CSE-312 COMPUTER ARCHITECTURE

YASH VERMA 19075083 CSE B.tech

1.)

(A) Cache size = 2^R

My roll no. = 19075083

Now, R = 1 + (3 % 3)

= 1

∴ Cache size = $2^1 = 2$

→ Initial state:

Index	V	Tag	Data
0	0		
1	0		

→ Access 000100 → Cache Miss

Index	V	Tag	Data
0	1	00010_2	memory (00010_2)
1	0		

→ Access 100101 → Cache Miss

Index	V	Tag	Data
0	1	00010_2	Memory(000100_2)
1	1	10010_2	Memory(100101_2)

→ Access 010110 → Cache Miss

Index	V	Tag	Data
0	1	01011_2	Memory(010110_2)
1	1	10010_2	Memory(100101_2)

→ Access 001111 → Cache Miss

Index	V	Tag	Data
0	1	01011_2	Memory(010110_2)
1	1	00111_2	Memory(001111_2)

→ Access 101110 → Cache Miss

Index	V	Tag	Data
0	1	10111_2	Memory(101110_2)
1	1	00111_2	Memory(001111_2)

(1)

(B) 2-way set associative caching

Initially:

Set	LRU	Way - 1 V	Way - 1 Tag	Way - 0 V	Way - 0 Tag
000	0	0		0	
001	0	0		0	
010	0	0		0	
011	0	0		0	
100	0	0		0	
101	0	0		0	
110	0	0		0	
111	0	0		0	

Memory-address accesses:

(i) 000100 - cache miss

Set	LRU	Way - 1 V	Way - 1 Tag	Way - 0 V	Way - 0 Tag
100	1	0		1	000_2

(ii) 100101 - Cache miss

Set	LRU	Way - 1 V	Way - 1 Tag	Way - 0 V	Way - 0 Tag
101	1	0		1	100_2

(iii) 010110 – Cache Miss

Update in Table: Way-1 . Way-0

Set	LRU	V	Tag	V	Tag
110	1	0		1	010_2

(iv) 001111 – Cache miss

Set	LRU	V	Tag	V	Tag
111	1	0		1	001_2

(v) 101110 – Cache Miss

Set	LRU	V	Tag	V	Tag
110	0	1	101_2	1	010_2

Final Table:

Set	LRU	Way-1 V	Way-1 Tag	Way-0 V	Way-0 Tag
000	0	0	$(101)_2$	0	
001	0	0	$(101)_2$	0	
010	0	0	$(101)_2$	0	
011	0	0	$(101)_2$	0	
100	1	0	$(101)_2$	1	$(000)_2$
101	1	0	$(101)_2$	1	$(100)_2$
110	0	1	$(101)_2$	1	$(010)_2$
111	1	0	$(101)_2$	1	$(001)_2$

4-way set associative Caching

Set	LRU	Way 3 V	Way 3 Tag	Way 2 V	Way 2 Tag	Way 1 V	Way 1 Tag	Way 0 V	Way 0 Tag
00	0	0		0		0		0	
01	0	0		0		0		0	
10	0	0		0		0		0	
11	0	0		0		0		0	

(iii) Access '000100 – Cache Miss

Set	LRU	Way 3 V	Way 3 Tag	Way 2 V	Way 2 Tag	Way 1 V	Way 1 Tag	Way 0 V	Way 0 Tag
00	1	0		0		0		1	$(0001)_2$
01	0	0		0		0		0	
10	0	0		0		0		0	
11	0	0		0		0		0	

(iv) Access 100101 – Cache Miss

Set	LRU	Way 3 V	Way 3 Tag	Way 2 V	Way 2 Tag	Way 1 V	Way 1 Tag	Way 0 V	Way 0 Tag
00	1	0		0		0		1	$(0001)_2$
01	1	0		0		0		1	$(1001)_2$
10	0	0		0		0		0	
11	0	0		0		0		0	

