR→ 110 . → −4/2 = −2.

Arithmetic shift right.

- Using gdb:

→ p & varname

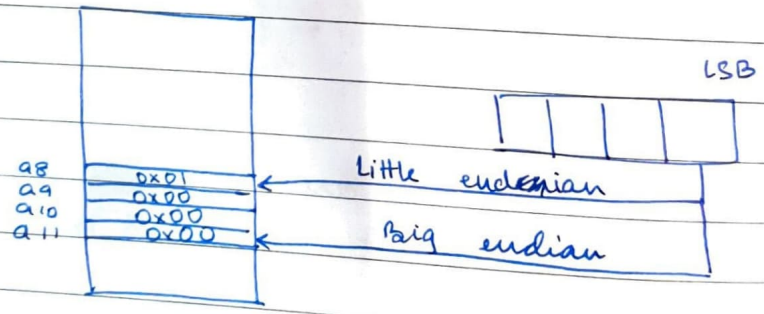
prints address of varname.

→ x/4b 0x FFF... ac

shows value stored in the 4 bytes starting from 0x7fff..ac

→ p/x 0x7fff....

→ p/x * address. → value in address



- Programming lang. provides.

1) Primitive data type [ISA]
⇕ compiler (bridge)
2) Mechanisms to compose new data types
[arrays, structs, unions]

→ Programming lang. must support arithmetic & logical op.

→ Should support control flow constructs. Turing complete.
  ↳ if-else
  ↳ while, for.

→ Memory transfer.

→ Turing machines.

For a lang. to be turing complete :-

- the ISA should have instructions for every line of code. [for every possible computational program]
- Ease of programmability

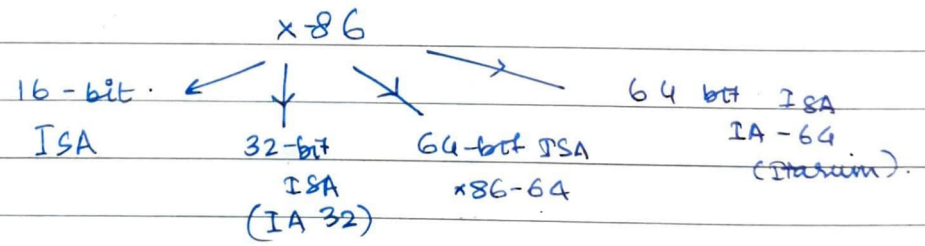
compiler. ALP

RISC CISC

fixed length enc. variable length enc.

×——×

- x-86 Instruction set Arch.



→ x86 - 64.

Total 16 registers.

8 bytes.

%rax %eax. → rep the last 4 bytes of rax

%rbx
%rcx
%rdx
%rsi
%rdi
%rsp
%rbp

r8
:
:
r15

→ ensuring compatibility on multiple processors

Split of %eax.

%rax → 64 bit / 8 bytes

↓

%eax → 32 bit / 4 bytes.

%ax (2 bytes) , %al (1 byte)

- Layout of Instructions:

Source Dest — Src, Dest.

movq $0x4, %rax

Imm

Reg → specifying the register.

The operand itself is encoded in the instruction

	Source	Dest.		Source, Dest.
				0x4 moved to rax
(1)	Imm.	Reg	movq	$0x4, %rax
		Mem.	movq	$-147, (%rax)
				↓ 2's complement stored in address stored in %rax.
(2)	Reg	Reg	movq	%rax, %rdx
		Mem	movq	%rax, (%rdx)
(3)	Memory	Reg	movq	(%rax), %rdx

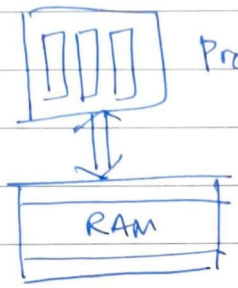
* Movement from one location in memory to another location in memory is not done ∵ not cost-effective.

from the memory add in %rbp offset 8

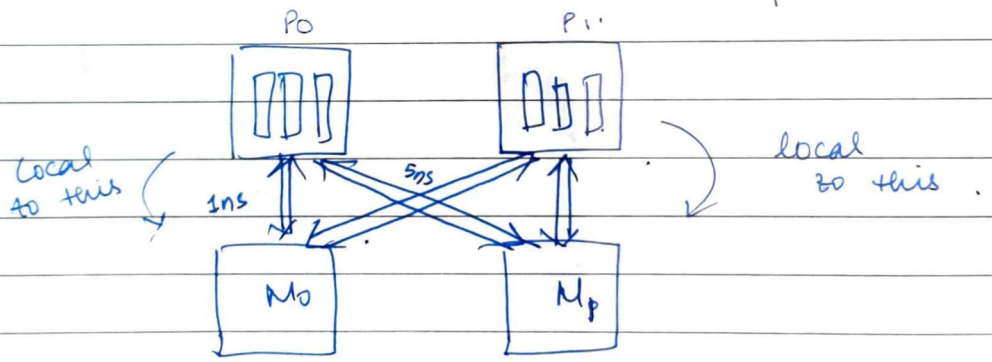
Spl: movq 8(%rbp), %rdx.

- Instruction classification :-
  1. Data Transfer. - control flow.
  2. Arithmetic & logic - data transfer.
  3. control transfer. - logical/arithmetic

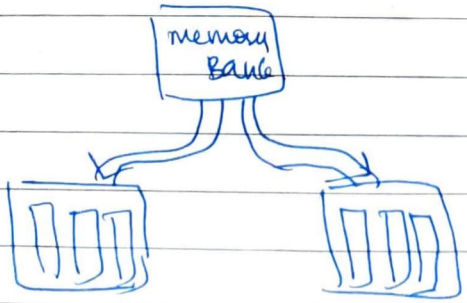
UMA : Uniform memory access (time).



NUMA : Non - uniform memory access.



Case :-



Memory bandwidth reduces since both processors access the same bank (pool) of data

Main memory layout :-

Arithmetic intensity = $\frac{\#\text{computations}}{\#\text{memory access}}$.

Goal: Maximize A.Intensity so as to keep the processor busy and doing work.

RISC

```
load   y  R1
load   z  R2
ADD    R1 R2 R3.
Store  R3 x.
Load
Inc    R3
Store  R3 x.
```

A.I : $\frac{2}{3}$

Exclude

$x = y + z$

∴ Minimize memory access [DATA REUSE]

eg:

CISC

```
LOAD   y  R1
ADD    z  R1
STORE  R1 x
```

→ x86 instruction set.

- Data Transfer.
  MOV :- used for both LOAD & STORE.

Structure of writing the instructions :-

D (Rb, Ri, S)
- D: displacement
- Rb: base reg
- Ri: index reg
- S: scale factor

(%rax) → normal.
8 (%rax) → 8 + (%rax)
-16 (%rax, %rbx) → -16 + rax + rbx.
32 (%ram, %rax, 4)

Analogous :- A[i].

To access 4th el.
Base + 4 × index.
⇒ (%rax, %rbx, 4).

Eg: P = &x[i];
y = *P;

P ⇐ %rax i %rcx.
X ⇐ %rbx. &y %rdx.

Instr:- lea. (%rbx, %rcx, 4) %rax. (calculates instruction; stores it.)
movd (%rax), %r10d
%r10d, (%rcx)

flags are not set by this instruction.

∴ cannot read 2 addresses from memory simultaneously.

Flags analogy (86 & ARM)

For : signed unsigned.

