6/1/25

CS2007

50 % ⟶ Endsem

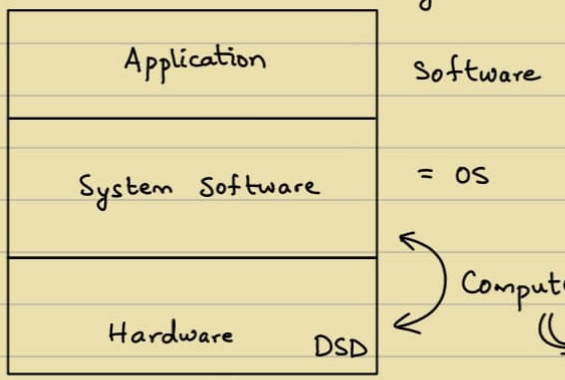
20 % ⟶ Project / Assignment

30 % ⟶ Midsem

Core ⟶ Revelance & Importance

↳ Stream

Electives



Computer Organization

Computer Architechture

Application

Function of Application

Control hardware and allow to run over it.

(Cannot access hardware directly)

↳ Only through system software

Execute programs — Application (OOPS/C)

Design program — How to solve problems

↓

[Application]

(DAA, TOC)

↳ Theoretical Computer Science

Alan Turing

↳ What should the computer do

Prerequisities : Combinatrics, Graph Theory

GPU + System software ⟶ Machine Learning

↳ Hardware

Computer Architechture

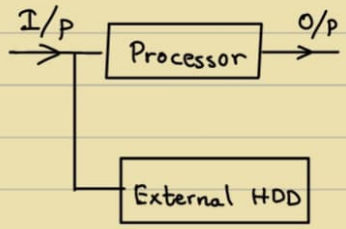
Strutural → Load

Architechture → Design & Properties

X functionals components in the computer — Structural

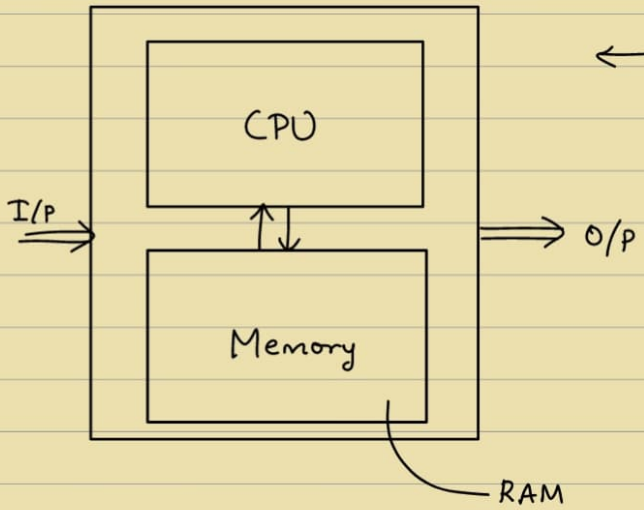
How they are placed — Architechture

↳ Computation



I/P & O/P → Seperate components

↳ Not part of Main processing unit



← Von Newman Architechture / Stored memory Architechture

a = 5 ← Temporary variable

b = 6

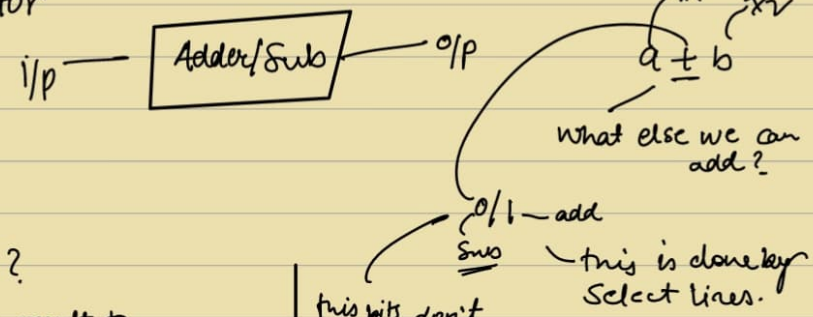
a+b → Stored in Registers

ALU : Arithmetic and Logical Unit

CU : Control Unit

8/1/25

calculator



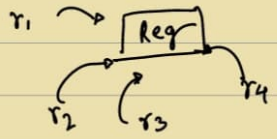
but what for a+b+c+d+e?

we need to store a+b result to add with c and so on.

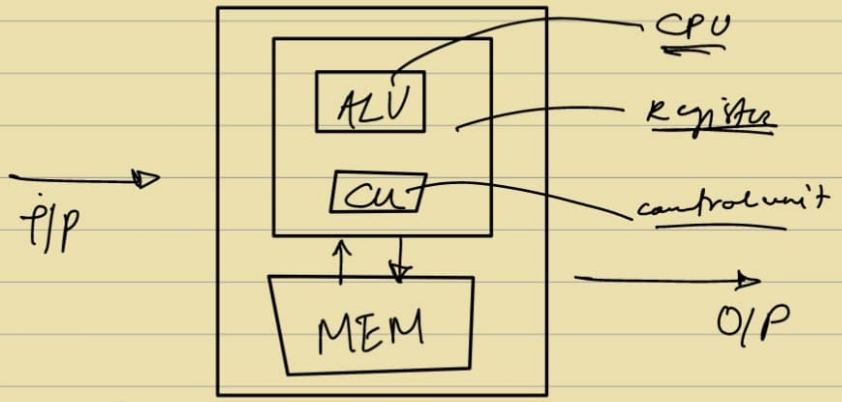
∴ we use register for this temporary storage

at most 16 bits are required to store the data $a+b \rightarrow r_1$, $r_1+c \rightarrow r_2$

$r_2+d \rightarrow r_3$, $r_3+e \rightarrow r_4$



processor or RAM which is imp?



* John Von Newman.

He used to work in chaotic kind of environment.

- Memory
  - Data [int a=5;]
  - Program [Set of Instructions]

- ALU : Circuits for operations which are performed
  → Very fast compared to main memory

- Registers : Fastest memory , Stores Temporary Data (Intermediate data)
  Smallest size , e.g. Sequence of bits / flipflops

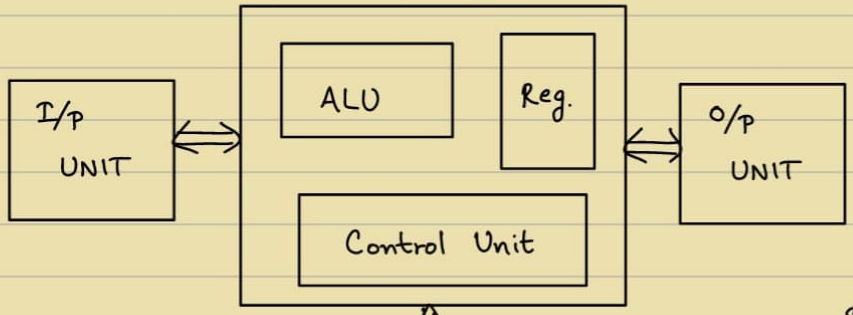
- Control Unit :
  (i) Timing Signal : Order of execution of instructions
  (ii) Control Signal : Control the read/write operations on registers .

- Main components are attached by bus.
  ↳ Implemented by Multiplexers

→ Word : Memory representational Unit (Along with bytes)
  Byte → Fixed
  Word → Multiple bytes

→ Accumulator : Intermediate data is stored temporarily

9/1



System bus - Connects

Memory — Internal Memory / RAM / Primary
[Capacity]

ROM

Primary Secondary

H.W: What is it made up of ?

Hard disk

SSD : Solid State Device

Data goes through RAM, cannot be taken directly from Harddisk.
(Memory → Referred as RAM)

F ↓ S

Reg
Cache
RAM

cost ↓ cheap

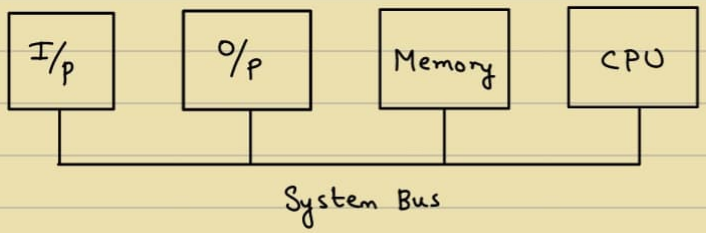
SDD

HDD

Magnetic Disk

Bottleneck - Hinderance

- System bus :



System Bus

Data Bits | Control Bits | Address bits

```
int *p ;
int a = 65 ;
p = &a ;
```

a + b (a, b → Variable)

a → Data Bus

\+ → Control Bus

b → Data Bus

Fetch the data from location & Transfer

Where to fetch from : Address Bus

What to fetch : Data Bus

Hard disk - External Memory

↳ Not connected to System Bus

- Connected via SATA cable

System Bus : Embedded into the chip itself

↳ Hardwired into processor

∴ Very fast

Performance Metric : Clock speed

↳ i.e. no. of instructions in one time unit (Hertz)

P : Min. time taken for one instruction

R : Rate ($1/P$)

High Level Computer Language

↓

Assembly

↓

Machine language → Checking speed

Instruction : Basic unit, to tell the processor what to do

Parameters :

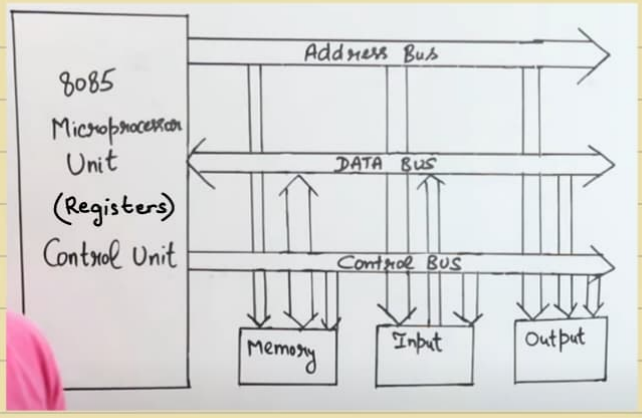
N : No. of instructions executed

M : Avg. time of an instruction

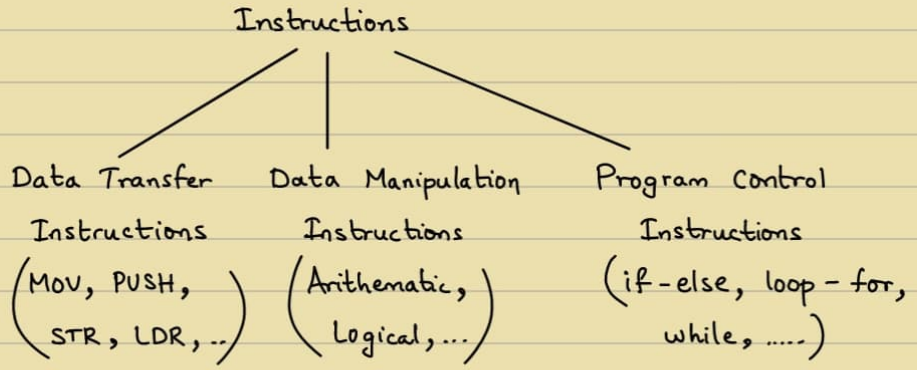
Cache → Permanant in memory but may/may not be used.

- Size : Less than main memory i.e. subset of main memory
- Whatever is frequently accessed
- Consists of levels - Level 1, 2, 3, Not used beyond 3.

→



→



- Load : Data is loaded into register from memory
  Store : Data is stored into memory from registers.
  Move : Move data from register / address to register

lscpu
lsmem
} Commands (Ubuntu-based)

Cache → Cannot be used instead of main memory.
↳ Temporary

Main Memory	Level 1	Level 2	Level 3

[Ref] TLP : Translational Lookaside Buffer

10/1

Server — More powerful system than computers

Params :

Throughput : Set of programs / tasks to be completed in a point of time

Response time

Performance Metrics :

(i) Clock Cycle time → Capacity of the Processor [Hardware]

Clock cycle : Positive half cycle + Negative Half cycle

CPU time = no. of programs x Clock cycle time

[Program = n x no. of instructions]

$$\Rightarrow \text{CPU time (sec)} = \frac{\text{no. of instructions}}{\text{per program}} \times \frac{\text{no. of cycles}}{\text{instructions}} \times \text{clock cycle time}$$

CPI : Cycles per instruction [Architechture]

Each instruction — Average of Instruction Set.

$$\frac{\sum_{i=0}^{n} P_i + n}{n}$$

Moore's Law : Size of a chip / transistor halves every 18 months.

↳ Intel

[Quantum Computing is independent]

i.e. same job can be done by a smaller semiconductor chip.

Python - User friendly not System friendly

(no. of instructions per program) ↑

∵ High level Language

CPU is not only dependent on the Hardware capacity.

Multi-processing : Parallel Computing

$$\text{Speed up} = \frac{\text{without parallelism}}{\text{with parallelism}}$$

[Divide the work & work]

Concurrency ⟶ Resources can be parallel

Parallelism ⟶ different tasks being done at the same time

- Degree of Multi-programming ⟶ Read [diff. b/w Parallelism & Concurrency]

⟶ Pipelining : Not in Mid Sem

13/1

Intel : Processor (Chip manufacturing company)

Xilinx : ARM, Processors, FGPA

[Virtual Machine Types offered by AWS (18 types)]

- CISC : Complex Instruction Set Computer

  RISC : Reduced Instruction Set Computer

x86 ⟶ 8086

c64 – Alternative

x → Intel & 8086 → Version

↳ Some value ↳ Intel Processor, comes under CISC.

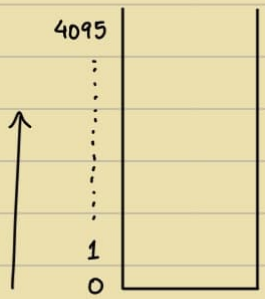
High level language

↓

(Add x_1, x_2) Assembly Language [8086]

↓

Machine Language



→ RAM

Byte based : Point out every byte in memory.

Word based : Point out every word in memory.

↓

Can be 2-3 bytes

Location of 0 → 8 bytes

Location of 1 → 8 bytes

⋮

Location of 4096 → 8 bytes

$2^{12} \times 2^3 = 2^{15}$ bits

Locations - Undivided further

Byte Addressing : 0 | A | B | ✗ Cannot be addressed individually

1 | B | | ✓

2^{12} ⟶ 12 bits ⟶ Address

∴ Address ⟶ 12 bits

In Word Addressing system,

word size = 2 bytes

∴ Cannot go less than 2 bytes :

$\frac{4096}{2} = 2048 \longrightarrow 2^{11}$

∴ Need 11 bits to represent

More memory ⟶ byte based Addressing becomes a problem.

A B

ADD 2086 , 2088

[2086 , 2088 ⟶ Addresses]

Processor fetches the operations.

CISC ⟶ Complex

Less instructions ⇒ More Load onto CPU

⇒ Flexibility ↓

More clock cycles

RISC ⟶ Simple

LOAD R1 , 2086 (RAM to Register)

LOAD R2 , 2088 [load data into register]

ADD R1 , R2

STORE 2081 , R1

Powerful Compiler ⟶ RISC

Humans manually fetch the operations.

More instructions ⇒ Less clock cycles

[General Case]

4 Operations in RISC = 1 Operation CISC → Easy for us, Complex for processors

Step - by - step

Automatic

4 Clock cycles (Assuming 1 operation takes 1 clock cycle.)

Not 1 clock cycles. Requires more than 1 clock cycle.