(iii) Access 010110 – Cache Miss

Set	LRU	Way-3 V	Way-3 Tag	Way-2 V	Way-2 Tag	Way-1 V	Way-1 Tag	Way-0 V	Way-0 Tag
00	1	0		0		0		1	$(0001)_2$
01	1	0		0		0		1	$(1001)_2$
10	1	0		0		0		1	$(0101)_2$
11	0	0		0		0		0	

(iv) Access 001111 – Cache Miss

Set	LRU	Way-3 V	Way-3 Tag	Way-2 V	Way-2 Tag	Way-1 V	Way-1 Tag	Way-0 V	Way-0 Tag
00	1	0		0		0		1	$(0001)_2$
01	1	0		0		0		1	$(1001)_2$
10	1	0		0		0		1	$(0101)_2$
11	1	0		0		0		1	$(0011)_2$

(v) Access 101110 – Cache Miss

Set	LRU	Way-3 V	Way-3 Tag	Way-2 V	Way-2 Tag	Way-1 V	Way-1 Tag	Way-0 V	Way-0 Tag
00	1	0		0		0		1	$(0001)_2$
01	1	0		0		0		1	$(1001)_2$
10	2	0		0		1		1	$(0101)_2$
11	0	0		0		0		1	$(0011)_2$

(2)

(A) Page size = 2^R kB

R = Last digit of Roll No. (19075083) % 3

= 3 % 3

R = 0

Page size = 2^0 = 1 kB = 2^{10} = 1024

Entry in Page table = 4 bytes

We have 64-bit addressing scheme

→ No. of page table entries = $\frac{\text{address space size}}{\text{page size}}$