(V) OV overflow — carry flag [C]

(N) SF sign — Zero [Z]

x86	ARM
ADD Q	ADDS
⋮	⋮ Arithmetic.
changes the flags.	ADDS → operates on old flags

→ CMP b, a → computes a − b

	unsigned	signed
CF == 1	⇔ $a <_u b$	$a <_u b$
ZF == 1	⇔ a == b	⇔ a == b
SF == 1	⇔ $a <_s b$	⇔ $a <_s b$
OF == 1		

OF flag is set if 2's comp

$(a>0 \;\&\&\; b<0 \;\&\&\; (a-b)<0) \;||$
$(a<0 \;\&\&\; b>0 \;\&\&\; (a-b)>0)$

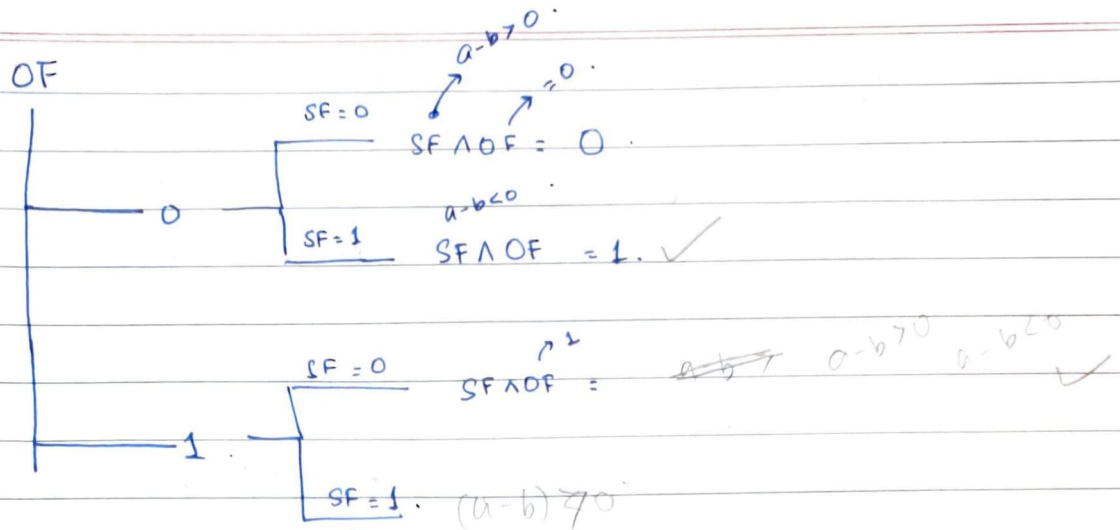
Claim : $a <= b$ ⇔ $(SF \wedge OF) \mid ZF$

$a <= b$ SF = 1 ZF = 0

→ equality :- $a == b \Rightarrow ZF = 1$.

$\therefore ZF = 1 \Rightarrow A == B$.

→ $a < b$ SF = 1 ZF = 0



Set X instruction : sets if satisfied
resets if not satisfied

- Basic layout :

Data
- Data section : \ . data
- N: .byte 10
- sum : .long

pgm start
- .text
- .global main

Squaring :

```
.text
    movl %ebx %ecx
    imul %ecx %ecx.
    ret
```

function ends.

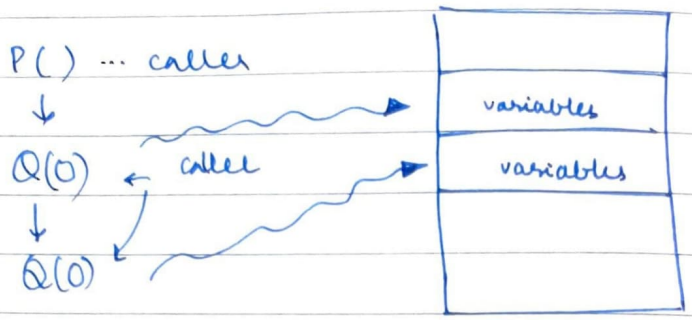
run , tui layout , layout , regs run

tui enable
run
show reg

Control-flow: (Function calls).

Local variables cannot be declared in the data section. ∵, in case of recursive calls this won't work.

Eg:



On stack at each activation

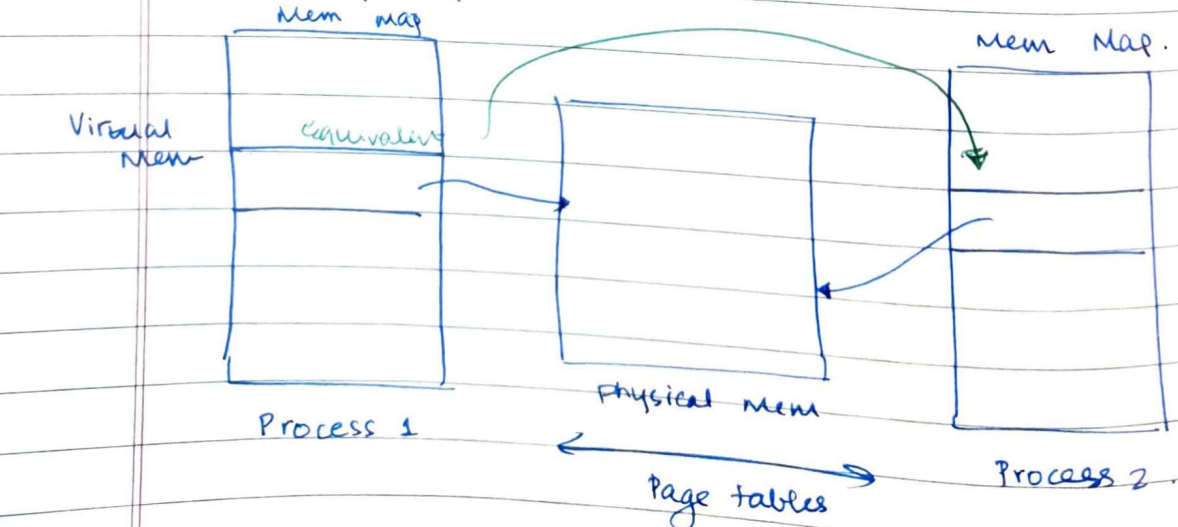
→ Scope:- within which (block) range the variable can be used.
Life:- How long the variable lives.

→ Memory Image Map slide

In case of ∞ recursive calls, at times the memory exceeds the space that is accessible.

Eg: It may enter a memory-mapped region for shared lib & gives seg-fault ∵ READ-only

→ For multiple processes :-



0x7fffffffdeaf

0x0000000000400495

Hello classmate

Date

Page

PROCEDURES & FUNCTIONS:-

→

ISA Primitives

Push	Call
Pop	Ret.

On using multiple abstractions there could be
- ~~performance~~ overheads.
- mitigates the performance.

- Handle control transfer
- Passing parameters.

Global var → always has the same address on repeated runs ∴ stored in data seg.

Local variables :- Stored in the stack, stack start is randomized.
[This is done to avoid buffer Overflow - attacks]
↓
read.

Passing parameters:

1. Calling library
2. x86 → calls square() in file.c.
   gcc compiler understands call square as the parameter is in %rdi
3. Conversion of file.c → file.s
   accepting arg

* In a jmp *(%eax) jumps to a value in %eax

→ Interrupts are asynchronous.

It is necessary to ensure that the return address is at the top of the stack when ret inst. executed.

ABI: Application Binary interface.

Default reg for variable parameters:- (6)
%rdi, %rsi, %rdx, %rcx, %r8, %r9

Passing 6+ parameters uses stack & also reduces performances ∵ arithmetic intensity red.