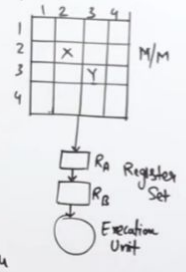
"CISC"

1) Complex Instruction Set Computer
2) Large Number of Instructions
3) Variable length Instruction format
4) Large No. of addressing modes
5) Cost is High
6) More Powerful
7) Several Cycle Instructions
8) Manipulation directly in Memory
9) Microprogrammed Control Unit
10) Examples: Mainframes, Motorola 6800, Intel 8080

MULT 2:2, 3:3

"RISC"

1) Reduced Instruction Set Computer
2) Less No. of Instructions
3) Fixed length Instruction format
4) Few no. of AM
5) Less cost
6) Less Powerful
7) Single Cycle Instructions
8) Only in Registers
9) Hardwired Control Unit
10) MIPS, ARM, SPARC, Fugaku




```
LOAD A, 2:2
LOAD B, 3:3
PROD A, B
STORE 2:2, A
```

- ARM : Advanced RISC machine.

Easier to design : CISC

Only Intel → CISC [∵ 1986 → Software was not powerful]

Converting RISC to CISC is possible, but not useful.
∵ Processor has the same load
Converting CISC to RISC is better.
i.e. High to Low

Processor → Computing Intensive
RAM → Memory Intensive.
[GPU → Process parallely, ∴ Reduce load on CPU
(CPU + GPU → Gaming)]

Non Parallelised → CISC
Efficiently Parallelised → RISC
∵ 1 operation → Many operations [1 Operation → 1 Clock cycle
∴ Easier to divide)
CISC can be parallelised but difficult for us.
∴ Not efficiently Parallelised.

16/1

MASM → .asm
↳ Assembly code
↳ Tool in which it is compiled

Debugging → Memory-based error
↳ Debugs line-by-line

ARM → Advanced RISC Machine

Program → Set of Instructions
Assembler → Makes sure Assembly language is executed on processor.

Without math.h :
```
gcc file.c -o file
./file
```

With math.h :
```
Add -lm
Add -ld
```

Assembly Language - Written precisely
↳ Assume running on processor, no memory involved.

ARM Processor ⟶ 32 bit processor

↳ Word size, i.e. 2^{30} words can be possible.

30 Addresses

RAM — 2^{30}

↳ Primary Memory / Main Memory / Memory

[Variation of ARM : MIPS, X86]

Contains 16 Registers

Program Counter, Stack Pointer, LR, r0, r1, ..., r12

↳ Store the address of the locations

↳ Next instruction be loaded is stored.

RAM

↙ Store the instructions

↘ Stores the Data

Assembly Language - Generic Term

↳ Instruction set

- Operations :

  - Arithmetic operations :

    (1) ADD

    (2) SUB

(1) ADD :

Operands : r_1, r_2, r_3 [Registers]

$r_1 = r_2 + r_3$

(2) SUB :

$r_1 = r_2 - r_3$

} Instruction Set Specific to ARM Processor

Ex. AND r_1, r_2, r_3 ≡ $r_1 = r_2 \& r_3$

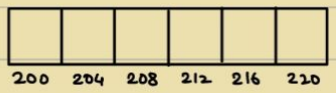
LDR $r_1, [r_2, \#40]$

↳ $r_2 + 40$ [r_2 : base address]

Load ↳ Register r_1

∴ $r_1 = [r_2 + 40]$

Content of r_2+40 = data of r_1



IntArray

↳ Base Address

Automatically 4 bytes are loaded.

LDR : Load Register → Assigning value to the register

STR : Store Register → Store memory content in the address

↳ Reverse of LDR

For loading only single byte,
LDRB is used.

STR r_1, [r_2, #40]

17/1

Lab :

```
gcc -g -o sample sample.c
```

./sample → Running the code

gdb ./sample → Debugging CLI [gdb : gcc debugger]

run → Run the code blw specified breakpoints & run the code entirely if no breakpoints mentioned

n → Execute the line, display the next line and get ready to debug the next line.

s → Step-into, Works only if the debugger is currently on the line where a function is called.

c → Continue onto the next breakpoint and get ready to debug

b <line number> OR b <function name> → Create a breakpoint at specified LOC

clear → Delete all breakpoints

clear <line number> → Delete the breakpoint at specified LOC.

display <variable name> → Display the value of variable & continue displaying it in the next debugging lines.

print <variable name> → Display the value of variable

quit → Exit from the debugging CLI.

bt → Print backtrace of all stack frames, or innermost COUNT frames

```
advance -- Continue the program up to the given location (same form as args for
break command).
attach -- Attach to a process or file outside of GDB.
continue -- Continue program being debugged, after signal or breakpoint.
detach -- Detach a process or file previously attached.
detach checkpoint -- Detach from a checkpoint (experimental).
detach inferiors -- Detach from inferior ID (or list of IDS).
disconnect -- Disconnect from a target.
finish -- Execute until selected stack frame returns.
handle -- Specify how to handle signals.
inferior -- Use this command to switch between inferiors.
interrupt -- Interrupt the execution of the debugged program.
jump -- Continue program being debugged at specified line or address.
kill -- Kill execution of program being debugged.
kill inferiors -- Kill inferior ID (or list of IDs).
next -- Step program, proceeding through subroutine calls.
nexti -- Step one instruction, but proceed through subroutine calls.
queue-signal -- Queue a signal to be delivered to the current thread when it is
resumed.
reverse-continue -- Continue program being debugged but run it in reverse.
reverse-finish -- Execute backward until just before selected stack frame is cal
led.
reverse-next -- Step program backward, proceeding through subroutine calls.
--Type <RET> for more, q to quit, c to continue without paging--
reverse-nexti -- Step backward one instruction, but proceed through called subro
utines.
reverse-step -- Step program backward until it reaches the beginning of another
source line.
reverse-stepi -- Step backward exactly one instruction.
run -- Start debugged program.
signal -- Continue program with the specified signal.
start -- Start the debugged program stopping at the beginning of the main proced
ure.
starti -- Start the debugged program stopping at the first instruction.
step -- Step program until it reaches a different source line.
stepi -- Step one instruction exactly.
taas -- Apply a command to all threads (ignoring errors and empty output).
target -- Connect to a target machine or process.
```

$r_0, r_1, \ldots\ldots, r_{12}$, PC, SP, LC

- Arithmetic ADD $r_1, r_2, r_3 \longrightarrow r_1 = r_2 + r_3$
  SUB $r_1, r_2, r_3 \longrightarrow r_1 = r_2 - r_3$
  Logical AND $r_1, r_2, r_3 \longrightarrow r_1 = r_2 \,\&\, r_3$ [Bitwise]
  OR $r_1, r_2, r_3 \longrightarrow r_1 = r_2 \mid r_3$ [Bitwise]

- MOV r_1, r_2 ← Syntax, i.e. Content of r_1 is in r_2.
  MOV $r_1, \#42$ ← Example

  MOV r_2, r_1 [Copy command]
  (from register to register)

- Loading content from the memory (location) specified in RHS into the register:

  LDR $r_1, [r_2, \#42]$
  (Load Register)
  → Number

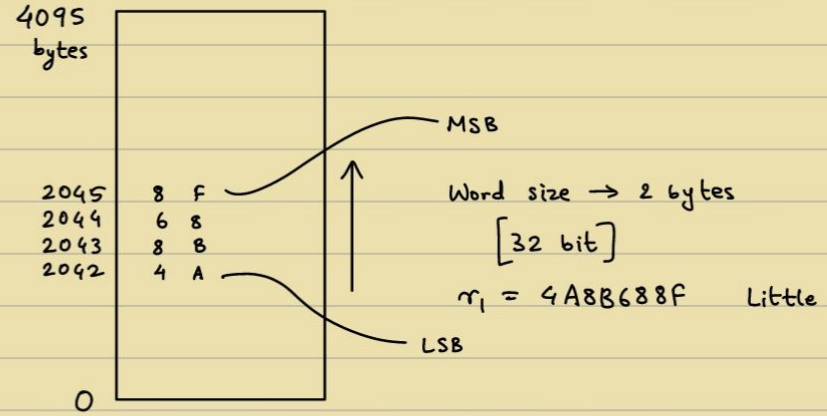
  Address = Base + Offset
  (x) (10)

  If $r_2 = 40$,
  $r_2 + 42 \longrightarrow 82$
  → r_1

  1 digit of Hexadecimal number → 4 bits
  ∴ 2 digits of Hexadecimal number → 8 bits
  ∴ 8 digits of Hexadecimal number → 32 bits

  Memory → Always in bytes

  Word size → How we access the content



LDRB $r_1, [r_2, \#42]$
→ Copy one byte of data from the memory location

$r_2 = 2000$
$r_2 + 42 \longrightarrow 2042$

- STR r_1, [r_2, #42]

If $r_2 = 2000$,

$2000 + 42 = 2042$

Copy the content of r_1 to 2042

↳ 4 bytes

STRB r_1, [r_2, #42]

↳ Copy one byte at a time

Q) Add two numbers

48, 62

Sol: MOV r_1, 48

MOV r_2, 62

ADD r_3, r_1, r_2

H.W

Q) a + b = c

↳ 20 bit (Memory location)

↳ 18 bit (Memory location)

Base = 2000 } a

Offset = 48 }

b → Immediate location [Ex. a → 2042, b → 2046, c → 2050]

Write a code to add.

c → Offset = 50

20/1

Sol: MOV r1, #2000

LDR r2, [r1, #48]

↳ 2048

LDR r3, [r1, #52]

ADD r4, r3, r2

STR r4, [r1, #56]

2000 is just a number, but in this context, it is an address.

$r_2 = a$

$r_3 = b$

→ ORR r_1, r_2, r_3

MVN r_1, r_2 : r_1 = bitwise negation of r_2

LSR → Left Shift (Cannot be used independently, use MOV or ADD)

ADD r_4, r_3, r_2, LSR #2 $\Longrightarrow$ $r_4 = r_3 + (r_2 >> 2)$

MOV r_4, r_3, LSR#2

[LSL : Left Shift
LSR : Right Shift]

Q) Base address ⟶ 2000

Offset ⟶ 48

d = a + b + c

Sol:

```
MOV r1, #2000
LDR r2, [r1, #48]
LDR r3, [r1, #52]
LDR r5, [r1, #56]
ADD r4, r3, r2
ADD r2, r4, r5
STR r2, [r1, #60]
```

- Comparison:

```
CMP r1, r2
```

↳ Should be used with a label

EQ, GT, GE, LE, LT, BEQ ↳ Branching

⇓ Equal ⇓ Greater than ⇓ Less than or equal to

Ex.
```
CMP r1, r2
EQ label
```
} ⟶ Outputs into label if $r_1 = r_2$

↳ To code efficiently

↳ Instruction / Instruction set

Q) if (a < b), add a + b + c

else, b + c

Sol:

```
MOV r1, #2000
LDR r2, [r1, #48]
LDR r3, [r1, #52]
LDR r5, [r1, #56]
ADD r4, r3, r2
CMP r1, r2
GT label
label : STR r4, [r1, #60]
ADD r2, r4, r5
STR r2, [r1, #60]
```

[GNU ARM Tool]

20/1

- MOV r_1, #4
  MOV r_2, #6
  CMP r_1, r_2 ⟶ Flags get activated.
  BEQ LABEL
  LABEL : ADD r_3, r_1, r_2

- MOV r_1, #4
  MOV r_2, #6
  CMP r_1, r_2
  BNE FIRST
  FIRST : ADD r_3, r_1, r_2

MOV r_1, #4
MOV r_2, #6
CMP r_1, r_2
BNE EXIT
FIRST : ADD r_3, r_1, r_2 (Optional)
EXIT :

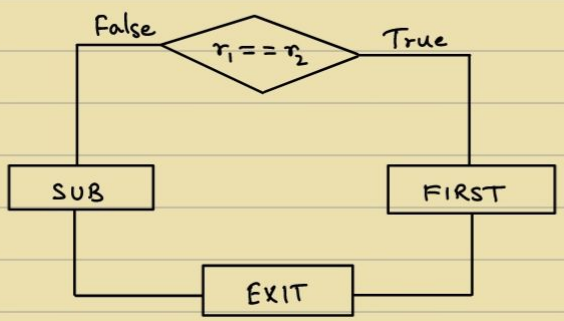
BNE : Not equal to
↳ Branching

MOV r_1, #4
MOV r_2, #6
CMP r_1, r_2
BNE EXIT
FIRST : ADD r_3, r_1, r_2
EXIT : SUB r_3, r_1, r_2

If a == b, both the FIRST label and the EXIT labels will be executed sequentially.

Fix :

MOV r_1, #4
MOV r_2, #6
CMP r_1, r_2
BNE EXIT
FIRST : ADD r_3, r_1, r_2
B EXIT
SUB : SUB r_3, r_1, r_2 ; comment using semicolon
EXIT :



Q) a = b = 2000

offset for a = 24

offset for b = 28

```
if (a > b)
  a >> 2
else
  b << 2
```

a = b = 2000

offset for a = 24

offset for b = 28

Offset for c = 32

```
if (a > b)
  c = a >> 2
else
  c = b << 2
```

Sol:

```
MOV r1, #2000
LDR r2, [r1, #24]
LDR r3, [r1, #28]
CMP r2, r3
BGT RIGHT
BLE LEFT
RIGHT : MOV r4, r2 LSR #2
B EXIT
LEFT : MOV r4, r3 LSL #2
EXIT :
```

```
MOV r1, #2000
LDR r2, [r1, #24]
LDR r3, [r1, #28]
CMP r2, r3
BGT RIGHT
BLE LEFT
RIGHT : MOV r4, r2 LSR #2
B EXIT
LEFT : MOV r4, r3 LSL #2
EXIT :
STR r4, [r1, #32]
```

$\begin{pmatrix} r_2 = a \\ r_3 = b \end{pmatrix}$

Q) $r_1 \longrightarrow \#4$

$r_2 \longrightarrow \#9$

$r_3 \longrightarrow \#92$ (offset)

Write Assembly code for:

```
while (r1 != r2) {
  r1++
  r3++
}
```

Sol:

```
MOV r4, #2000
LDR r1, [r4, #4]
LDR r2, [r4, #9]
LDR r3, [r4, #92]
MAIN : CMP r1, r2
BNE BLOCK
B EXIT
```

```
BLOCK : CMP r1, r2
ADD v1, v1, #1
ADD v3, v3, #1
B MAIN
EXIT :
```

Lab: [20/1]

disass <function_name> : Shows assembly code
info f : Give details of the frame
frame <function_name> : Give details of the frame for specified function
layout src : Shows an interface
info register / i r : Show details of all registers
print $<register name> : Displays value of specified register
disassemble <function_name> : Shows assembly code for the function
info frame : Information about current frame

21/1

Q)
```
k = 4
do {
  k = k+1
}
while (k < 12)
```

Sol:

```
MOV r1, #4
ADD r1, r1, #1
MAIN : CMP r1, #12
BGE EXIT
ADD r1, r1, #1
B MAIN
EXIT :
```

[OR]

```
MOV r1 #4
WHILE : ADD r1, r1, #1
CMP r1, #12
BLT WHILE
```

→ Registers :

r0, r1, r2, ..., r12, pc, sp, lr

(r0, r1, r2, ..., r12 → General Purpose registers)

Stack pointer : Stores the starting address of the stack

```
4040  MOV  r4, #2000
4044  MOV  r1, #3
4048  MOV  r2, #2
4052  ADD  r1, r2, r3
4056  BL   #5000
```

SUBTRACT :

```
5000  SUB  r1, r2, r3
5004  STR  r1, [r4, #3]
      MOV  pc, lr
```

[BL : Branch Link]

Current instruction's next instruction address is stored in lr.