$= \frac{2^{64}}{2^{10}} = 2^{54}$

→ Size of page table = No. of entries × entry size

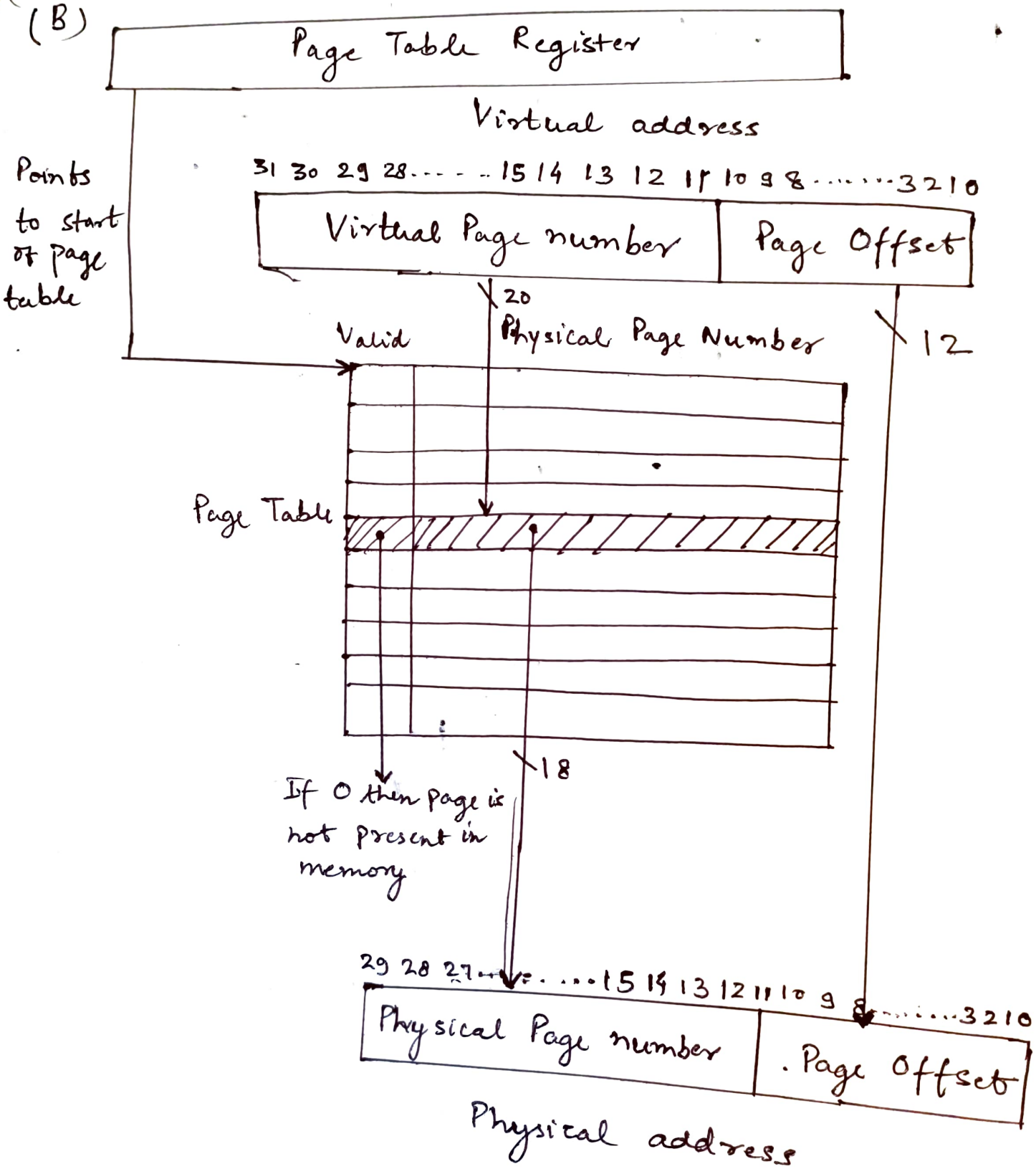
$= 2^{54} \times 4$

$= 2^{54} \times 2^2$

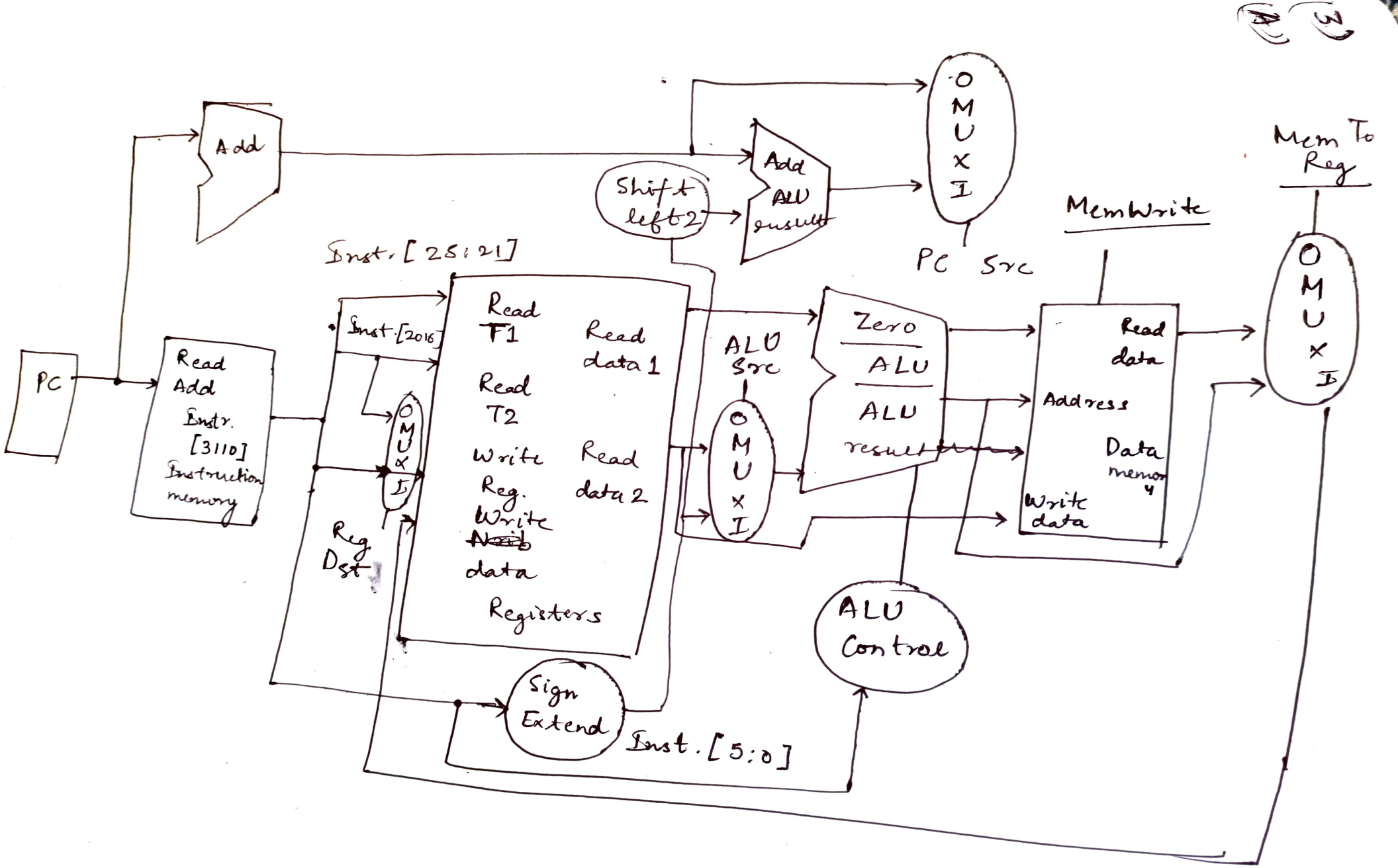
$= 2^{56}$ bytes

(2) Page Offset = 12 Page size = 1 kB = 2^{10} bytes

(B)



Page Table: Page size 4kB, virtual address space 4 GB, physical memory 1 GB.


Add
Add
ALU result
Shift left 2
M
U
X
PC Src
MemWrite
Mem To Reg
Inst. [25:21]
PC
Read Add
Instr. [3110]
Instruction memory
Inst. [2016]
Read F1
Read T2
Write Reg.
Write data
Read data 1
Read data 2
Registers
Reg Dst
ALU Src
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
ALU Control
Sign Extend