Eg: of a parameter of a struct, accessing multiple values with pointer.

- Stack
  - → keeps track of the functions running in a pgm
  - → when a function is called, a stack frame is created supporting the functions exec. (activation record)
    this includes local variables & parameters.
    + info for callee to return to caller.
  - → Grows towards lower memory.
  - → esp points to top.
  - → Housekeeping data → address of per stack frame + address of the next inst after exit
  - → function prologue: Saving %esp in %ebp
  - → When a function is called within another, the 2 stack frames form something of a linked list.

test a, → performs bitwise AND
Sets OF & CF flags are set to 0.
SF is set to most sig bit of AND

Procedures + Func:

→ Involves passing parameters & return values.
Allocate local ~~add~~ variable & deallocate.

Callee save : %ebx, esi, edi.
Caller save :- %eax, %edx %ecx.

→ Recursive calls :- Each call is assigned its own space on the stack.

Non-modifying code → Reentrant.
Modifying code → ~~#~~ Non-Reentrant.

Eg: Computing sum of first N numbers.

Modification is not allowed
in the linux os system text section is
READ-ONLY. (security).

text section

→ When code is compiled using -g option, compiler is less agressive (~~no~~ removes some optimisations) unlike when you compile using -O2, -O3 etc. These are called 'Heisenbug'.

→ Symbol table : Mapping memory & reg. (virtual mem

Heisenbug is detected when -O flag is utilized but may not be detected otherwise.

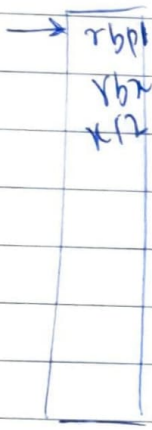
Conditional instructions use PC rel. addressing

When :-

```
foo () {
    [push %rbx]
    who() {
    }
    [pop %rbx]
}
```

→ Caller save

instead if:- Inside the who() func :-

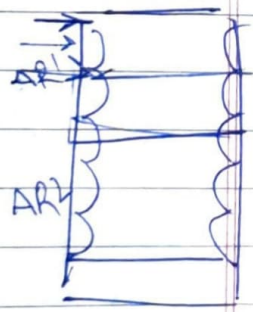



```
who() {
push %rcx.
    {
pop %rcx
```

→ Callee save

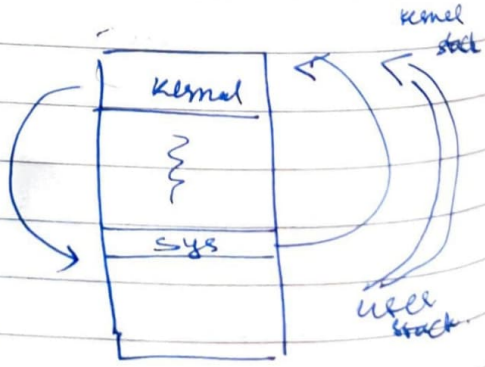
If it is a callee saved reg. the callee can use it shamelessly & vice versa.

Callee saved %rbx, %r12, %r13, %r14
%rbp, %rsp



syscall :-

Looks for the syscall no:
in %rax
%rdi
%rsi → address
%rdx → no: of bytes.



Switch statements make use of a jump table.

→ Encoding any instructions :-

4 bits	4 bits	8 bits	8 bits	8 bits
class	function	source	dest	Imm / Disp.

inst. code.

Eg :- addq %rA, %rB

60 ↑ ↑

0 6

Encoding: movq $0xabcd, %rdx

30 F2 abcd

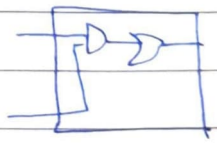
PUSH: A0 rA F pushes rA

POP: B0 rA F Pops rA

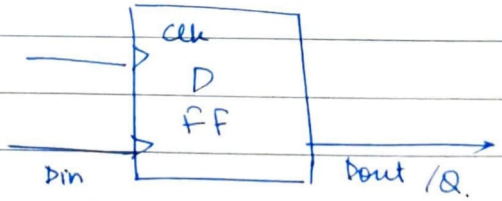
GOAL: Designing a processor.

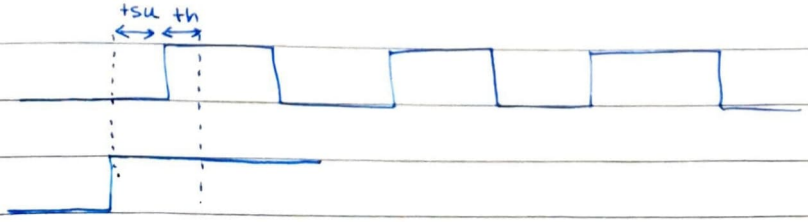
↓ Digital circuits.

Req: combinational circuits, sequential el
adder / multiplexers



D-flip flop.





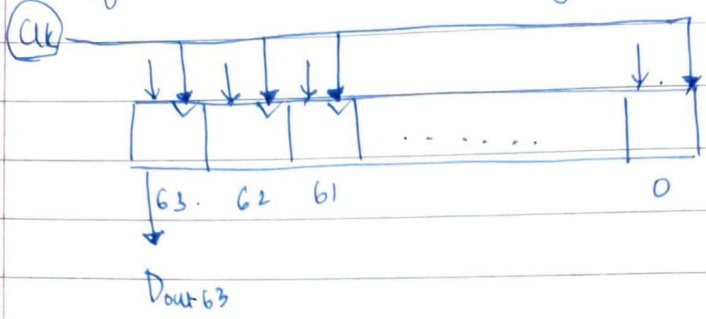
In the interval [t - tsu, t + th] signal must be stable

→ clock-2-Q time :- delay in obtaining the output

∴ Output ob. at :- $t + t_{clk-Q-2}$.

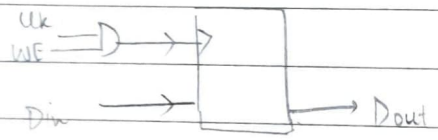
REGISTER FILE:

Q1 → Design a 64 bit register.

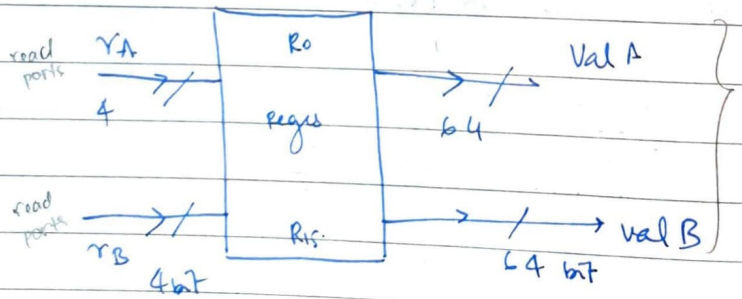


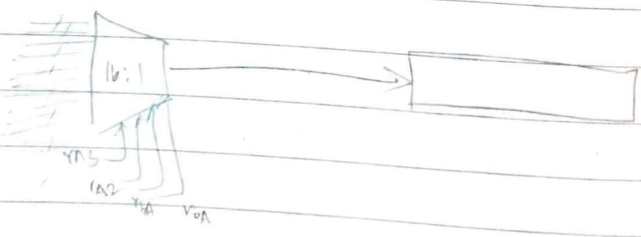
Q2 → Design a WE (write Enable) D-FF.