PC → 5000 (Actual Next instruction)
LR → 4060 (Supposed to be Next instruction)

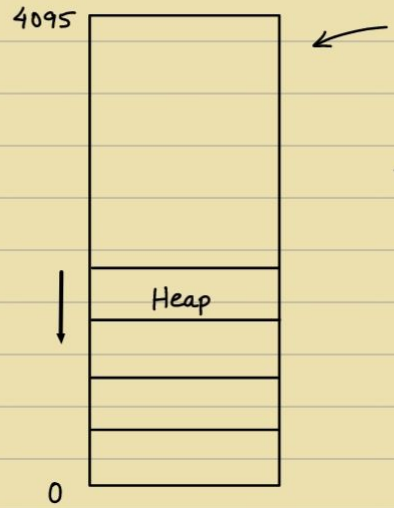
Procedure call → Set of Instructions
Label → Sequential code in seperate memory location
↳ Used for looping

- Local variables :

r_0, r_1, r_2, r_3 → Passing the arguments conventionally, as reserve.
$r_4, r_5, \ldots, r_{11}$ → Store the value and give it back conventionally.

Stack → Required when registers are not available
↳ Temporary → Existing registers are stored in stack.
Restored into registers when needed.

Stack pointer : Points to top of the stack



* when sp updates → top will decrease
4000 → 3996 → 3992

22/1

```
A() {          lr = 42        B() {
  int a = 8;                    int a = 4, b = 100;
(76) B();      lr = 80          a = a + b;
  a = a + 4;                  }
}
```

r_4 to r_{11}

lr → insufficient

main() ⟶ A() ⟶ B()

lr → 80, Hence 42 gets overwritten

B() will have count of how many variables are there.

B will save old registers.

Assembly Language Function : Procedure

Stack pointer ⟶ Incremented to create space if we don't know whether space is available or not

To store 3 values :

```
MOV r3, sp
SUB sp, sp, #12      → Erasing Old data
MOV sp, r3                                  sp → 52
STR r0, [sp, #8]     (0, ∵ Top)             sp → 48
STR r1, [sp, #4]                            sp → 44
STR r2, [sp, #0]                            sp → 40
```

sp ⟶ location of the top of the stack

Use LDR to restore the values.

24/1

push ⟶ decrease add ⟶ sp

pop ⟶ increase adds → sp

```
MOV r1, #40          MOV r1, #40
MOV r2, #46          MOV r2, #46
MOV r3, #80          MOV r3, #80
SUB sp, sp, #12      LDR r2, [sp, #8]
STR r2, [sp, #8]     LDR r3, [sp, #4]
STR r3, [sp, #4]     LDR r4, [sp, #0
STR r4, [sp, #0      ADD sp, sp, #12
```

P1 :

```
2000 MOV r5, #6
2004 ADD r4, r5, #4
2008 BL P2
```

PC = 2012

LR = 2012

P2 :

```
3000 MOV r6, #6
3004 ADD r7, r6, r9
3008 BL P3
3012 LDR lr, [sp, #0]
3016 MOV pc, lr
```

PC = 3012

LR = 3012

P3 :

```
4000 MOV r7, #9
4004 ADD r7, r8, #8
     MOV pc, lr
```

Standard way to end a function & return back to previous call

Update lr ⟶ new lr

27/1

A

```
4000 ADD r4, r4, r5
4004 BL B
```

LR = PC

PC = 5000

(LR = 4008)

B

```
5000 SUB r6, r6, r7
5004 BL C
```

PC = 6000

LR = 5008

4008 is being overwritten

C

```
6000 ADD r8, r9, r9
6004 SUB r8, r9, r9
6008 MOV pc, lr
```

∴ B

```
5000 SUB r6, r6, r7
5004 STR lr, [sp, #0]
5008 BL C
5012 LDR lr, [sp, #0]
5016 MOV pc, lr
```

r_0, r_1, r_2, r_3 ⟶ Arguments

↳ Return value which can be stored

A

```
ADD r4, r4, r5
BL B
ADD r4, r0, r4
```

B

```
SUB sp, sp, #8
SUB r6, r6, r7
MOV r0, r6
STR lr, [sp, #0]
STR r0, [sp, #4]
BL C
LDR lr, [sp, #0]
ADD r7, r7, r8
MOV pc, lr
```

C

```
ADD r0, r9, r9
SUB r8, r9, r9
MOV r0, r8
MOV pc, lr
```

(Store lr in stack)

D

Store r_4 & r_5 stack

```
MOV r4, #9
MOV r5, #9
ADD r4, r5, r4
LDR r4, [sp, #0]
```

A	B	C	D
ADD r_4, r_4, r_5	SUB sp, sp, #8	SUB sp, sp, #8	Store r_4 & r_5 stack
BL B	SUB r_6, r_6, r_7	STR lr, [sp, #0]	MOV r_4, #4
ADD r_4, r_0, r_4	MOV r_0, r_6	STR r_0, [sp, #4]	MOV r_5, #9
	STR lr, [sp, #0]	ADD r_0, r_9, r_9	ADD r_4, r_5, r_4
	STR r_0, [sp, #4]	SUB r_8, r_9, r_9	LDR r_4, [sp, #0]
	BL C	MOV r_0, r_8	
	LDR lr, [sp, #0]	MOV pc, lr	
	ADD r_7, r_7, r_8		
	MOV pc, lr		

Every Procedure has its own frame to store local variables.

Frame ⟶ Part of Stack, specific for a procedure call / function

Flags ⟶ z, c, v, 0, ...

↳ 32 bits

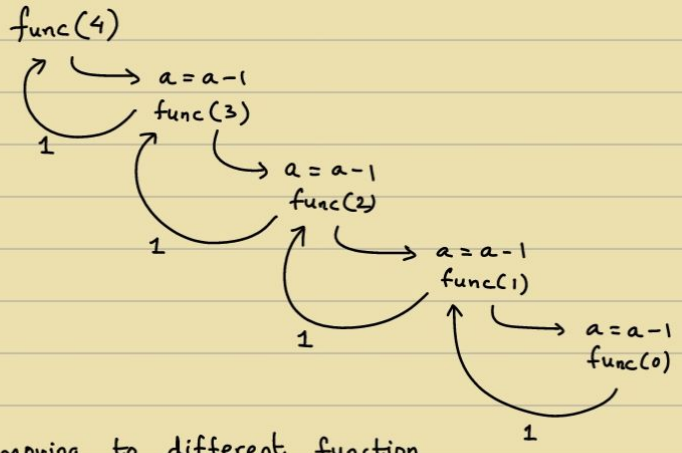
CMP command updates these flags.

H.W

```
int func(int a) {
  if (a<1)
    return a
  else {
    a = a-1;
    return func(a)
  }
}
```

[a ⟶ Random +ve number]

(a = 4)



28/1

Sol:

```
MOV r1, #4
FUNC: SUB sp, sp, #8      [2008 → 2000]
      STR lr, [sp, #4]
      STR r0, [sp, #0]
      CMP r1, #1
      BGE ELSE
      MOV r0, #1
      MOV pc, lr
ELSE: SUB r1, r1, #1
      BL FUNC
      LDR lr, [sp, #4]
      ADD sp, sp, #8
      MOV pc, lr
```

When moving to different function

BL: Unconditional function call

27/1 (Lab)

```
sudo apt install gcc-arm-linux-gnueabi qemu-user gdb-multiarch -y
```

QEMU – Virtual Machine emulator
↳ ARM processor based system (Server)

- **Compilation:**

```
arm-linux-gnueabi-gcc -g -o main sample.c
qemu-arm -g 8080 -L /usr/arm-linux-gnueabi ./main
```

New terminal:

```
gdb-multiarch ./main
    set architecture main          / set arch main
    target remote localhost:8080
    break main                     / b main
    continue                       / c
    ni : next instruction
    b address *address : Go to the location
```

29/1

Q)
```
int func(int a) {
  if (a<1)
    return a
  else {
    a=a-1;
    return (func(a)+a);
  }
}
```

Sol:

```
MOV r1,#4
FUNC: SUB  sp,sp,#8
      STR  lr,[sp,#4]
      CMP  r1,#1
      BGE  ELSE
      MOV  r0,#1
      MOV  pc,lr
```

```
ELSE: SUB r1,r1,#1
      STR r1,[sp,#0]
      BL FUNC   ; Backtracking
      LDR r1,[sp,#0]
      LDR lr,[sp,#4]
      ADD r0,r0,r1
      ADD sp,sp,#8  ; Deallocate stack space
      MOV pc,lr
```

31/1

Q)
```
int func(int k) {
    if (k<1)
      return 1;
    else {
      k = k + func(k-1);
      return k;
    }
}
```

- Two's Complement :

① Sign bit representation :

5 → 0101

−5 → 1101

Issue : +0 : 0000 } Must be same

−0 : 1000

② One's Complement :

5 → 0101 +0 : 0000

−5 → 1010 −0 : 1111

③ Two's Complement

5 → 0101 +0 : 0000

−5 → 1010 −0 : 1111

1 1

1011 10000

Ignored → sign bit

- Multiplication :

$$\begin{array}{r} {\scriptstyle 2}\\ 14\\ \times\ 25\\ \hline 70\\ 280\\ \hline 350 \end{array}$$ } Shift + Addition (3)

$$\begin{array}{r} 14\\ 25\\ \hline 39 \end{array}$$

$n^2 + n + \ldots$ → Depends on the no. of digits of Multiplicand & Multiplier.

$\times \underline{102304}_{000004}$ is more complex than $\times \underline{363}_{102}$

000004 → 6×6 ; more complex → Acc. to computer ; 102 → ∵ 3×3

Binary Multiplication :

```
  1 0 1   ⟶ 5
x 1 1 0   ⟶ 6
---------
  0 0 0
 1 0 1 0
1 0 1 0 0
---------
1 1 1 1 0  ⟶ 30
```

Not recommended

Recommended : Booth's Algorithm

Shifting can be used instead of Multiplying
↳ Left shift

SC : Size of Multiplier

B : Base

A : 0000 (Initially)
↳ Accumulator

B ⟶ use

sum = 0

sum = sum + a[i]

3/2

A = 0

QR = Multiplier (Analyzed)

BR = Multiplicand (Operation)

n bits ⟶ $Q_{n+1} = 0$ [n → no. of bits of QR]

Sc ⟶ n

Ex. 7 x 5

↳ 0101 ⟶ +5

0111 ⟶ +7

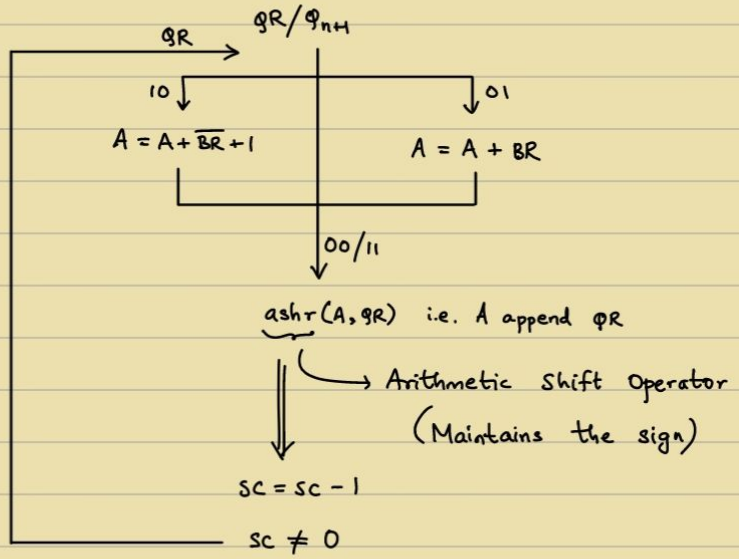
Works for signed numbers (Booth's Algo)

QR ⟶ no. having more continuous 1s or 2s
(Analyzed easier)

Append $Q_{n+1} = 0$ @ the end

01110

Compare 2 bits @ a time



Ex. 0111

→ 01110

1011 0111
A qR

1 → 11011011 ⇒ 10 (Appended value) [01110 ⇒ A - BR]
A qR

2 → 1110 1101 ⇒ 11 [01110 ⇒ Shift Right]

3 → 1111 0110 ⇒ 11 [01110 ⇒ Shift Right]

4 → 0010 0011 ⇒ 01 [01110 ⇒ A + BR]

A : 1111 (updates)

BR : 0101 (doesn't update)

32 +2+1 → 35

0010 0011

0100 0110

Ex. 3 × 5

BR

101 → 0101 → +5

011 → 0011 → +3

qR

q_{n+1}

00110

$\overline{BR}$ → 1010

$\overline{BR}$ + 1 → 1011

10 : A 1011 0011 qR

[sc = 4] 1101 1001 ← Final

00110

11 : 1110 1100

[sc=3]

00110

01 : 0011 1100

[sc=2] 0001 1110

1110
0101
10011

00110

00 : 0000 1111

[sc=1]

∴ 15

[Shifting always]

4/1

Ex. 6×7

0110 0111

Sol : A = 0000

QR = 0111

BR = 0110

Q_{NH} = 0

Sc = no. of bits of QR = 4

00/11 ⟶ ashr(A, QR)

01 ⟶ A = A + BR & Right Shift & Sc--

10 ⟶ A = A + $\overline{BR}$ + 1 & Right Shift & Sc--

Sc

④ QR = 01110

1010 0111

↓ Right shift

Sc

③ 1101 0011

QR = 0111

A = 0000

BR = 0110

⟹ A = 1010

QR = 01110 ⟹ 1110 1001 Sc = ②

QR = 01110 ⟹ 1111 0100 Sc = ①

QR = 01110

↓

0101 0100

↓ Right shift

0010 1010 Sc = ⓪

A = 1111

BR = 0110

1)0101

⟶ 10 + 32 = 42

∴ 6×7 = 42

Ex. -4×5

$-4 : 1100 \longrightarrow QR$

$A \longrightarrow 0000$

$BR \longrightarrow 0101$

$$\begin{array}{r} 0100 \\ 1011 \\ \quad\;\; 1 \\ \hline 1100 \end{array}$$

$QR = 11000$

	A	QR	
	0000	1100	
③	0000	0110	11000
②	0000	0011	11000
	1011	0011	
①	1101	1001	11000
⓪	1110	1100	11000
	↳ -ve		
(-ve)	0001	0100	$\longrightarrow -20$

Ex -3×7

$A = 0000$

$QR = 7 = 0111$

$\Rightarrow QR = 01110$

$Sc = 4$

$3 : 0011$

$$\begin{array}{r} 1100 \\ \quad\;\; 1 \\ \hline 1101 \end{array} \longleftarrow BR$$

Sc	A	QR		
	0000	0111		
③	0011	0111	01110	$1101 \longrightarrow 0011$
	1001	1011		
②	1100	1101	01110	
①	1110	0110	01110	
⓪	1011	0111	01110	
	1101	1011		
	↳ -ve			

$$\begin{array}{r} 1110 \\ 1101 \\ \hline 1)1011 \end{array}$$

$\Rightarrow -21$

$$\begin{array}{r} 11011011 \\ 00100100 \\ \quad\;\; 1 \\ \hline 00100101 \end{array}$$

$\Rightarrow 21$

Q) -2×-3

QR = 1110

Sc = 4

QR = 11100

BR = 1101

A = 0000

(-2)	(-3)
0010	0011
1101	1100
1	1
1110	1101

Sol:

Sc	0000	1110	⇒ 11100
③	0000	0111	⇐ ⇒ 11100
②	0011	0111	⇐ ⇒ 11100
	0001	1011	⇒ 11100
①	0000	1101	⇐ ⇒ 11100
⓪	0000	0110	⇐ 11100

$A + (BR)^{2s}$ = 0000 + 0011 = 0011 ; 1101 + 0010 + 1 = 0011

If Starting from 1 ⇒ Right Shift is Circular

If Starting from 0 ⇒ Omit ending digit, start with 0.