Inst. [5:0]

Given Instruction:

add $1, $2, $3

Steps for implementation of Single Cycle:

→ First we need to fetch the instruction, we need adder, register to data access, ALU for performing arithmetic operations and some multiplexers, memory.

→ After setting datapath for single cycle implementation. Instruction memory read address of program counter and in decode phase. Instructions are decoded to loaded to registers.

→ $2 register is read first, then $3 register is read for data 2. destinantion register is given by instruction [15:11]

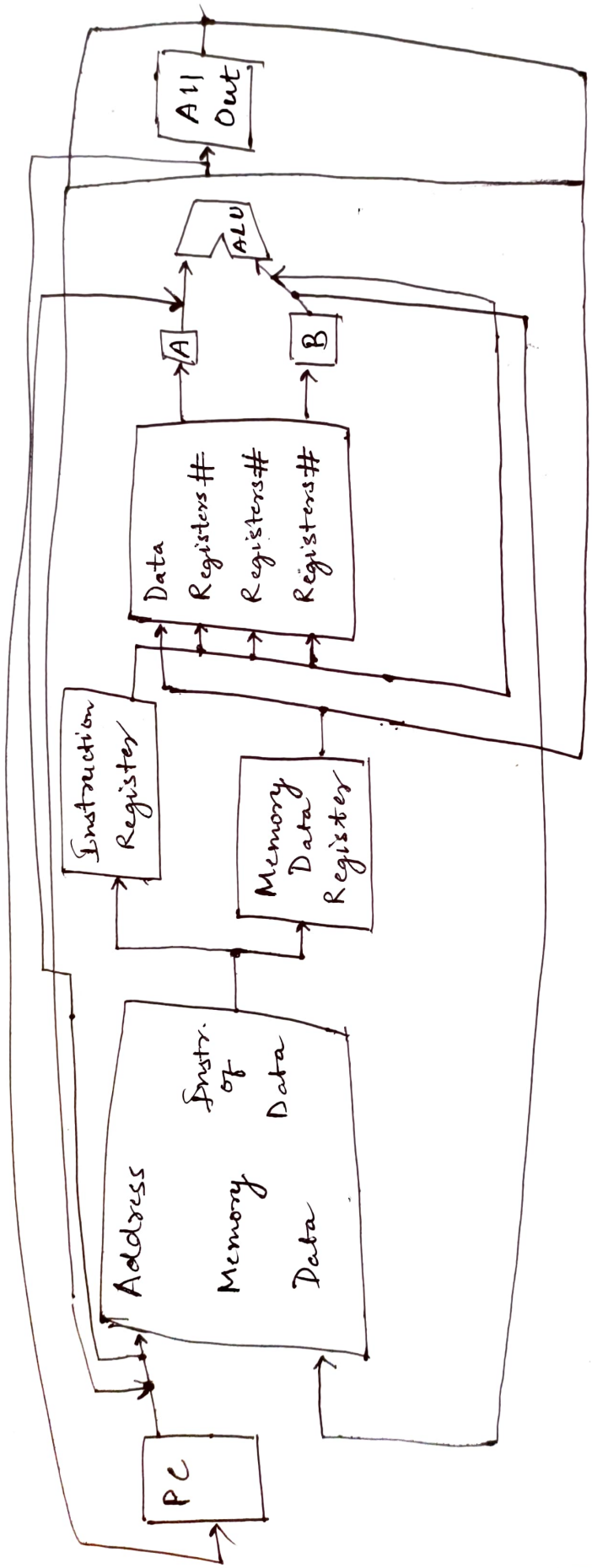
→ $2 and $3 values are loaded to ALU and with help of ALU op, operation is performed