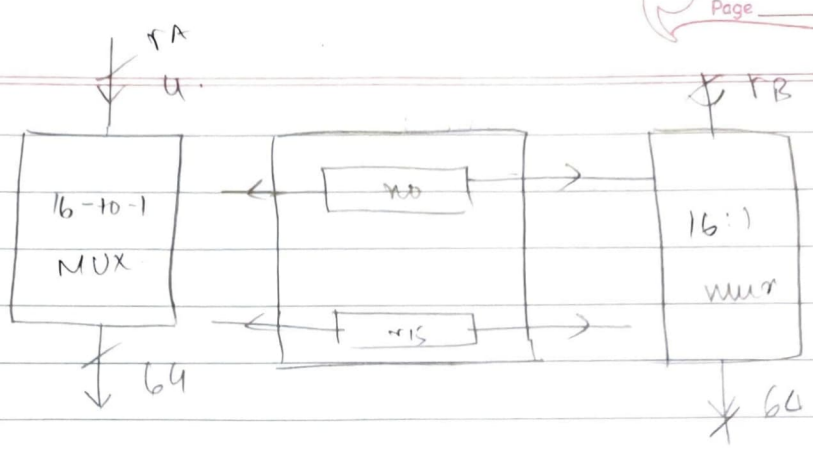
Method 1:



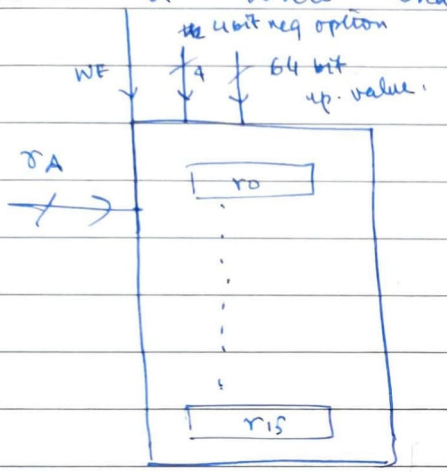
Q3 →



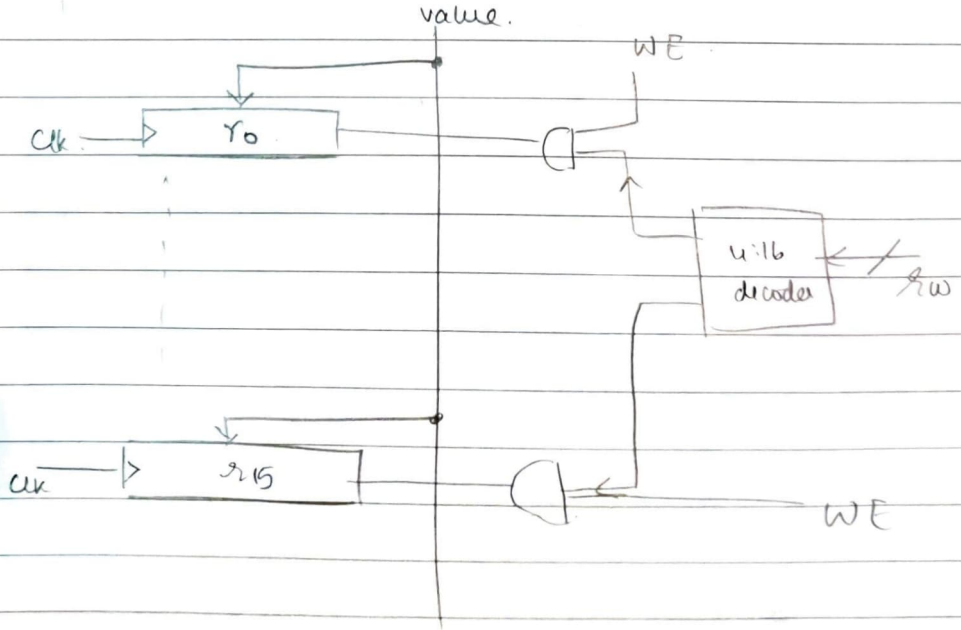




Q4 → Create a write enable in the registers :-



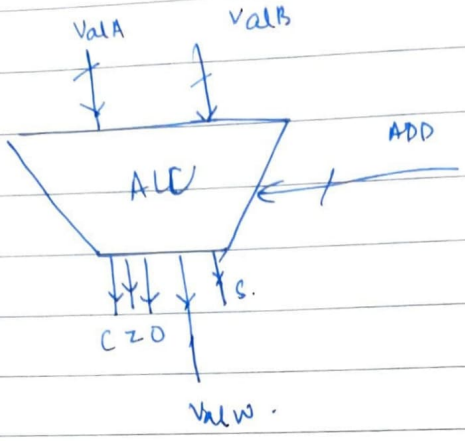
Naive: Will always update at least one register to avoid this scenario

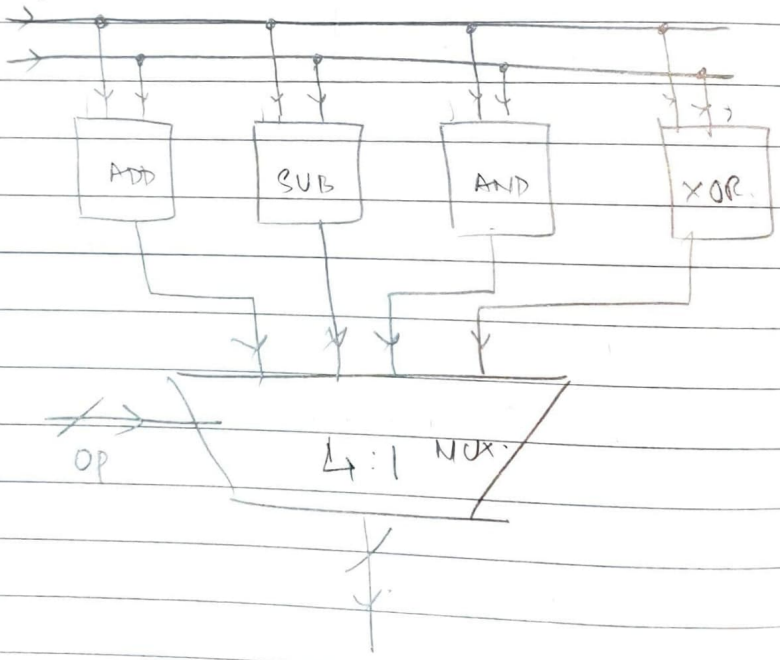


If the components aren't clock synced, then it may so happen that some wrong registers

values are updated.
ie: At each rising edge all required values for an updation are available.