$\therefore -3 \times -2 = (0110)_2 = 6$

Ex. -13×-5

5/1

- Another method:

Q) 6 : 0110 (Multiplicand)

5 : 0101 (Multiplier)

Sol:

0000	0000
0000	0110 ← Step 1
0000	0110 ← Step 2
0001	1110 ← Step 3
0001	1110 ← Step 4

30

0 : Do Nothing

Booth's Algorithm ⟶ better

Ex. 7 × 8

7 × 8

Multiplicand [Shift Left] 0111

Multiplier [Shift Right] 1000 (Initially)

00000000 1000

0 Initially:

Mr → 0100

Md → 01110

00000000 0100

0 :

Mr → 0010

Md → 11100

00000000 0010

0:

Mr → 0001

Md → 111000

00000000 0001

1 : A → A + Md

Then, shift

00111000 0000 → 56

Md → 0000 } Unnecessary

Mr → 1110000

Steps:

① Add if 1

② Shift →

→ Right → BR – Mr

→ Left → BR – Md

- Adding Floating Point numbers:

Bigger value should be converted into smaller value

49.323 → 4.9323 × 10¹ (Bigger) ⟹ 49.323 × 10⁰

4.86 → 4.86 × 10⁰ (Smaller) 4.86 × 10⁰

} Now Add

Ex. $42.50 \Longrightarrow 101010.1 \Longrightarrow 1.010101 \times 2^5$

$31.625 \Longrightarrow 11111.101 \Longrightarrow 1.1111101 \times 2^4$

1.010101×2^5	1.1111101×2^4
$E = b + e$	$E = b + e$
$b = 127$	$b = 127$
$e = 5$	$e = 4$
$\Rightarrow E = 132$	$\Rightarrow E = 131$
$\Rightarrow$ 0 10000100 0101010	$\Rightarrow$ 0 10000011 1111101

Can't add $\because 2^5$ & $2^4 \longrightarrow$ Different

10.10101×2^4

1.1111101×2^4

$$\begin{array}{r} 10.1010101 \\ 1.1111101 \\ \hline 100.1010001 \times 2^4 \end{array}$$

$\Rightarrow 1.001010001 \times 2^6$

42.50
31.625
74.125

7/2

$42.50 \Longrightarrow 1.0101010 \times 2^5 \Rightarrow$ 0 10000100 0101010

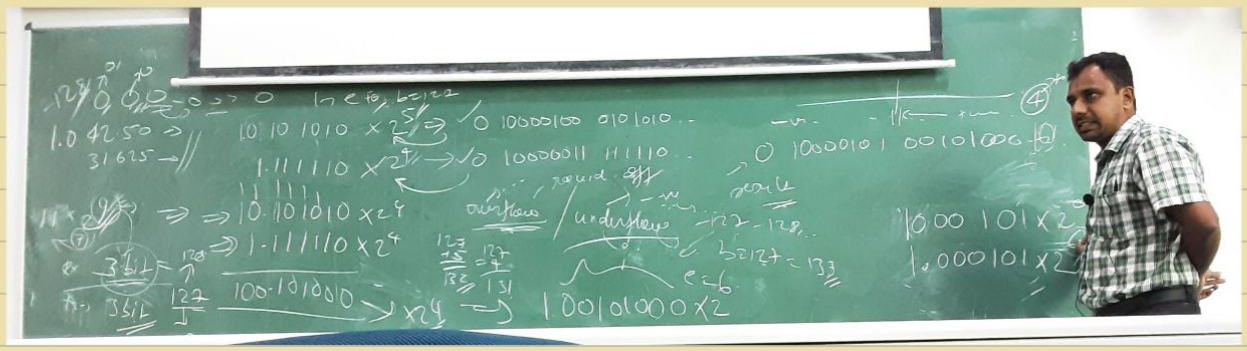
$31.625 \Longrightarrow 1.11111 \times 2^4 \Rightarrow$ 0 10000011 11110

$$\begin{array}{r} 10.101010 \\ 1.111110 \\ \hline 100.101000 \times 2^4 \end{array} \Rightarrow 1.00101000 \times 2^6$$

$e = 6$

Overflow : $0.1001 \Rightarrow 0.101 \Rightarrow 0.11$ (Round off)

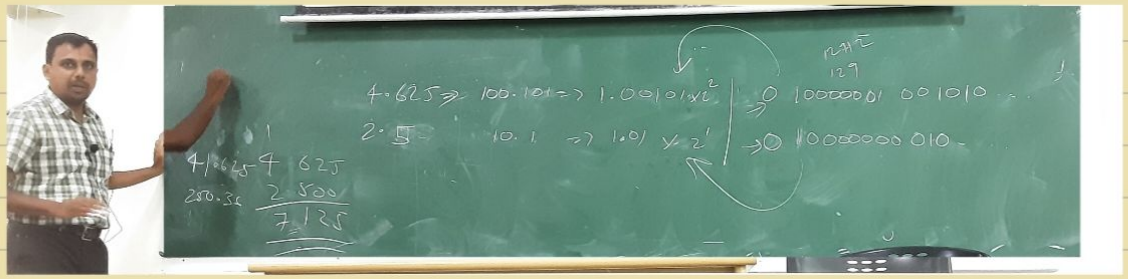
Underflow : Add min no.

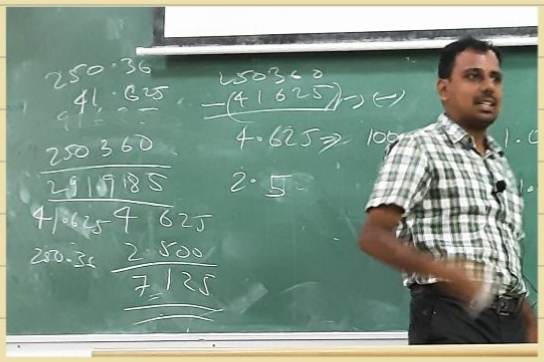


1000101×2^0

1.000101×2^6

$1.00011 \Longrightarrow 1.0010$





10.0101×2^1

010100×2^1

$4 : 100.0$

$2.5 : 10.1$

10/2

$10.10 \times 2^4 \longrightarrow$ Fixed Point number

$1.010 \times 2^5 \longrightarrow$ Floating Point number

$-2.5 \Rightarrow 2.5 : 10.1$

$4 \Rightarrow 4 : 100.0$

$\therefore 2.5 \rightarrow 010.1$

101.0

$\quad\quad 1$

$-2.5 \rightarrow 101.1$

Now, $4 + (-2.5)$

$4 : 100.0$

$-2.5 : 101.1$

1001.1

$(1.1)_2 \equiv (1.5)_{10}$

① → 32 bit: 0 | E | M

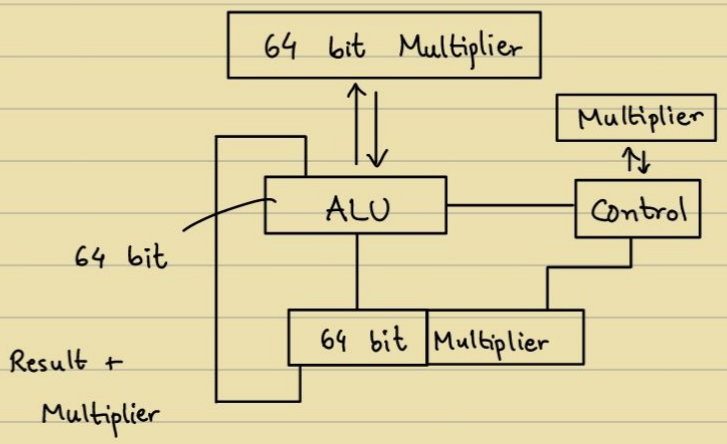
0 | E | M

FPN ① + FPN ②

Multiplicand ⟶ 32 x 64 bits

Multiplier ⟶ 32

Result : 65 bits X 64 bits



Optimising storage space

13/2

Revision :

- int fun (int a) {
  a++;
  }

$r_0 - r_3$ ⟶ Reused , For storing values.

$r_4 - r_{11}$ ⟶ Not Reused

lr ↗

```
fun : ADD r1, r1, #4
```

- int fun (int a) {
  if (a<6) {
  a++;
  }
  }

```
fun : ADD r1, r1, #4
      CMP r1, #6
```

```
int fun (int a) {
  if (a<6) {
  a++;
  }
  else {
  exit (1);
}
```

```
fun : ADD r1, r1, #4
      CMP r1, #6
      BLT L1
L1 : ADD r1, r1, #1
```

↳ This is wrong (∵ Sequential)

```
fun : ADD r1, r1, #4
      CMP r1, #6
      BGE L1
      ADD r1, r1, #1
L1:
```

-
```
int fun(int a){
  while (a<6){
    a++;
  }
  else{
    exit(1);
  }
}
```

```
int main(){
  fun(a);
  return 0;
}
```

```
4000   fun: ADD r1, r1, #4
4004 WHILE: CMP r1, #6
4008        BGE L1
4012        ADD r1, r1, #1
4016        B WHILE
4020    L1:
        MOV pc, lr
```

```
5000 MAIN      PC: 5004
5004 BL fun    PC: 5008 ⇒ lr = 5008
5008           PC: 4000
```

-
```
int fun(int a){
  while (a<6){
    a++;
  }
  return a;
}
```

```
int main(){
  int a=4;
  fun(a);
  return 0;
}
```

```
4000   fun: ADD r1, r1, #4
4004 WHILE: CMP r1, #6
4008        BGE L1
4012        ADD r1, r1, #1
4016        B WHILE
4020    L1:
        MOV pc, lr
```

```
5000 MAIN
5004 MOV r1, #4
5008 BL fun
```

```
int fun(int a){
    while (a<6) {
        a++;
    }
    fun2(int b);
    return a;
}
```

```
int main() {
    int a = 4;
    fun(a);
    return 0;
}
```

```
fun2(int b) {

}
```

```
4000    fun : ADD r1, r1, #4
4004  WHILE : CMP r1, #6
4008          BGE L1
4012          ADD r1, r1, #1
4016          B WHILE
4020      L1: MOV r0, r1
              MOV pc, lr
              STR lr, [sp, #0]
              LDR lr, [sp, #0]
```

```
5000  MAIN
5004  MOV r1, #4
5008  BL fun
```

MIDSEM : No pipelining

18/2 **Division :**

4 bit - Divisor, Dividend

4 bit - Remainder, Quotient

5/3

5 : 0101

3 : 0011

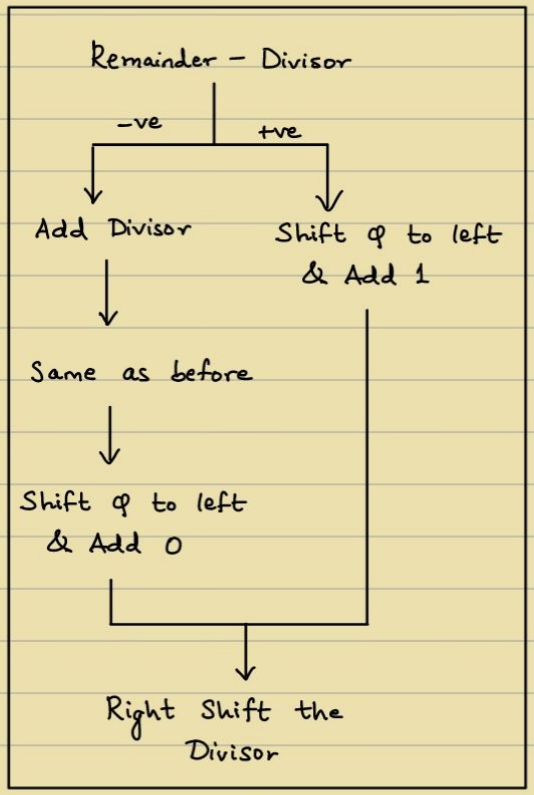
$4 + 1 \longrightarrow 5^{th}$

$\downarrow n$

8 bit — Divisor

8 bit — Remainder

4 bit — Quotient



	q	Divisor	Remainder
	0000	0011 0000	0000 0101
①	0000	0001 1000	0000 0101
			①11 .....
②	0000	0000 1100	0000 0101
			①11 .....
③	0000	0000 0110	0000 0101
			11 .....
④	0000	0000 0011	0000 0101
⑤	0001	0000 0001	0000 0010

Quotient : 1

Remainder : 2

Triple Adder → In Syllabus

Fixed Point Addition

Fixed Point Subtraction

Ex. 6/2

6 : 0110

2 : 0010

$4 + 1 \longrightarrow 5^{th}$

$\downarrow n$

	q	Divisor	Remainder
	0000	0010 0000	0000 0110
①	0000	0001 0000	0000 0110
②	0000	0000 1000	0000 0110
③	0000	0000 0100	0000 0110
④	0001	0000 0010	0000 0010
⑤	0011	0000 0001	0000 0000

Quotient : 3

Remainder : 0

Ex. 11/4

11 : 1011

4 : 0100

5 + 1 ⟶ 6th Step

↳ n

	Q	Divisor	Remainder	
	00000	00100 00000	00000 01011	⟶ ①1.....
Step ①	00000	00010 00000	00000 01011	⟶ ①.....
②	00000	00001 00000	00000 01011	
③	00000	00000 10000	00000 01011	
④	00000	00000 01000	00000 01011	
⑤	00001	00000 00100	00000 00011	(11-8) ⟶ ①11.....
⑥	00010	00000 00010	00000 00011	

00010 → Quotient : 2

00000 00011 → Remainder : 3

Q) −4/2

4 : 0100 ⟶ 1011

2 : 0010 → 1 / 1100 → −4

4 + 1 ⟶ 5th Step

↳ n

	Q	Divisor	Remainder	
	0000	0010 0000	0000 1100	⟶ ①1.....
①	0000	0001 0000	0000 1100	
②	0000	0000 1000	0000 1100	
③	0001	0000 0100	0000 0100	
④	0011	0000 0010	0000 0000	
⑤	0110	0000 0001	0000 0000	

0110 → 3 bit

0000 0000 → Remainder : 0

0110 → 110 ⟶ (2's Comp.) ⟶ 010

110
001
1
010

010 → Quotient : −2

H.W

Q) −4/−2 ⟶ Check

No 2's complement @ the end

19/2

① $7/3 \Rightarrow$ Q: +ve, R: +ve
② $-7/3 \Rightarrow$ Q: -ve, R: -ve
③ $7/-3 \Rightarrow$ Q: -ve, R: +ve
④ $-7/-3 \Rightarrow$ Q: +ve, R: -ve

$Q = Div^d \oplus Div^r$
$R = Div^d$

Q) $-13/5$

Dividend : 01101
Divisor : 00101
Quotient : 00000

$n+1 = 5+1 = 6$

	Q			
	00000	00101 00000	00000 01101	①1 .....
①	00000	00010 10000	00000 01101	①1 .....
②	00000	00001 01000	00000 01101	
③	00000	00000 10100	00000 01101	
④	00000	00000 01010	00000 01101	
⑤	00001	00000 00101	00000 00011	①1 .....
⑥	00010	00000 00010	00000 00011	

Q : 2 ; R : 3 $\Rightarrow$ R = -3

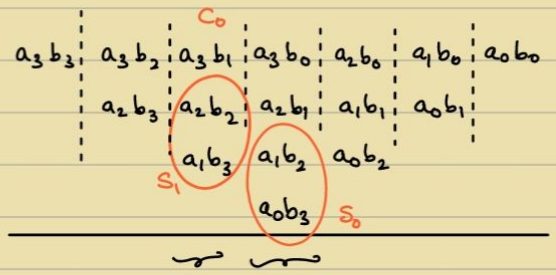
Can be done with 4 bits – Manually
Conventionally & Systematically $\longrightarrow$ 5 bits