→ Now, ALU result will be stored in register instead of data memory.

→ PC will be increased simultaneously.

5(3)(B)

sw $t5, 8($t3) in multi-cycle implementation



Multiple Cycle Approach requires

→ single memory for data & instruction

→ Single ALU, no extra address

→ extra register to hold data between cycles.

We break the instructions into following potential execution steps:

→ Instruction fetch and PC increment (IF)

→ Instruction decode & Register fetch (ID)

→ Execution, memory address, congatation or branch completion. (Ex)

→ Memory access or R-type instruction completion (MEM)

→ Memory rest completion

(4)

(A) functional unit delays :

memory – 3 ns

ALU and address – 3 ns

FPU add – 8 ns

FPU multiply – 18 ns

Register file access (read/write) – 2 ns

$R = 1 + (3 \% 5) = 1 + 3$ [Roll no. 19075085]

$\boxed{R = 4}$

Instruction:

Instruction memory → A

Register read → B

ALU operation → C

Data Memory → D

Register Write → E

FPU add → F (or ~~and~~ sub)

FPU multiply → G (or divide)

Instruction	A	B	C	D	E	F	G	Total
Load	3	2	3	3	2			13
Store	3	2	3	3				11
R-format	3	2	3	0	2			10
Branch	3	2	3					8
Jump	3							3
FP add/sub	3	2			2	8		15
FP mul/divide	3	2			2		16	23

(a) a single cycle implementation using a fixed period clock.

~~Ans~~

Ans: longest time (of an instruction)

= i.e. 23

(b) Variable period clock:

$= 13 \times 30\%$ (load) $+ 11 \times 20\%$ (store) $+ 10 \times 27\%$ (R-format) $+ 8 \times R\%$ (branch) $+ 3 \times (7-R)\%$ (Jump) $+ 15 \times 8\%$ (add/sub) $+ 23 \times 8\%$ (mul/div)

Now, ~~R ≠ 8~~, $R = 4$

$$= 13 \times \frac{3}{10} + 11 \times \frac{2}{10} + 10 \times \frac{27}{100} + 8 \times \frac{4}{100} + 3 \times \frac{3}{100}$$

$$+ 15 \times \frac{8}{100} + 23 \times \frac{8}{100} \quad . \quad . \quad .$$

$$= 3.9 + 2.2 + 2.7 + 0.32 + 0.09 + 1.2 + 1.84$$

$$= 12.25 \text{ ns}$$

Ans: 12.25 ns

Comparison:

$$\frac{\text{performance(a)}}{\text{performance(b)}} = \frac{23}{12.25} = 1.877$$

$$\approx 1.88$$

(4)
(b) In computing, Page Fault is an exception that the memory managment unit raises when a process accesses a memory page without proper preparations.