ALU COMP Design an ALU to support



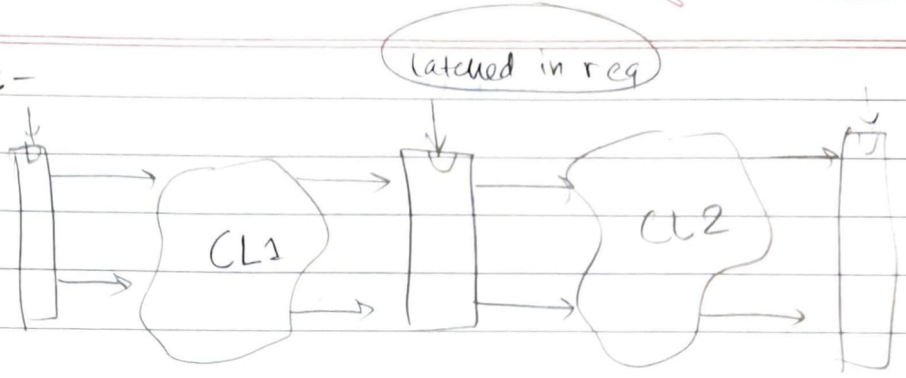


→ Assumption :- $clk \geq t_{cp}^{MUX} \geq \max(t_{ADD}, t_{SUB}, t_{AND}, t_{XOR})$

→ For a series of combinational circuit.

$$C \geq \max(t_{cp}^{1}, t_{cp}^{2})$$

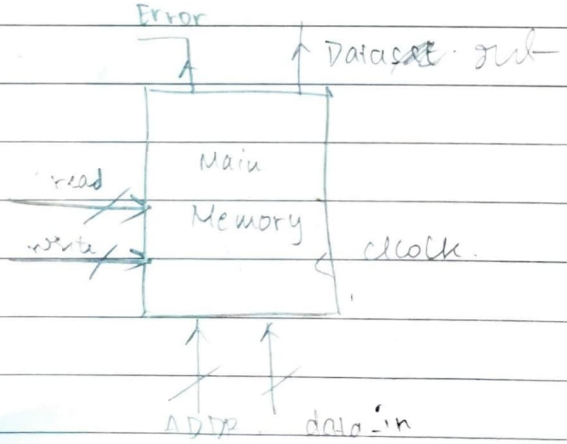
→ Where :-



→ When a function is called :-

- Its return addressed is pushed onto the stack
- If any caller saved reg are used they must be pushed onto the stack.
- Assign additional space on stack for local variables (if necessary)

● MEMORY MODULE :-



Assume data is in one memory location
inst. are in another memory loc

MEMORY HIERARCHY:

3 important components for any computational device
- ↳ Processor
- ↳ Memory
- ↳ I/O devices

Mem. is a linear array of bytes
Physical realisation varies from proc to proc
And each bit of a byte is realised using SRAM or DRAM technology.
- SRAM → 4–6 transistors
- DRAM → 1 trans + 1 cap
  - ↳ charged = 1
  - ↳ uncharged = 0.

Memory refresh cycles ensures state is maintained in a DRAM Eg: Cap. should'nt discharge

~~DRAM~~ & SRAM — volatile

Flash memory → loads the micro loader from the harddisk

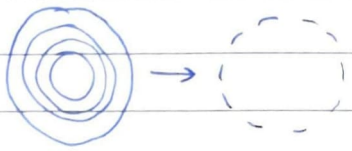
Processor ⟺ Memory protocol

- Send add onto add bus
- Enable read control signal
- Latch the value in the register

Proc
↕
Cache — intermediate
↕
Main Mem

Hard disk - collection of platters. (2 sided)

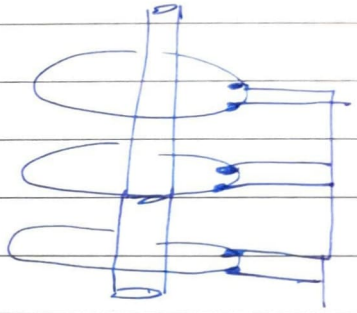
In each platter there are tracks
In each track there are gaps.
Each sector is a primary storage unit → atomic units.



→ Storage density = Recording den × track den.
To change any value → Read, Modify, Write

A sector of data ensures that it is either fully written or not written at all (cannot be partially written).

Every surface has its own R/W handle.



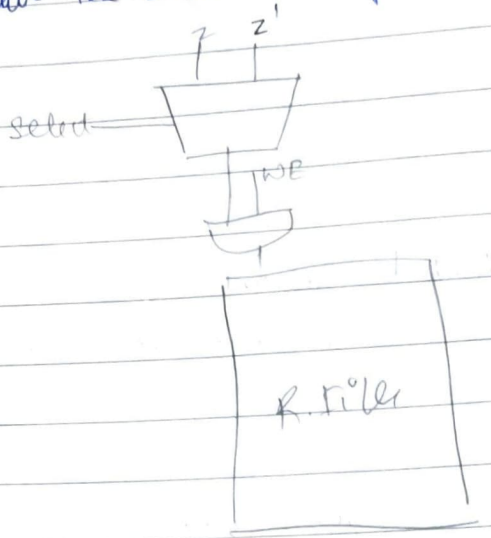
Q: What is the DT time if
R → RPM
K → sectors / track

→ Seek time : latency to move the R/W head
+ → Rotational latency
+ → Time for Data transfer.

Q: How to use cache to bridge the time lag b/w processor & memory.

SRAM: bistable. can have only 2 states. unstable in the remaining states.

With zones, each track is divided into zones. The tracks in a particular zone will have the same no: of sectors. (innermost).



→ Control transfer
→ Jump instruction

$T_{avg} = T_{transfer} + T_{avg\ seek} + T_{avg\ rotational}$

Two types of sorting
→ Internal, external sorting algorithm.

→ STEPS TO READ

① → Disk drivers / disk controller (1, 2, 3) + 7 tuple param

② → sector to be read detected on disk controller
→ through DMA it accesses the memory

③ → Sends an interrupt to the processor.

* Processor does not remain idle till then. It keeps working.

● Memory wall :-
Processor ⟺ Memory mismatch
Instructions & data cannot be supplied at the same rate.

Memory Hierarchy → Combine the 2

Sram — Expensive, Fast, small.

Dram — less exp., Slow, Large

Data items which will be accessed in the near future, put it in the cache.

What data is required by the user at the moment place that in the SRAM. Later useage ones place it in the DRAM

Instructions

lq: for (i=0; i<= 100; i++) (i)

→ Temporal locality. same instruction repeated

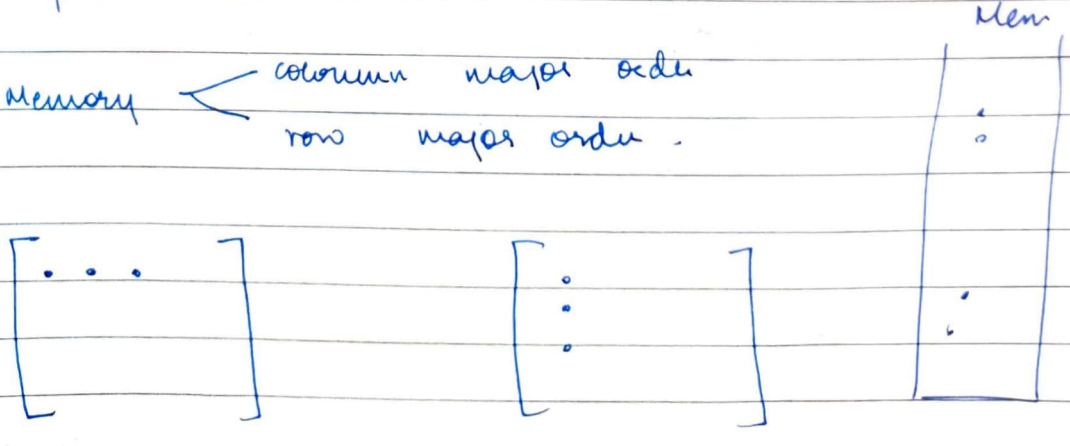
→ Spatial locality executing inst that are spatially close.

accessing A[0], A[1], A[2] etc.

Stride-1 memory access pattern: accessing array el. in order (difference of indices = 1).

For frequent accesses of variable optimum locations for its storage are either cache but preferably a register

Memory ← column major order
row major order.