21/2

$\longrightarrow$ Wallace Tree Multiplication :

			a_3	a_2	a_1	a_0	
			b_3	b_2	b_1	b_0	
			a_3b_0	a_2b_0	a_1b_0	a_0b_0	
		a_3b_1	a_2b_1	a_1b_1	a_0b_1	X	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2	X	X	
a_3b_3	a_2b_3	a_1b_3	a_0b_3	X	X	X	

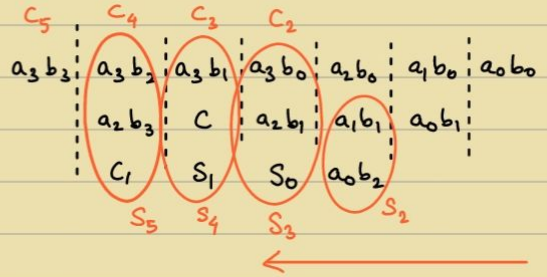
			1	0	1	0	
			1	1	1	0	
			0	0	0	0	
		1	0	1	0	X	
	1	0	1	0	X	X	
1	0	1	0	X	X	X	
1	0	0	0	1	1	0	0

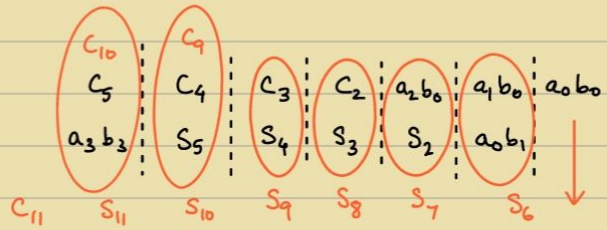


More partial products
(4 levels)

Along with carry → 4 levels

∴ Reduce





→ Addressing Modes (ISA)

Immediate Addressing Mode : MOV r_1, #42

Register Addressing Mode : MOV r_4, r_1

Direct Addressing Mode : MOV r_1, [#42]

Indirect Addressing Mode : MOV r_1, #42
MOV r_2, [r_1]

Similar to :
Index Addressing Mode
Base Addressing Mode
} Future topics

Floating Point Multiplication — Not in Midsem Syllabus : Until pipelining

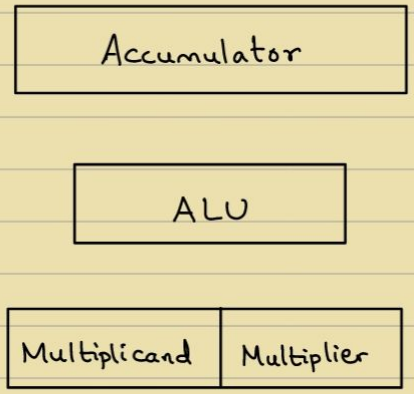
→ Traditional Hardware Multiplication :

	(64 bit) Acc	(64 bit) Multiplicand	(32 bit) Multiplier	
	0000 0000	0000 0110	0100	(32 bit) Multiplicand : ←
①	0000 0000	0000 1100	0010	(32 bit) Multiplier : →
②	0000 0000	0001 1000	0001	
③	0001 1000	0011 0000	0000	Multiplicand : 4 bit } 8 bit
④	0001 1000	0110 0000	0000	Multiplier : 4 bit }
	→ 24			Accumulator : 8 bit

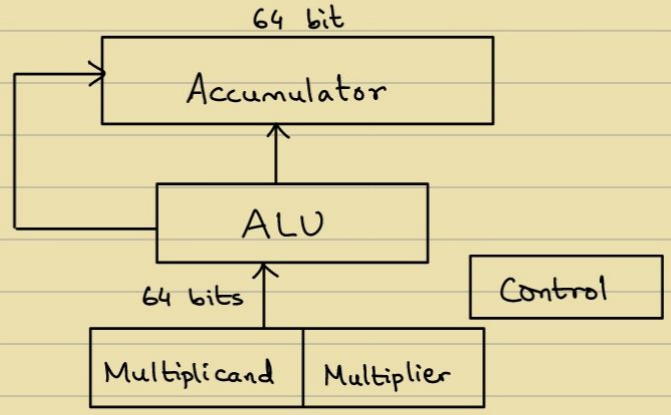
→ Steps : No. of bits of Multiplier
& = No. of bits of Multiplicand Initially

1 : Acc = Acc + Mul^d

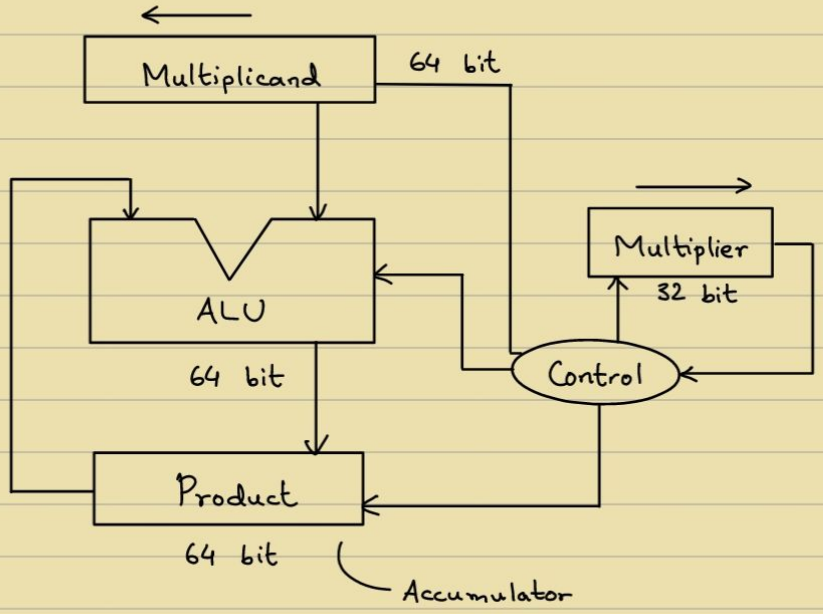
Naturally :



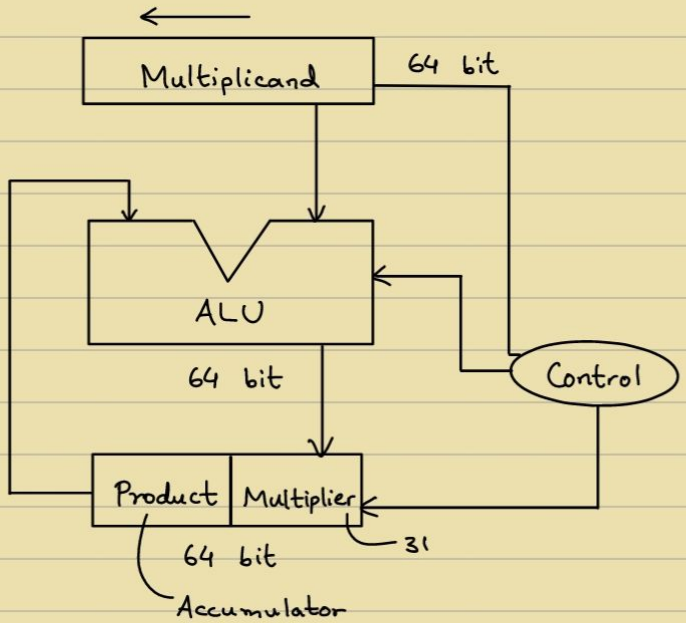
Optimization :



21/2



Control Unit : Read the LSB



0111 - Mul^d

0101 - Mul^r

0000 0101

0000 0010

0111 0010

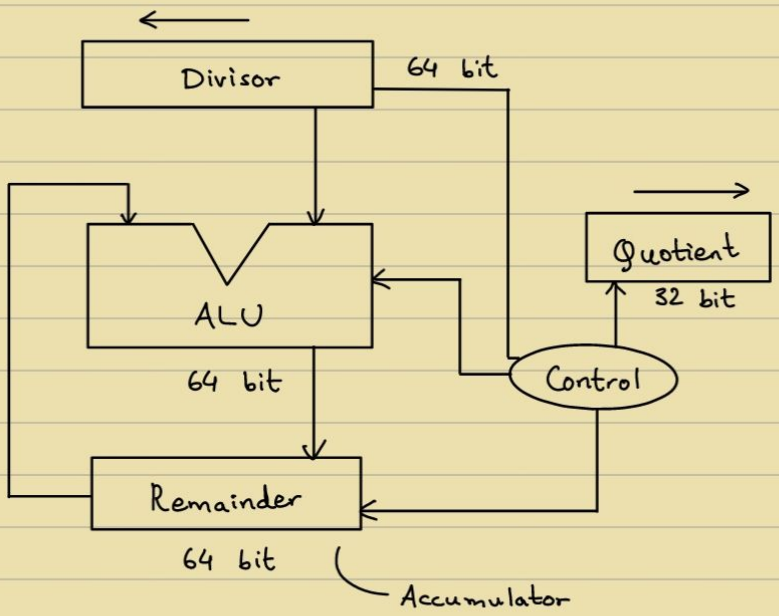
0011 1001

0001 1100

0111 0

1000 1100

0100 0110



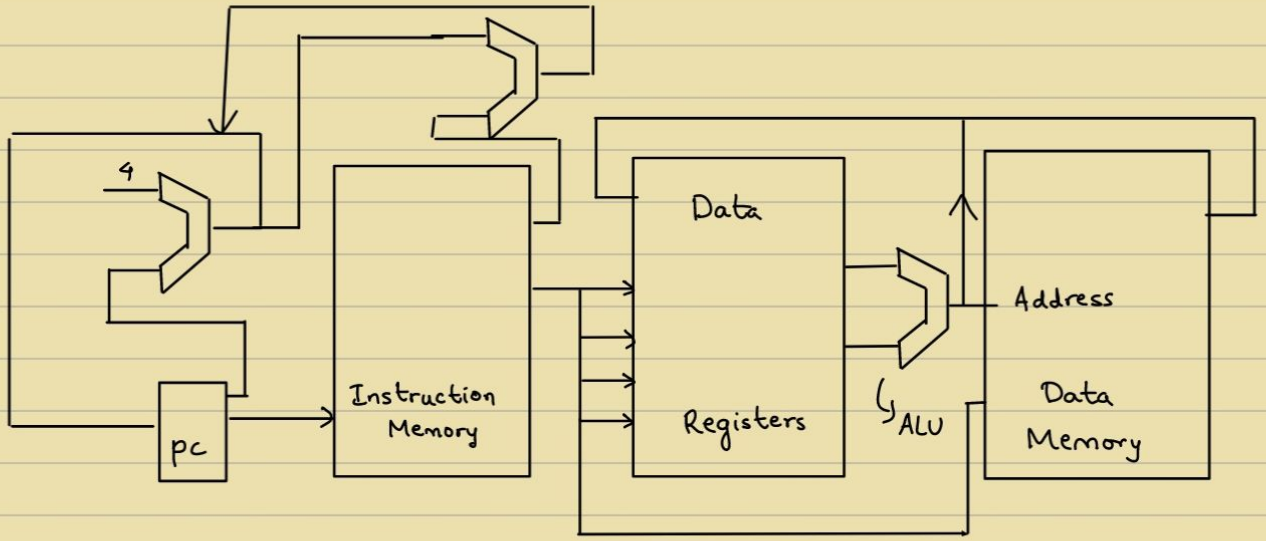
No need to read Quotient

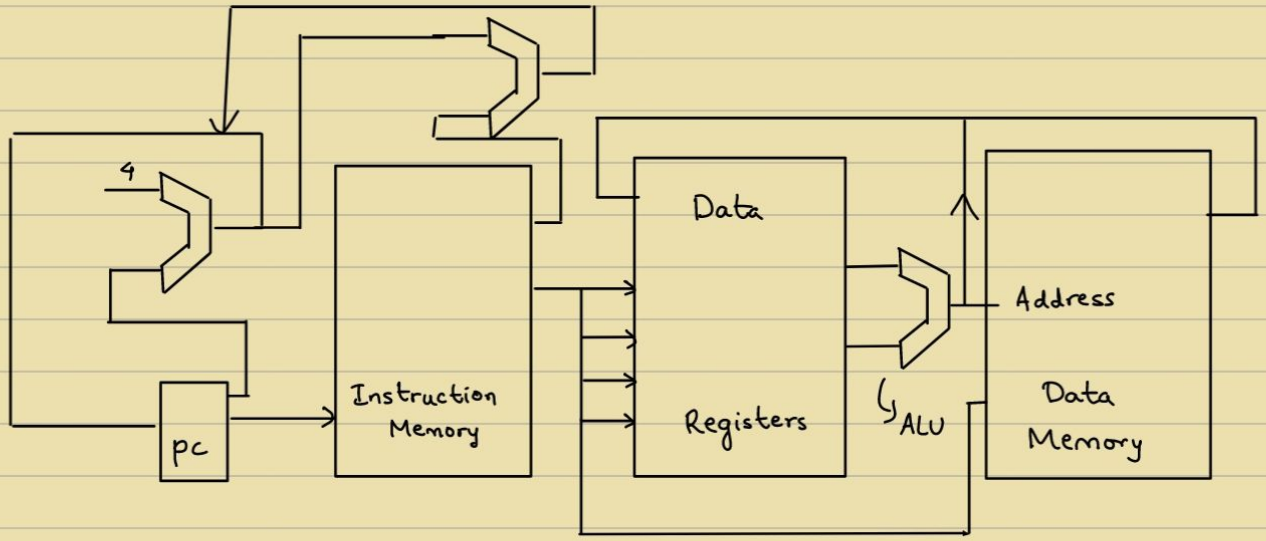
4/3

① Memory Instructions
② Arithmetic and Logical Operations
③ Branching

MIPS : Million instructions per second

↳ Microprocessors without Internal pipelining stages

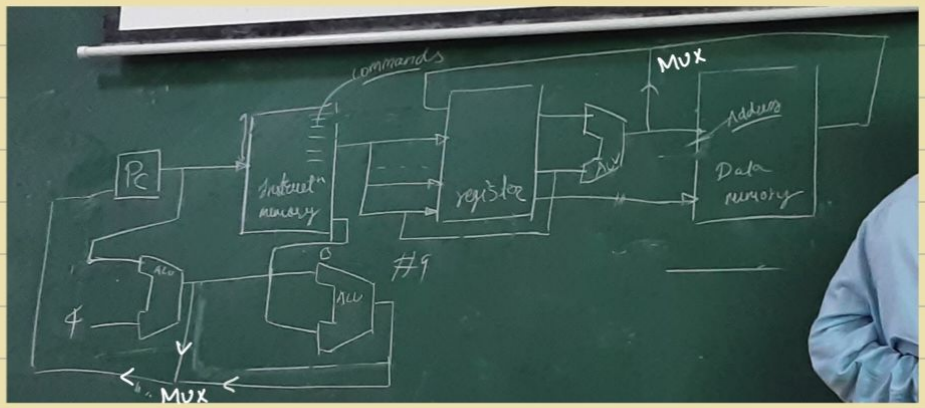




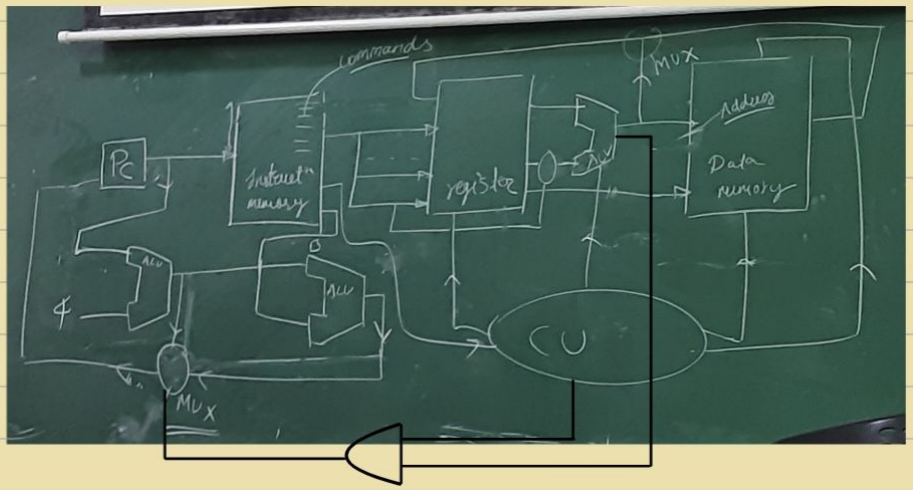
5/3

iVerilog → Software

GTK → (Analysing Circuits)



STR : Data is transferred from register to location



ALU → doesn't store anything

↳ Not sequential, i.e. Comb"

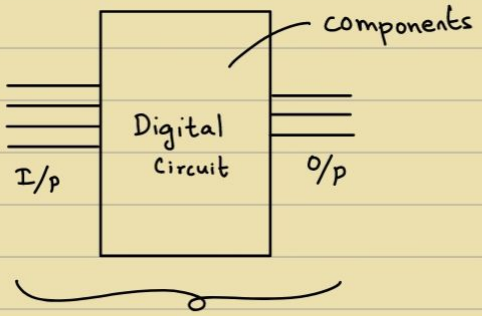
State machine

More states ⇒ More speed of clock cycle.