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So, when page fault occurs, following sequence of events occur:

→ The Computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.

→ An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.

→ OS finds that page fault has occured and tries to find out which virtual page is needed

→ Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.

→ If virtual address is valid, the system chicks to see if a page frame is free.

If no frames are free, the page replacement algorithm is run to remove a page.

→ If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.

→ As soon as, page frame is clean, OS look up disk address where needed page is, schedules disk operation to bring it in.

→ When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.

→ Faulting instruction is backed upto state it had when it began and PC is reset. Faulting is scheduled, OS returns to routine that called it.

→ Assembly routine reloads registers and other state information, returns to user space to continue execution.

(5)

(A) For this consider the following Computational Task :-

To compute the dot product of 2 vectors :-

Let V_1 and V_2 be the vectors of length l. Then, their dot product is defined as

$$\therefore \text{Dot product} = \sum_{i=0}^{n-1} V_1(i) \times V_2(i)$$

Thus, the program for computing the dot product of 2 vectors is :-

```
Move      #V1 Vec, R1     ; R1 points to vector V1
Move      #V2 Vec, R2     ; R2 points to vector V2
Move      #N, R3          ; R3 serves as Counter
Clear     R0              ; R0 accumulates the dot
                            product
Loop Move (R1)+, R4       ; Compute the product of next
Multiply  (R2)+, R4       ; components
Add       R4, R0          ; Add to previous sum
Decrement R3              ; Decrement the counter
Branch>0  Loop            ; Loop again if not done
Move      R0, DOTPROD     ; store dot product in
                            memory
```

→ The above program computes the dot product and stores the result in memory to location DOTPROD. The

The first elements of each vector, $V_1(0)$ and $V_2(0)$, are stored at memory locations V_1Vec and V_2Vec, with the remaining elements in the following word location.

The above programs was 5 registers. Instead of using R_0 to accumulate the sum, the sum can be accumulated directly into DOTPROD. This means that the last Move instruction in the program can be removed, but DOTPROD is read and written on each pass through the loop, significantly increasing memory access. The 4 registers, R_1, R_2, R_3 and R_4, are still needed to make this program efficient, and they are all used in the loop. Suppose that R_1 and R_2 are retained as pointers to the V_1 and V_2 vectors, Counter register R_3 and temporary storage register R_4. Could be replaced by memory location in a ~~2-to~~ 2-register machine, but the number of memory access would increase significantly.

Hence, we can say - "Having a large number of processor register makes it possible to reduce the no. of memory accesses needed to perform complex tasks."

5. (b) $s0 holds the value 128_{ten}.

(i) Instruction:

add $t0, $s0, $s1

Consider the following instruction set:

add $t0, $s0, $s1

The memory allocated to one ~~two~~ instruction is 4 bytes.

The range of the number can be calculated as 2^{n-1}.

n = no. of bits

n = 32

The range is $2^{31} - 1$ to -2^{31}

The range is 2,147,483,647 to -2,147,483,648

The overflow condition ~~ra~~ is as follows:

Range of $s1 = 2,147,483,647 - 128$

$= 2147483519$

After this value, the condition will overflow.

Thus, the overflow range of $s1 is as follows:

Overflow range ($s1) = [2,147,483,520] to [2,147,483,647]

(ii) Value of $s0 = 128_{10}

Consider the following instruction set:

sub $t0, $s0, $s1

The memory allocated to one ~~in~~ instruction is 4 bytes.

The range of number can be calculated as 2^{n-1}.

$n = 32$

The range is $2^{31} - 1$ to -2^{31}

The range is 2,147,483,647 to −2,147,483,648

The overflow condition is as follows:

Range of $s1 = −2,147,483,648 + 128$

$= -2,147,483,520$

Thus, overflow range of $s1 is:

Overflow range ($s1) = [−2,147,483,648 to −2,147,483,521]