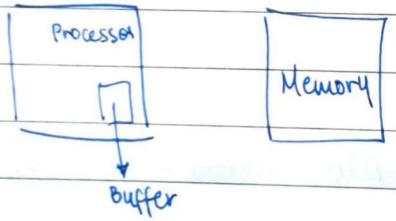
This is however a regular memory access pattern

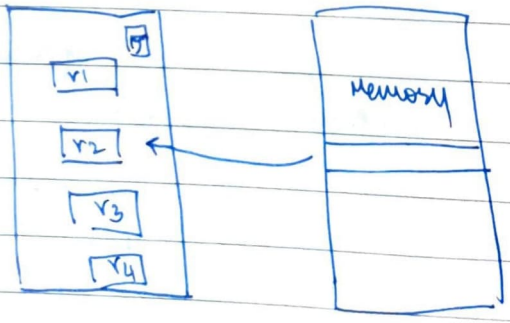
In case of a linked list, the memory allocated will be random & stride is non uniform. Irregular memory access.

Loop perm. trans :- compiler tries to get the best stride possible.

× ——— ×

→ Fetching data takes time. Significant effort. Hence maintain a buffer in the processor that holds the data for some interval of time. This is where caching comes into play.





Whenever an addressed is touched in any instruction, the address, the value & some meta information is cached.

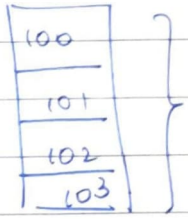
With simple hashing, the register to which the content must be transfered to is decided.

Assume 2^n registers :-
then we will need to consider the last n bits in the LSB of the address.

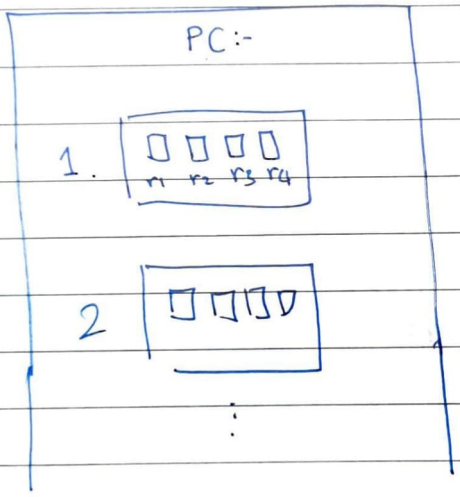
Eviction policy: kick out the already existing data if present.

Block size : no: of bytes we fetch in one-go.

eg:

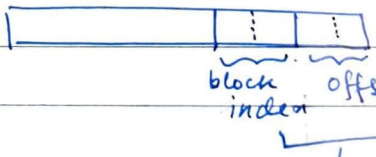


To ensure that the whole block of addresses the processors registers are also blocked



∴, By from an address (1 byte) of a block.

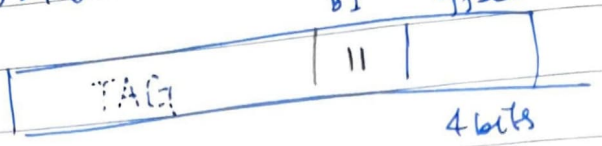
add:



depends on no: of registers.

In block addressing, it adjusts the block bounds depending on the fetched addressed.

Eg:- 32KB & 16 bytes.



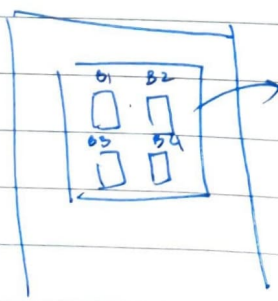
$$\therefore \quad \frac{2^5 \times 2^{10}}{2^4} = 2^{11} \qquad 11 \text{ bits}$$

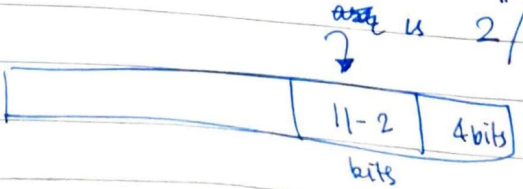
If BI & offset bits are same for 2 addresses there could be an error. Hence we first match the TAG bits.

In case of a collision :- linearly probe. set a max limit (associativity of cache) which is at max - how many blocks you will probe through.

In case all blocks (asso. cache) are occupied then use an eviction policy LRU . kick out the one that is least used by the processor

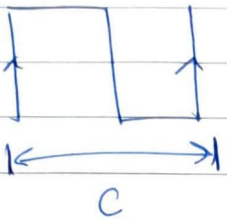
Q: 2MB mem 32KB cache 16 byte 4 way set associative





~~ass~~ is $2^{11}/2^2$ → essentially grouping ~~blocks~~ 4 blocks into 1.

- Single cycle design.

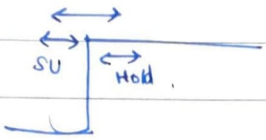


$C > t_I$.

Where $I \in \{ MOV, ADD, SUB \ldots \}$

Disad:- Some part of the clock cycle may go waste.

Eg:- t_{add} = fetch time + ALU delay + setup time



t_{memory} = Fetch time + address comp. delay + memory access + setup time.

- Multicycle design :-
Split the instruction into multiple steps

Eg: $t_{add} = t^1_{add} \cdot t^2_{add}$ } $C > max$

could include overheads → $t_{memory} = t^1_{memory} \cdot t^2_{memory} \; t^3_{memory}$ }

freq. increases. $\because C' = C/(K)$

Actually the new frequency becomes

$$C' = \frac{C}{K} + (\Delta)$$ → accounting for reg mov. etc.

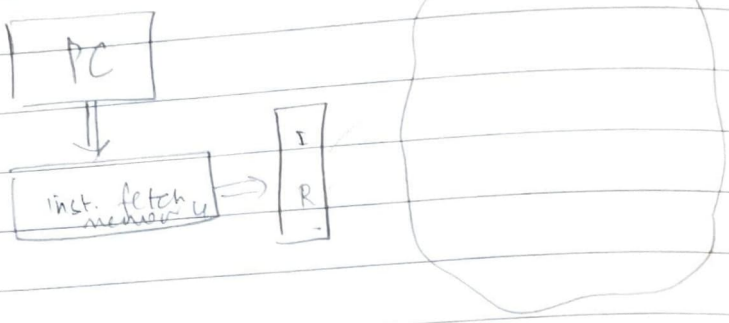
$10^6 \times C$ vs $4 * 10^6 \left(\frac{C}{4} + \Delta\right)$

$10^6 C$ vs $10^6 C + (4\Delta \times 10^6)$ ∴ 0

this Δ value ~~is~~ creeps in due to the set-up time
⇒ Processor w higher freq ⇒ better performance

- For any single instruction only, single cycle is better
- Where there are multiple inst. multi-cycle design

HW:-



- Microarchitechture :

i) Single cycle.
ii) Multi cycle.
iii) Pipelined. → embedded system (In order)
iv) Superscalar → desktop servers (out of order)
v) VLIW. → embedded.

If approach is ~~orde~~ out-of-order :-

start → mv mov (%.ram), (%.rbx) } ~~rando~~ out of order execution
ADD %.rcx, %.rdx
control → jmp

Error mv mov (%.rax), %.rbx
%.rbx, %.rdx

A window of 16/64 instructions continously reorders the inst. that must be executed at run time only. (done by hardware)
Static reordering could be done at compilation but requires more chip area & power.