If no. of clock cycles/sec ↑, ⟹ Capacity of the Hardware
& complexity of operations ↑, ⟹ Hinder the system

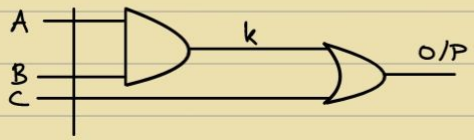
① Manufacturer
② Architect → RISC (Smaller → easier to execute)
- Minimizes the CPI, decides how fast the system is
③ Compiler

6/3 ASIC : Application specific
Logisim : Realising small circuits
↳ Not Scalable [VLSI ↳ Very Large Scaling Integration]



↳ Simple language → Logisim
Better for design → Verilog

- Verilog :
① Supports GATE level Design
Each operation is a function



AND (k, A, B)
OR (Y, k, C)

Problem with this 1st level :
Interconnections → Complex
More no. of Gates

② Supports Data Flow Model
- Uses Gates directly
- Data flow is decided by commands

k = A & B
Y = k | C

③ Behaviour Model

- System translates by itself
- Advantage : No need to worry about no. of gates, design of circuit
- Ex. MUX

MUX ≡ Switch - Case (C language)

4 Inputs ⟹ 2^4 testcases

Best way to Realising output : LED Light (for small output)

For complex operations : ?

To analyse 'n' set of outputs ⟶ Timing Diagram

GTK ⟶ Timing Diagram : Data ⇒ Waveform (Analyse Output)
↳ of Testbench

UUT ⟶ TestBench

iVerilog ⟶ Compiler (Takes Verilog file and gives output)

Verilog ⟶ Just a behaviour Model allowing us to give set of inputs

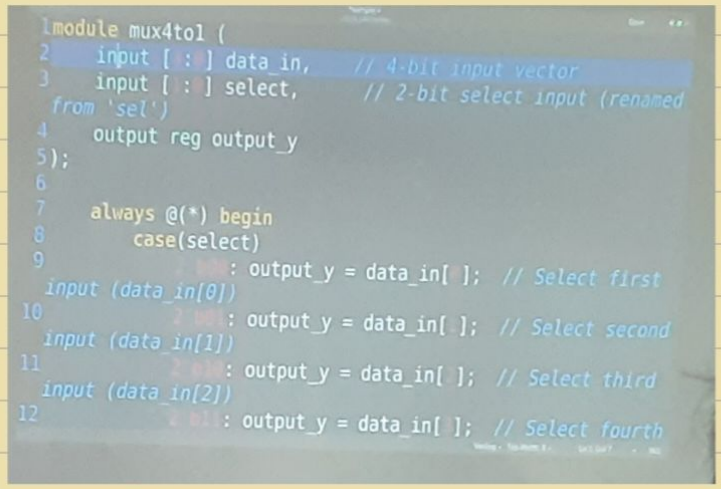
iVerilog is for basic circuits.

For advanced circuits,

Vivado (IDE) is used to design the circuits.
(License Based)

gedit sample.v ⟵ For iVerilog

```
module mux4to1 (
    input [3:0] data_in,
    input [1:0] select,
    output reg output_y
);
```



If reg is not mentioned, then it is a wire.

Wire : Does not store any value

input wire [3:0] data_in

↳ Default, no need to explicitly mention

3:0 ≡ 4:1 ⟹ Syntax

↳ Array of pins

Output : Always a reg

always @(data_in, select) ≡ always @(*)

↳ Whatever input is given, the change of either of the two is monitored.

Asterisk ⟶ When change is detected in any of the inputs.

begin ⟶ end

↳ Start keyword

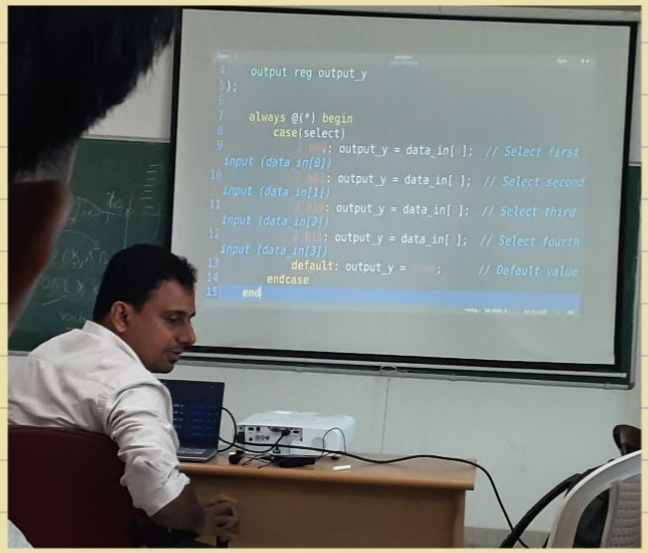
case ⟶ endcase

Select ⟶ 2 bits [: 1:0]

2'b00 ≡ 4'h6 ↳ 0110

↳ Indicates bits

↳ No. of bits



Hexadecimal ⟺ Binary

output_y ⟶ 1 bit

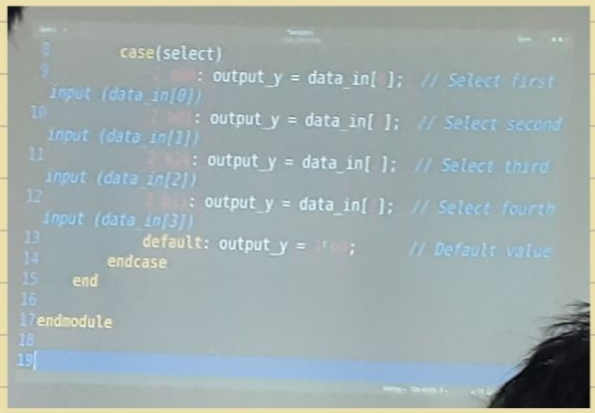
3:0 ⟶ 0110

3 2 1 0

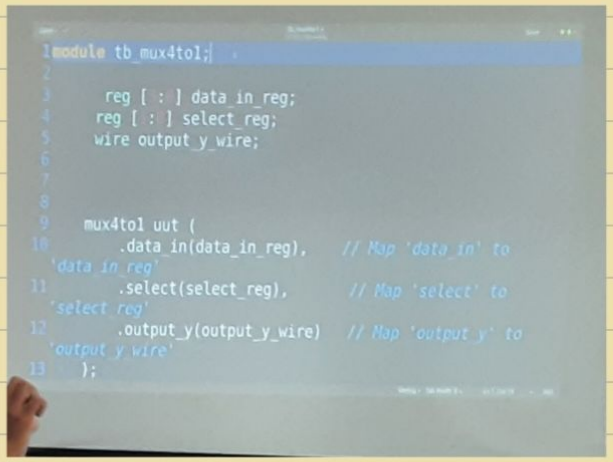
module ⟶ endmodule

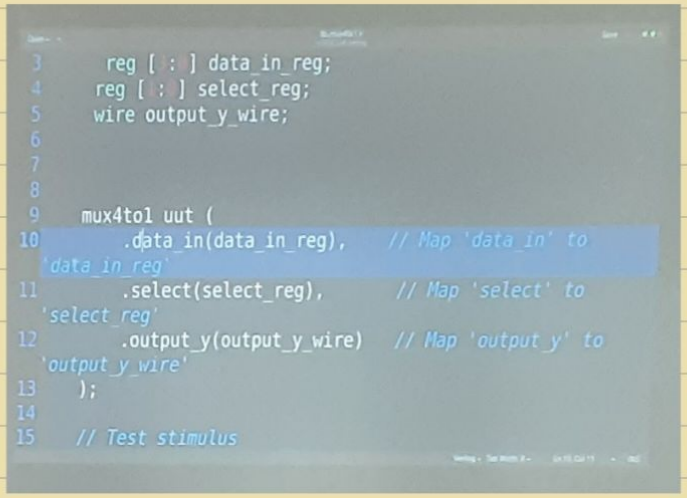
No Indentation rules

Commenting : // comment



- Testcase :



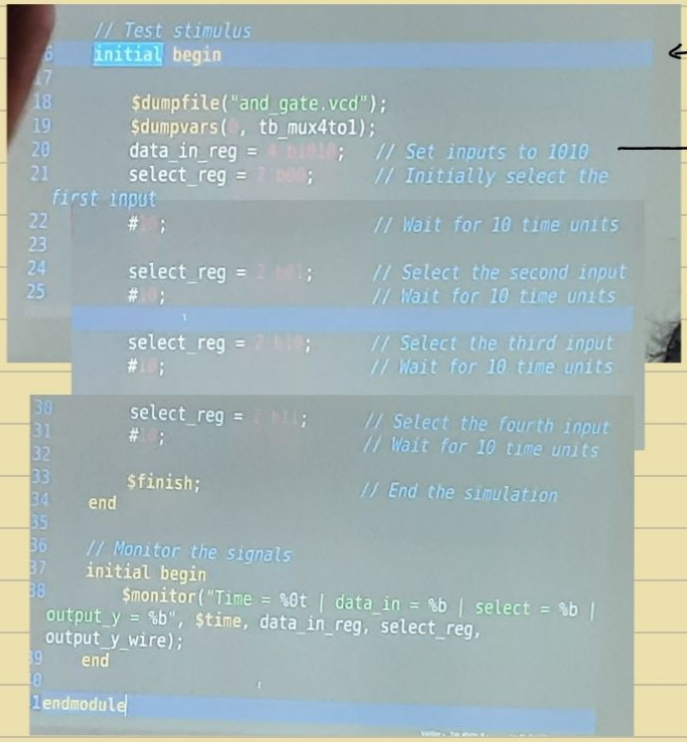


```
    reg [ : ] data_in_reg;
    reg [ : ] select_reg;
    wire output_y_wire;


    mux4to1 uut (
        .data_in(data_in_reg),    // Map 'data_in' to
'data_in_reg'
        .select(select_reg),      // Map 'select' to
'select_reg'
        .output_y(output_y_wire)  // Map 'output_y' to
'output_y_wire'
    );

    // Test stimulus
```

Mapping to the sample.v



```
    // Test stimulus
    initial begin

        $dumpfile("and_gate.vcd");
        $dumpvars(0, tb_mux4to1);
        data_in_reg = 4'b1010;   // Set inputs to 1010
        select_reg = 2'b00;      // Initially select the
first input
        #10;                     // Wait for 10 time units

        select_reg = 2'b01;      // Select the second input
        #10;                     // Wait for 10 time units

        select_reg = 2'b10;      // Select the third input
        #10;                     // Wait for 10 time units

        select_reg = 2'b11;      // Select the fourth input
        #10;                     // Wait for 10 time units

        $finish;                 // End the simulation
    end

    // Monitor the signals
    initial begin
        $monitor("Time = %0t | data_in = %b | select = %b |
output_y = %b", $time, data_in_reg, select_reg,
output_y_wire);
    end

endmodule
```

← Initialise the value

→ 4 : 4
b : bits
1010 : input

3:0 ⇒ 3 2 1 0
1 0 1 0

uut : keyword

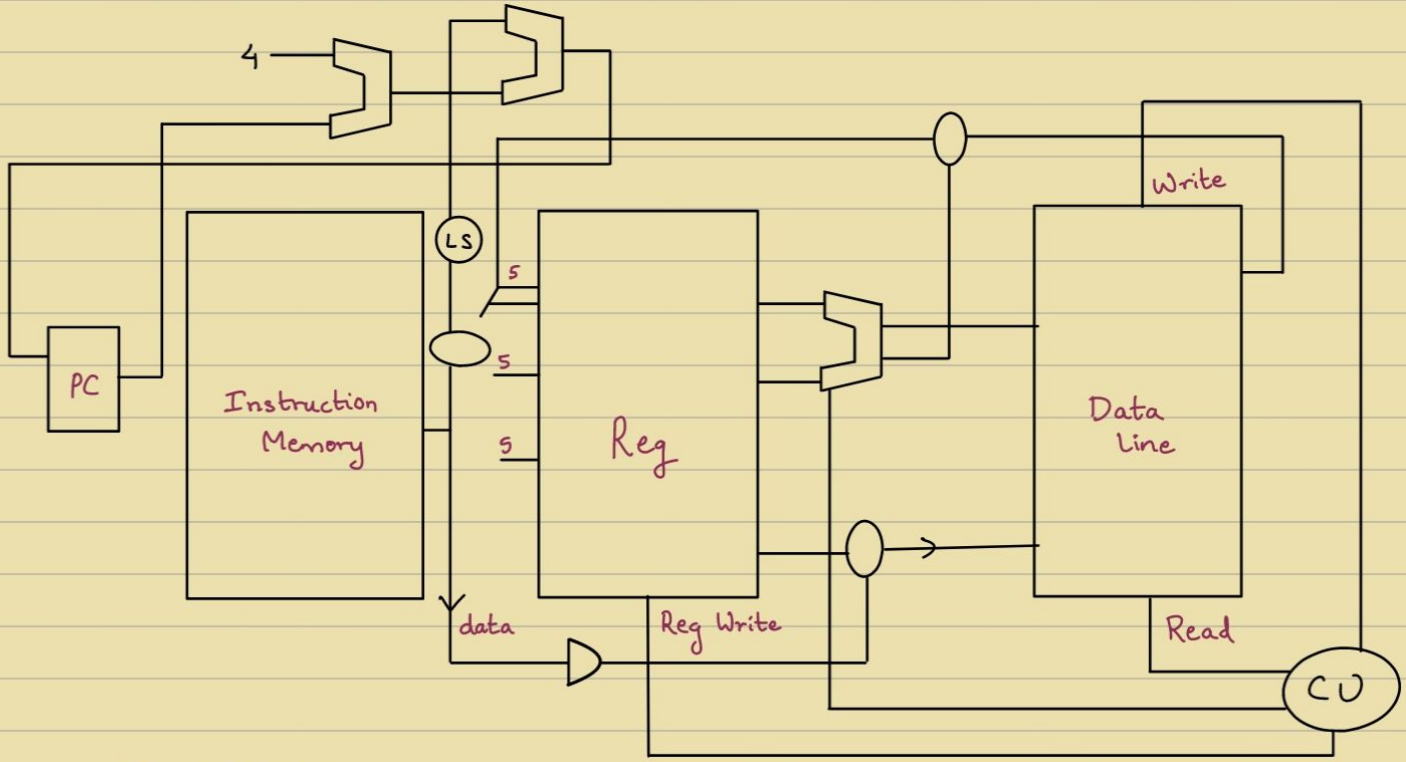
Outputs are stored in registers.
For realising output in waveform,
Data is put in the dumpfile
[0 → default value]

iverilog sample.v -o test → Compilation

iverilog sample.v tb_mux4to1.v -o testobj

tb_mux4to1.v → Testcases file

vvp testobj → Executes the file



LS : Left Shift

+ve :