Instruction scheduling → Static
Dynamic

Another issue could be the jmp instruction

Eg:-

```
add ....
sub ...
jz  label
```

Branch prediction strategy predicts the probability of picking one of the 2 branches

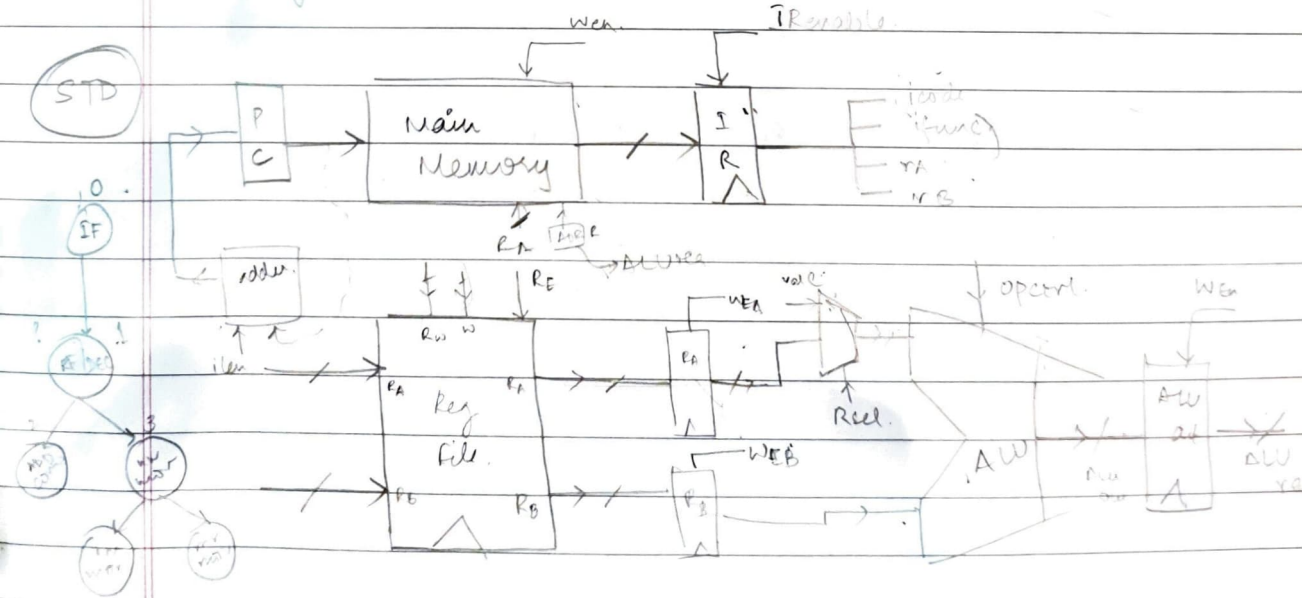
jmp ↙ ↘ do not jmp.

↳ does that what BP recomends.

Speculative execution

If the incorrect branch is taken the processor must restore its state. Wipe out the state done

Multicycle Processor:-



DATA PATH DESIGN.

→ We cannot update in I.F. cycle .. we won't know by how much the PC must be inc.

→ To optimize the PC inc. we can utilize the ALU itself (∵ multicycle)

3 – ADD, SUB, XOR,

4 – mr mov

5 – rm mov

3 – ir mov

jmp → 2

→ jmp instruction can be done in 2 clock cycles.

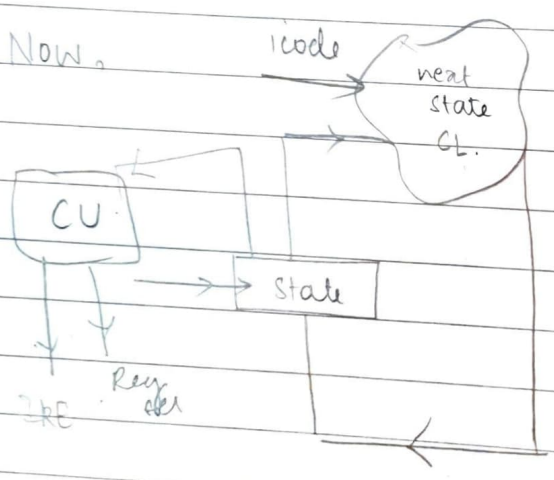
→ cmov → 3 clock cycles.
Reg to Reg

Single cycle : separate data & instruction memory
∴ it could lead to a structural hazard

but in multi clock cycle – 1 for IF, 1 for Data
∴ no problem

Advantage of Multicycle Proc: shorter clock cycles

→ Now,



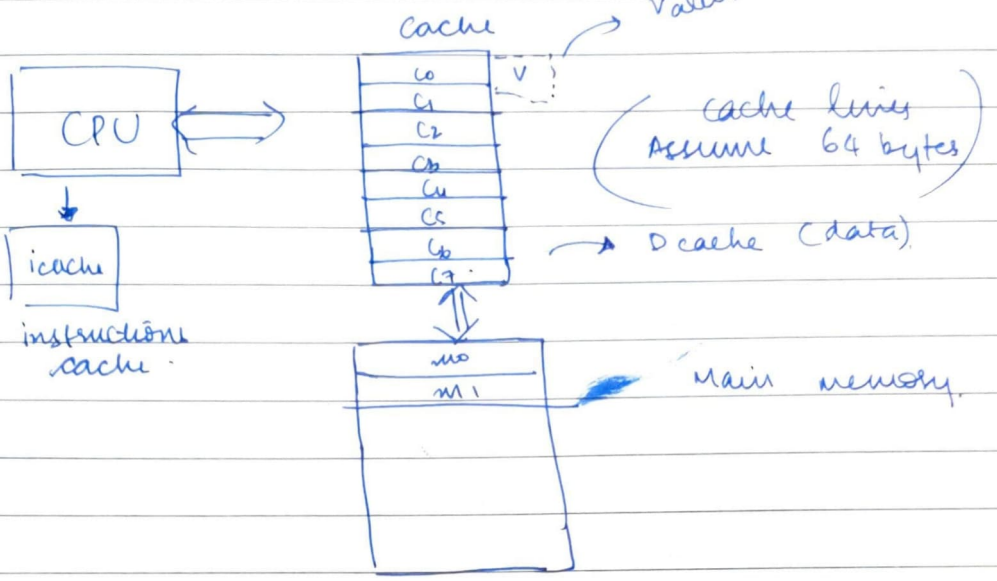
Control Path Design

CU :- 0
1

Read order mem

Eviction policy → LRU : least recently used
→ FIFO

→ SRAM memory → cache memory.
Main memory → DRAM.



Consider :- for (int i=0 ; i< 128 ; ++i).
sum += A[i] ;

A[0] A[1]
↓ ↓.
cache miss M0. [✓] M0.
[x]. cache line

Cache miss ratio : 16/128

"optimal method → farthest use in future".

cache misses incurred — cold misses, compulsory misses

→ Prefetching strategies
↳ Hardware. → Software
Strem stride (Irr. memory access patterns)

avoid compulsory

Stream & stride prefetchers fail miserably in case of linked lists.

Code 2:

```
for (j=0 ;  j < 32 ; j++) {
    for (i=0 ; i<128 ; i++) {
        sum += f(j, A[i]);
    }
}
```

For each iteration of j; the blocks accessed are: $m_0, m_1 \ldots m_{15}$
⇒ there are 16 misses.

Capacitive misses :- unintentional. Bootstrapping process of missing the 16 misses when j=0. But unfortunately cache size is small & hence there will be a miss.

Here $m_0, m_1, \ldots m_{15}$ (working set) > cache size

Here this conditions ⇒ capacitive misses will arise

Rectification : 1:- To reduce cap. misses interchange the loops

then the working set size becomes m_i.

Avoid capacitive misses

Direct mapped cache org.

C0 : m0, m8, m16
C1 : m1, m9 M · etc.

C = M / 8

So for

[x] i = 0 m0 . . . . . m0 (32 times)

i = 1 m0 . . . . . m0

⋮

i = 7 m0 . . . . . m0

use m0 8 times

[x] i = 8 m1 . . . . . m1 (32 times)

- **Rectification 2:**

```
for (j=0; j<32; ++j)
    for (i=0; i<64; ++i)
        fun (j, A[i]);
```

```
for (j=0; j<32; ++j)
    for (i=64; i<128; ++i)
        fun (j, A[i]);
```

Code 3:

```
for (i=0; i<32, ++i)
    sum += A[i] + B[i];
```

A[0] = { m0, m32 }

B[0] = { m8 - m11 }.

A[0]	B[0]	A[8]
m0 ↓ [x]	m8 [x]	[x]

Fully associative cache organization

Accessing A[1] is also a miss since currently m8 is present in C0.

Therefore all iterations are misses here. These are called conflict misses

Resolving this problem:- any main memory block can go to any block
FACC. (no conflict misses).
We cannot always have this as best method ∵, searching for which ~~case~~ cache block the mi cits in will be expensive to do it in small clock cycles. Hence we may have to do a fully parallel search. ∴, chip area & power cons. ↑.

→ 2-way set associative.

S0 - {C0, C1} - m0, m4, m8, m12.
S1 - {C2, C3} m1, m5, m9, m13

∴ No: of misses now will be 8.

Base address randomisation of arrays can ~~make~~ reduce conflict misses
Optimisation on one machine may not reflect on another.
Depends on cache sizes
Code portability exists, but performance portability does

Installing libraries causes a tuning library to sleep in which then enables the program to function well

Atlas/ Autotuning

Writes are asynchronous
Reads are synchronous

Write miss.

Write no-allocate

→ On missing a memory block write nothing, simply write into it dont upgrade.

→ Write through cache update cache line as well as the main memory

Write allocate

→ Write back
Just update the cache block. But on eviction totally rewrite that main memory block this track is kept with the help of the dirty bit.

Cache characteristics:
(E, S, m, B).

$C = S \times E \times B$

First identify the S then match tag bits then Offset of B block size.

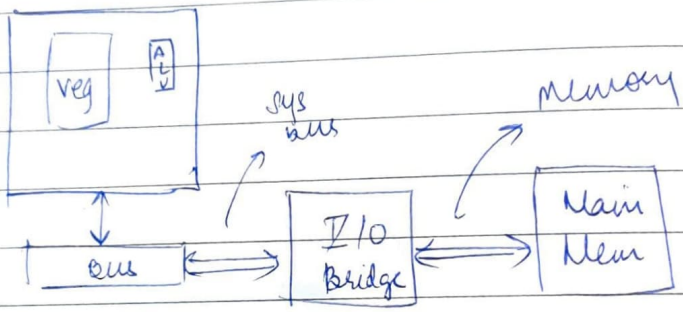
NOTES

1) SRAM - static : Exp, Fast, small
DRAM - dynamic : Slow, large, cheap

2) SRAM - bistable, can retain its state indefinitely as long as it is powered.

3) SRAM & DRAM - volatile ; ROM, flash - non-volatile

4) Basic CPU structure



5). $T_{access} = T_{seek\ avg} + T_{avg\ rot-} + T_{dt}$.

$$T_{avg\ rot-} = \frac{1}{2} \times \frac{60}{\text{given rot}} \times 1000.$$

$$T_{avg\ transfer} = \frac{60}{\text{given rot}} \times \frac{1}{500} \times 100$$

6) For the various disks are mapped to logic disc blocks. The firmware creates a tuple of 3 parameters (surface, track, sector) thereby accessing it. CA lookup table / mapping is maintained.

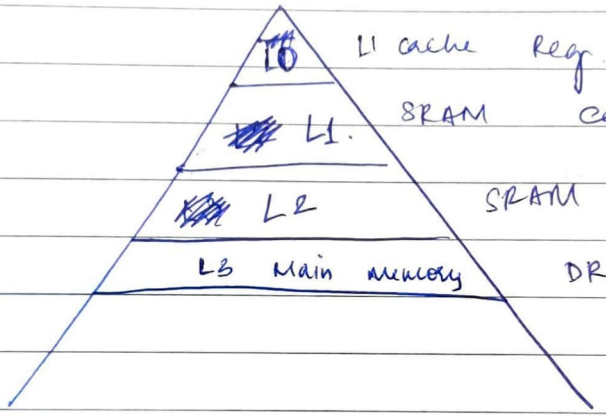
7) System bus & Memory bus are CPU specific

8) DMA: A process where a device performs a read/write on its own without the involvement of the CPU.

9) Principle of locality: Tendency to reference data that is near recently referenced or that was itself recently referenced.

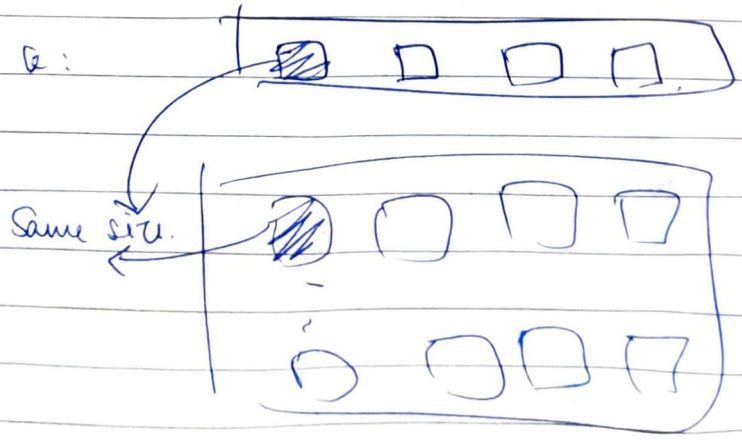
10) Smaller the stride ⇒ ↑ spatial locality.

11) Memory hierarchy :-



12) Each level in this hierarchy caches data in levels from lower hierarchy.

13) Cache blocks



→ Further, at any point in time the contents of blocks at level k are copies / subsets of k+1

14) A transfer unit is a block of unit that is used to move data back & forth b/w the levels k, $k+1$.

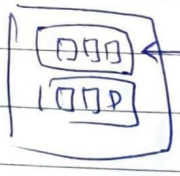
15) Data already cached at level k & to be read from level $k \Rightarrow$ cache hit else cache miss

16) Evicted block - victim block

17) Empty cache → cold cache.
~ Misses of this kind :- compulsory misses
~ Restrictive misses are :- conflict misses
~ Working set > cache size :- capacity misses

Problem 6.6 :

18) Terminology for how cache blocks are mapped
$S \rightarrow$ set



E - cache line.
b - offset within a set.

19) A cache with exactly one cache line per set is Direct Mapped Cache organization.

20) Steps involved in accessing/finding a word from cache/memory.
- Set selection
- Lines matched
- Word extraction

21) To check if a particular word is present or not the valid bit must be set & tag bits in the cache line match tag bits of word

22) The term thrashing describes when the cache is repeatedly loading & evicting the same set cache block

23) In fully associative ~~cache~~ cache org. there are not set index bits

24) Write through caches - write no-allocate
Write back - write allocate.

25) Cache that holds both instruction & data is called unified cache.