4 bits : 0011

8 bits : 0000 0011

-ve :

4 bits : 1011

8 bits : 1111 1011

Offset ⟶ Variable

↳ 16 bits ⇒ 2^{16} bytes

$= 2^{10} \times 2^{6}$ bytes

= 64 kilobytes

Small value of Offset ⟹ More space in registers

But for operations to be possible : 16 bits ⇒ 32 bits

Offset is used in :

(i) Branching

(ii) Memory

(00000) 0 ⟵ 0 (0000)

(00100) 4 ⟵ 1 (0001)

(01000) 8 ⟵ 2 (0010)

(01100) 12 ⟵ 3 (0011)

(01000) 16 ⟵ 4 (0100)

(10100) 20 ⟵ 5 (0101)

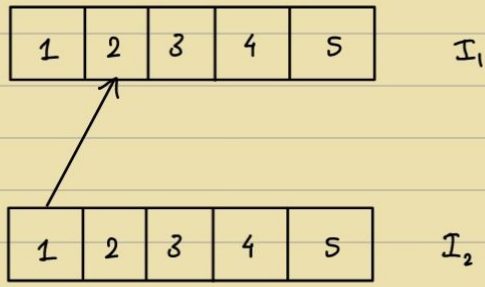
↳ Requires only 3 bits , but we get 2 bits extra (last two → 00)

Locality of Memory ⟶ Later Topic

13/3 :

Absent

14/3 :



① 1 of I_1

Extra clock cycle

	0	1	2	3	4	5
I_1	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	
I_2		I_{21}	I_{22}	I_{23}	I_{24}	I_{25}

Best way of arranging 5 Instructions

	0	1	2	3	4	5	
I_1	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}		
I_2	X	I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	
I_3	X	X	I_{31}	I_{32}	I_{33}	I_{34}	I_{35}
I_4	X	X	X	I_{41}	I_{42}	I_{43}	I_{44} I_{45}

Total no. of cycles taken for execution : $k+n-1$

17/3

n : no. of instructions

k : no. of stages

non-pipelined = $n \times k \times t$

pipelined = $(n+k-1) \times t$

$$\text{speed up} = \frac{n \times k \times t}{(n+k-1) \times t}$$

I_1	100	200	200	200	300	
I_2	X	100	200	200	200 (st)	300
I_3	X	X	100	200	200 (st)	200 (st) 300

$S_1 = 100$ ns
$S_2 = 200$ ns
$S_3 = 200$ ns
$S_4 = 200$ ns
$S_5 = 300$ ns

Max : 1600

Stalling

Overhead : Work more instead of actually getting the benefit
Thrashing : Operation itself is less loaded, but fetching it is more load.

CISC - Pipelining is difficult

Sol: Max size of clock cycle & fix for all stages.

$300 \longrightarrow (3+5-1) \times 300 = 2100$
But max = 1600

After k stages $\Rightarrow$ pattern repeats

H.W : Calculate for 10 instructions & Check (Manual & Split I_1-I_5 & I_6-I_{10} method)

- Hazard - Threat to system

I_1 S_1 | S_1

I_1 : ADD r_1, r_2, r_3
I_2 : SUB r_1, r_1, r_4

I_1 : IF ID AR MEM RW
I_2 : X IF ID AR MEM RW

$\therefore$ Data Hazard

Hardware $\longrightarrow$ Data accessed into buffers (Overlooking) $\rightarrow$ AKA Bypassing

LOAD $\longrightarrow 4^{rth}/5^{th}$
ADD $\longrightarrow 3^{rd}$

Bubbles $\longrightarrow$ Skipping pipeline instruction to align.

Q) n = 18 instructions

k = 4 stages

t = 1 cycle

Each cycle ⟶ 10 ns (Wait for 10 ns between each cycle)

for 1 cycle ⟶ 100 ns

T for pipeline ?

$(18 + 4 - 1) \times 100$ without interleaving time

Time for pipelining :

$(18 + 4 - 1) \times (100 + 10) - 10$

↳ for the last cycle

$(n + k - 1)$ times

Sol : $[(18 + 4 - 1) \times 100] + [(18 + 4 - 2) \times 10]$

H.W

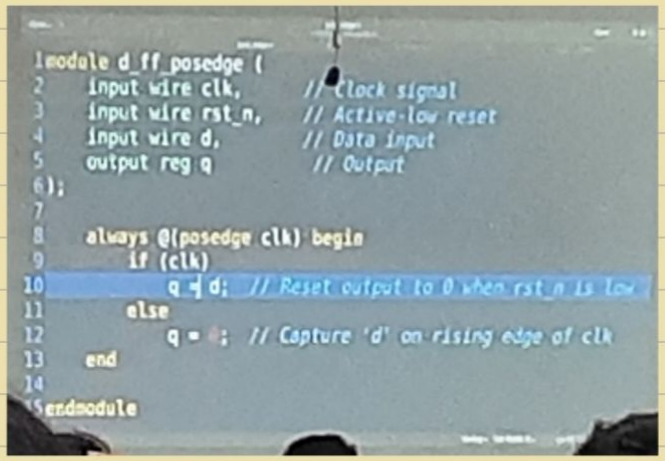
Q) n = 18 instructions

k = 4 stages

t = 3, 4, 3, 4, 5

1 cycle ⟶ 100 ns

17/3



```
module d_ff_posedge (
    input wire clk,      // Clock signal
    input wire rst_n,    // Active-low reset
    input wire d,        // Data input
    output reg q         // Output
);

    always @(posedge clk) begin
        if (clk)
            q <= d;  // Reset output to 0 when rst_n is low
        else
            q <= d;  // Capture 'd' on rising edge of clk
    end

endmodule
```

sudo apt-get install gtkwave

gtkwave counter.vcd

↳ dumpfile nam

18/3

- Bypassing AKA forwarding : Skipping part of instructions

t_1	t_2	t_3	t_4	t_5
IF	REG	ALU	MEM	REG
	IF	REG	ALU	MEM	REG

Skip the entire instruction ⟶ Bubbles

- Hazards :
  (1) Data Hazard
  (2) Structural Hazard
  (3) Control Hazard

— Control Hazard :

```
CMP r1, #0
BEQ FACT
```

IF REG ALU MEM REG
IF REG ALU MEM REG } Correct

IF REG ALU MEM REG
IF REG ALU MEM REG } Usual, But wrong

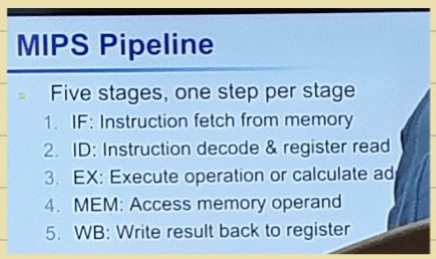
∴ Solution : Bubbles

Solution : Branch prediction (Precalculation)

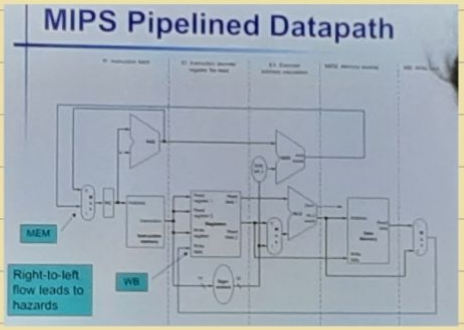
-

		IF	REG	ALU	MEM	REG
ADD	r_1, r_2, r_3	IF	REG $\{r_2, r_3\}$	ALU	MEM	REG $\{r_1\}$
SUB	r_4, r_1, r_2		IF	REG	ALU	MEM	REG
ADD	r_5, r_4, r_1			IF	REG	ALU	MEM	REG
SUB	r_6, r_1, r_2				IF	REG	ALU	MEM	REG
STR	$r_6, [r_7, \#4]$					IF	REG	ALU	MEM	REG

21/3

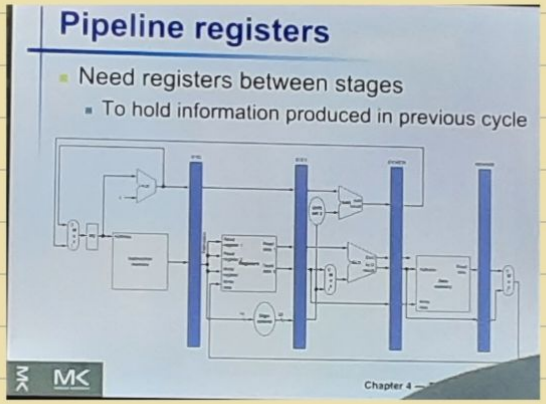


MIPS Pipelined Datapath

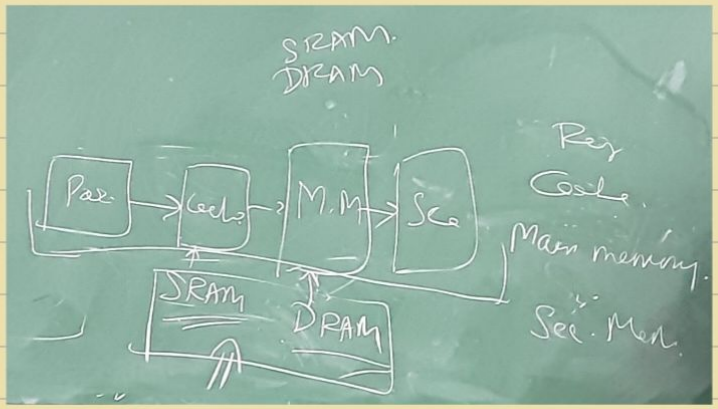


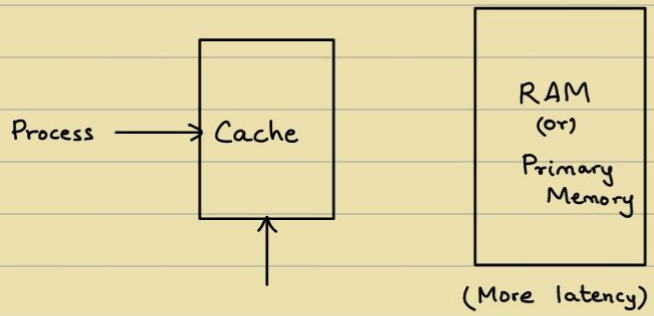
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



4/4





hit / miss

Effective Access Time

A B 100

10 sec 100 sec

0.4 0.6

$0.4 \times 100 \times 10$

Latency : Overhead (For understanding - Delay)

9) Hit rate

Miss rate : 40%.

hit → 1 cycle

RAM → 10 cycles

no. of instructions = 100

EAT : $(100 \times 1) + (0.4 \times 100 \times 10)$

$= 500$ cycles

Q) Cache : 10 ns

Memory : 40 ns

hit rate : 90 %.

EAT : $\left(\frac{10}{100} \times 40\right) + (10 \times 1)$

$= \underline{14}$

Cache : Has to be small

∵ Expensive,

Sequential processing ⟶ More Latency ⟶ More time taken to access

⟶ Fast access for small chunk of data

∴ Two caches : L_1 cache and L_2 cache

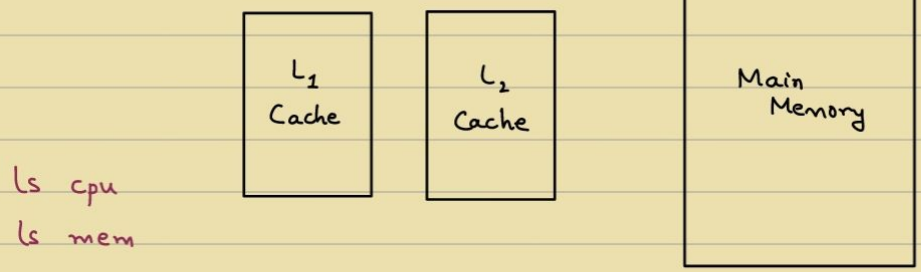
L_1 : Nearest to the processor

Speed :

$L_1 > L_2 > L_3 > L_4$

Size :

$L_1 < L_2 < L_3 < L_4$

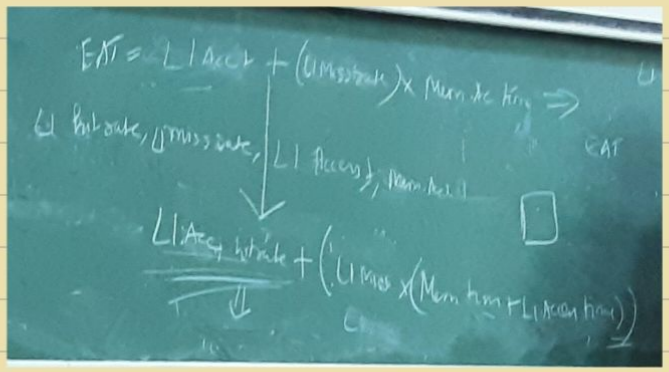


ls cpu

ls mem

L_1 hit, L_1 miss, L_1 Miss penalty, L_1 Hit time

4/4



Q) L_2 Miss : 2%

L_1 Miss : 4%

L_1 Access : 10 cycles

L_2 Access : 20 cycles

MMAT : 100 cycles

$$(0.2 \times 20) + (0.4 \times 10)$$

Q) L_1 Miss : 3%

L_1 Access time : 10 cycles

Main mem Access time : 100 cycles

$$(1 \times 10) + (0.3 \times 10) + (100 \times 1)$$

(L_1 Miss rate x L_1 access time) + (L_1 Miss rate x Main mem Access Time)

Q) 1.5 cycles ⟶ 1 memory access

1000 instructions

1 instruction : 3 cycles

MAT = 100 ns

L_1 AT = 10 ns

L_2 AT = 20 ns

L_1 M = 4%

L_2 M = 2%

} EAT = ?

Q) If no L_2 :

$$10 + (4\% \times 2000) = \underline{90}$$

3000 cycles : 2000 memory access

If L_2 :

$$10 + 4\%.(20 \times 1 + 2\%.(100))$$

Q) How many cycles to access L_2 cache ?

7/4

Interrupt : External Disturbance

Exception : Internal Disturbance

Handling Exceptions

In MIPS, exceptions managed by a System Control Coprocessor (CP0)

Save PC of offending (or interrupted) instruction

- In MIPS: Exception Program Counter (EPC)

Save indication of the problem

- In MIPS: Cause register
- We'll assume 1-bit
  - 0 for undefined opcode, 1 for overflow

Jump to handler at 8000 00180

An Alternate Mechanism

Vectored Interrupts

- Handler address determined by the cause

Example:

- Undefined opcode: C000 0000
- Overflow: C000 0020
- ...: C000 0040

Instructions either

- Deal with the interrupt, or
- Jump to real handler

Exceptions in a Pipeline

Another form of control hazard

Consider overflow on add in EX stage

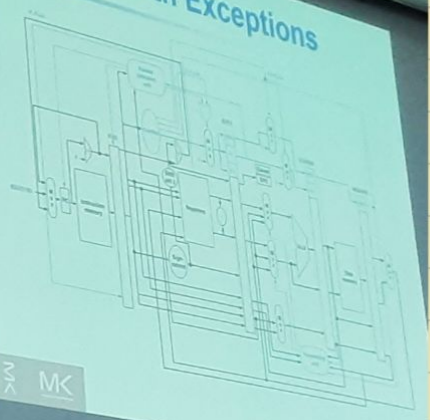
```
add $1, $2, $1
```

- Prevent $1 from being clobbered
- Complete previous instructions
- Flush add and subsequent instructions
- Set Cause and EPC register values
- Transfer control to handler

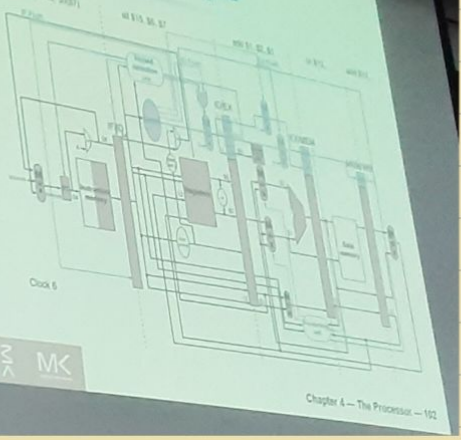
Similar to mispredicted branch

- Use much of the same hardware

Pipeline with Exceptions



Exception Example



11/4

128 KB → Cache

Main memory : 64 GB → $2^6 \times 2^{10} \times 2^{10} \times 2^{10}$

↳ 1 word = 4 bytes

1 block = 1 word

tag = ?

68, 128, 64, 83, 128 → Hit/Miss

Soln :

64 GB → $2^6 \times 2^{10} \times 2^{10} \times 2^{10}$

= 36 bits

128 KB — $2^7 \times 2^8 \times 2^2$

36 − 15 − 2 = 19 bits

↳ tag

8 + 7 = 15

↳ Index

4 byte

2 : Offset

↳ Blocks = 2^{15}

Index : 15

(Multi-word mapping)

64 KB ⟹ Cache

⟹ $2^6 \times 2^{10} \times$ byte

⇒ 2^{16} ⟹ 16 blocks

$2^{16}/2 = 2^{15}$ sets

32 bit : 16 + 16 + 0

32 bit : 16 + 2 + 14

↳ 2 bits

32 bits : 16 + n

↳ n ways

Size of cache : constant

∴ tag → constant

& index → changes

Q) 128 KB ⟶ cache

⟶ 1 byte

↳ 8 way associative mapping

2^{17}

64 GB ⟶ $2^6 \times 2^{10} \times 2^{10} \times 2^{10}$

= 36 bits

128 KB — $2^7 \times 2^{10}$

$2^{17}/2^3 = 2^{14}$

↳ Index : 14

Set : 3

36 − 14 − 3 = 15 bits

↳ tag

2^{14} sets

	0	1
Set 0	4	8
Set 1		
Set 2		
Set 3	7	15

3 % 4 = 3

4 % 4 = 0

15 % 4 = 3

0	8
1	
2	
3	3
4	4
5	
6	
7	7

tag + set ⟶ no. of checks

Q) 3, 4, 15, 8, 14, 16, 11, 16

8 blocks

↳ 1 block = 1 byte

2 way set associative mapping

0	8
1	
2	
3	3
4	4
5	
6	14
7	15

$3 \% 8 = 3$ $\quad 3 \% 4 = 3$

$4 \% 8 = 4$ $\quad 4 \% 4 = 0$

$15 \% 8 = 7$ $\quad 15 \% 4 = 3$

$8 \% 8 = 0$ $\quad 8 \% 4 = 0$

$14 \% 8 = 6$ $\quad 14 \% 4 = 2$

$16 \% 8 = 0 \longrightarrow$ Miss $\longleftarrow 16 \% 4 = 0$

$11 \% 8 = 3$ $\quad 11 \% 4 = 3$

$8 \% 8 = 0 \longrightarrow$ Hit

$16 \% 8 = 0 \longrightarrow$ Miss $\quad 16 \% 4 = 0 \longrightarrow$ Hit

	0	1
Set 0	4	8
Set 1		
Set 2	14	
Set 3	3	15

$\Rightarrow$

	16	8
	14	
	3	15

$\Rightarrow$

	16	8
	14	
	11	15

15/4

→ Virtual Memory :

For processor → It exists

For RAM → It does not exist

Virtual Address (Pages) → Disk (CPU uses)

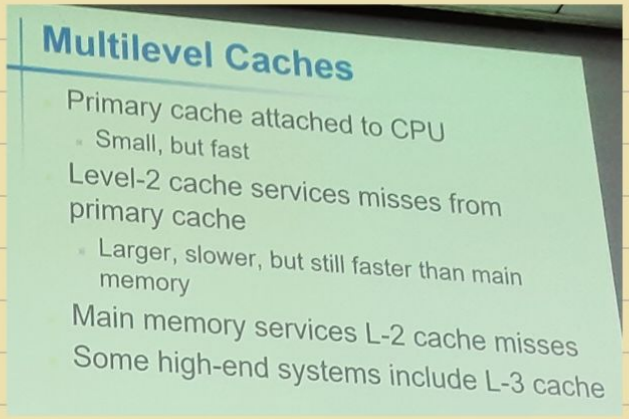
Physical Address (Frames) → RAM

TLB (Transational look-aside Buffer)

- Cache
- Stores page number (I) ⇔ frame number (O) correspondence
- Virtual Address

Index → Page Table

16/4

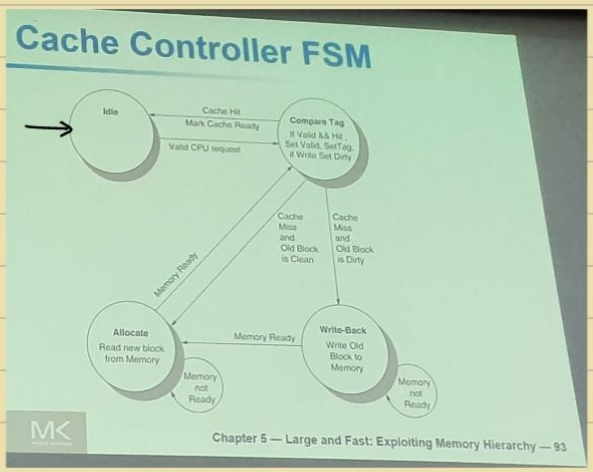


L1 → Fast access

More hit rate ⇒ Better

Nearer to CPU

L2 → Access more data



Valid bit != 1 ⇒ Proper data & is read

Dirty bit : Whether the address of content of valid bit is updated or not. (written or not)

Tag bit : Block updated in the memory

Coherence

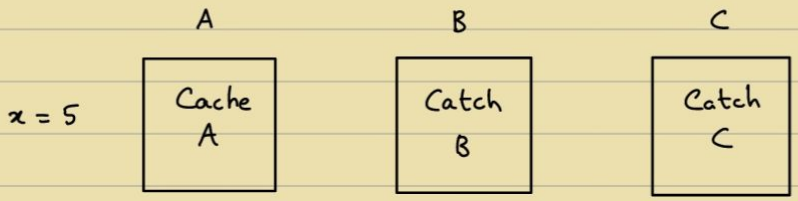
Consistency

Cache consistency : Two caches accessing same shared memory

Snooping : Lock the main memory such that all existing data in the other caches are invalidated.

∴ No Dirty read

16/4



Shared / Local

Data invalidated

Snooping ⟶ Cache coherence

AMAT : Average Memory Access Time

EAT : Effective Access Time

Q) L_1, L_2

L_1 MR : 4 %	L_1 Penalty : 10 cycles
L_2 MR : 6 %	L_2 Penalty : 80 cycles

L_1 cache : 2 cycles (Hit)

60 % ⟶ Access memory

⟶ Hit

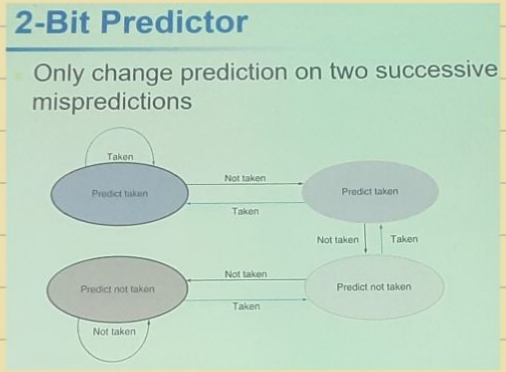
$(2 \times 1) + (0.04)(1 \times 10 + 0.06 \times 80)$

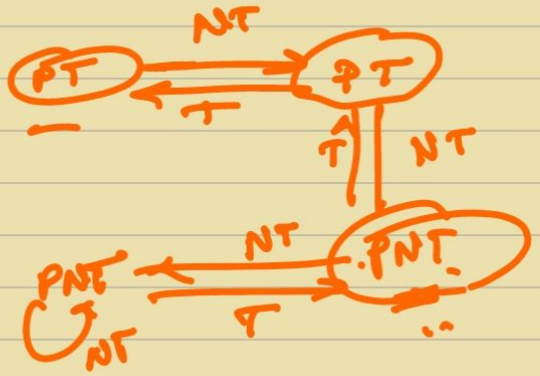
$\Rightarrow 2 + 0.04(14.8)$

$\Rightarrow 2.59 \cdot i$

Write : 10

17/4



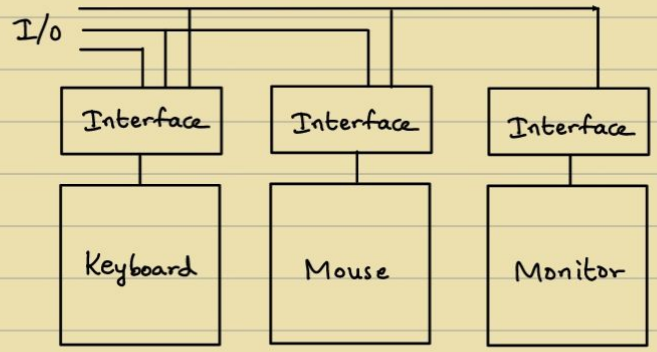


CPU ⟶ very fast

Keyboard, Mouse ⟶ Moderately slow

Printer, Speaker ⟶ Slow

- Control Bus:



⟶ Control (To synchronize both the CPU and devices)

- Modes of operations:

① Programmed I/O

CPU ⟶ breaks in interval ⟶ leave & start again

② Interrupt - Driven I/O

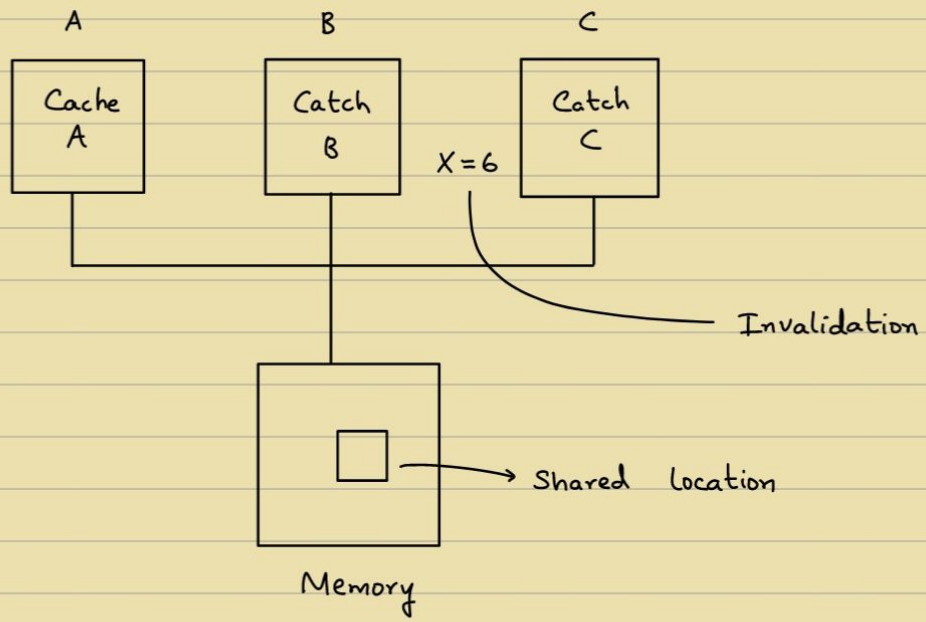
Requires access to the CPU ⟶ I/O device calls CPU

③ DMA: (Direct Memory access)

Takes access of the bus, CPU just waits

All operations are done by it's own.

22/4



Process synchronization

Block ← lock

n memory units

→ DMA:

(1) Programmed I/O
(2) Interrupt Driven I/O
(3) DMA

Peripherals → I/O

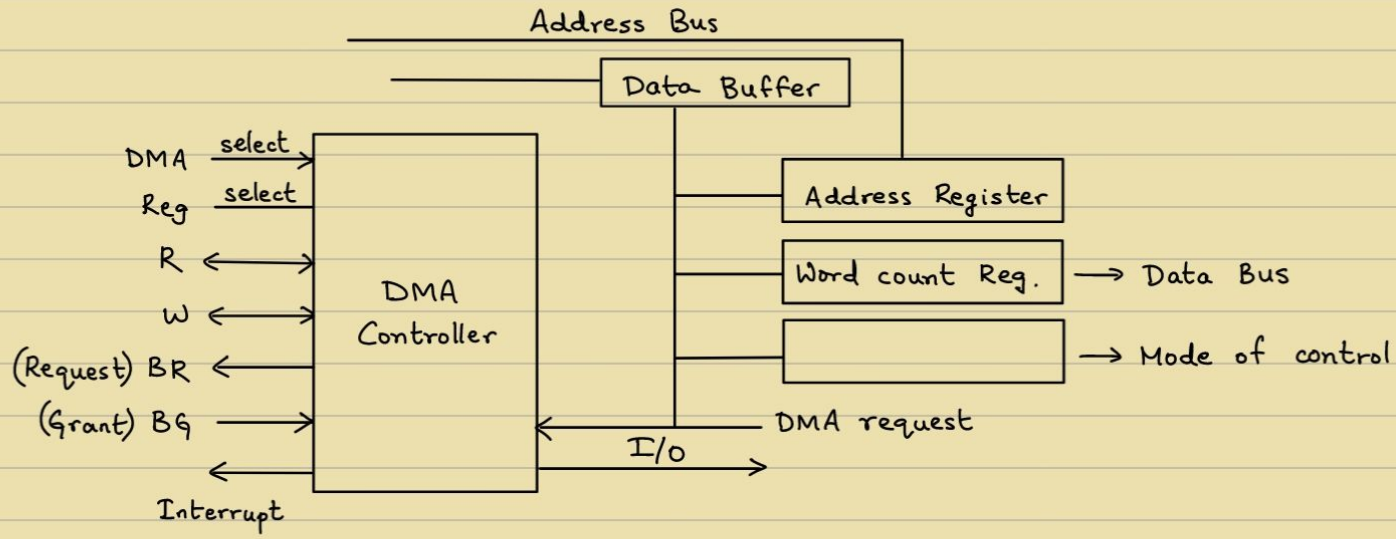
Boot loader → ROM
Microsoft : MBE
Linux : GRUB

Virtual Machine → Ubuntu (LTS)

Linux Distributions

Debian
Ex. Ubuntu

RHEL
Ex. Fedora



Interrupt Handler [Diagram]

KDE
GNOME

EPC

Cause