Unit 3 - Computer Organization & Architecture

Control Path Implementation

ALU Control

The MIPS ALU defines the 6 combinations of 4 control units

	ALU control lines	Function
0	0 0 0 0	AND
1	0 0 0 1	OR
2	0 0 1 0	add
6	0 1 1 0	subtract
7	0 1 1 1	set on less than
12	1 1 0 0	NOR

→ Depending on the instruction class, the ALU will need to perform one of those first five functions

Load word & store word : ALU computes memory address by addition

R-Type instructions : ALU needs to perform AND, OR, subtract, add or set less than depending on the value of the 6-bit funct field in the instruction

Beq Instructions: ALU must perform a subtraction

* Generating the 4-bit ALU control input

→ use a control unit that inputs the function field of the instruction and a 2 bit control field called ALUOp.

→ ALU op indicates the operation performed by the ALU

add for load and stores = 00

subtract for beq = 01

determined by operation encoded in funct field = 10

→ The ALUOp along w/ the funct field => **4 bit signal** which can directly control the operation of the ALU.

Instruction opcode	ALU op	Instruction operation	Funct	desired ALU action	4-bit ALU control input
LW	0 0	load word	X X X X X X	add	0 0 1 0
SW	0 0	store word	X X X X X X	add	0 0 1 0
Beq	0 1	branch equal	X X X X X X	subtract	0 0 1 0
R-type	1 0	add	1 0 0 0 0 0	add	0 0 1 0
R-type	1 0	sub	1 0 0 0 1 0	subtract	0 1 1 0
R-type	1 0	AND	1 0 0 1 0 0	AND	0 0 0 0
R-type	1 0	OR	1 0 0 1 0 1	OR	0 0 0 1
R-type	1 0	set on less than	1 0 1 0 1 0	slt	0 1 1 1

0 → 10 → 100 → 101 → alt 0s & 1s

→ When the ALUOp is 00 or 01, the desired ALU action does not depend on the function code field → use don't cares

→ If ALUOp = 10, then the function code is used to determine the ALU control input

* Deriving ALU control signals using a truth table

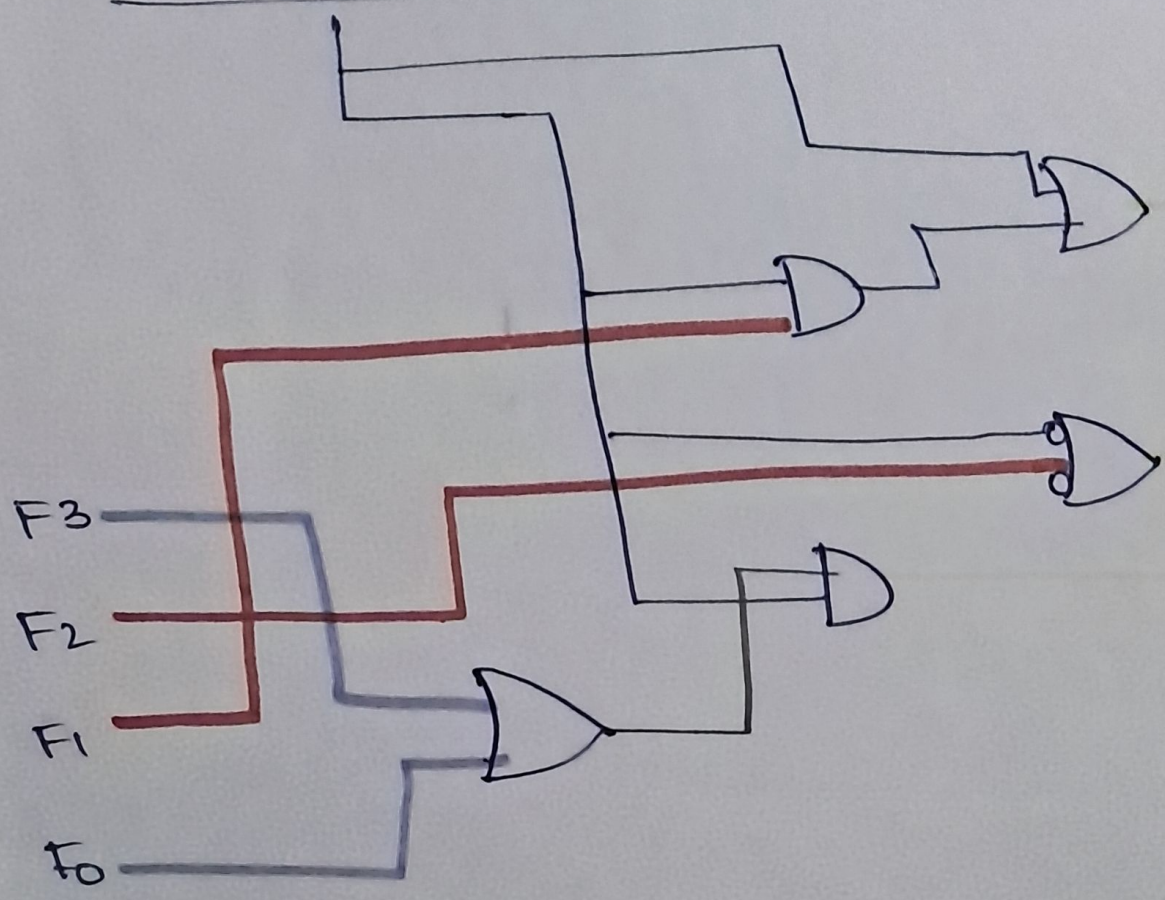
ALU op		Funct Field						Operation				from prev table
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0					
0	0	X	X	X	X	X	X	0	0	1	0	line1
0	1	X	X	X	X	X	X	0	1	1	0	line3
1	0	X	X	0	0	0	0	0	0	1	0	
1	X	X	X	0	0	1	0	0	1	1	0	
1	0	X	X	0	1	0	0	0	0	0	0	
1	0	X	X	0	1	0	1	0	0	0	1	
1	X	X	X	1	0	1	0	0	1	1	1	

1→10, then order as
1X
10
10
1X

(From the previous table
copy last 4-bits
MSB 2 bits = don't care)

* Mapping of ALU control functions to gates

only 3 operation o/p since MSB is always 0



Operation

Unit - 3

Building Data Paths

Datapath → refers to elements that process data and addresses in the CPU.

There a 5 main steps to execute any instruction

Step 1: (F)etch instruction

Step 2 : (I)nstruction decode and read registers

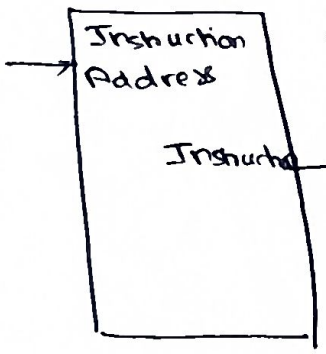
Step 3: (A)LU operation, branch address computation

Step 4 : (L)w / store in data memory

Step 5 : (R)egister Write

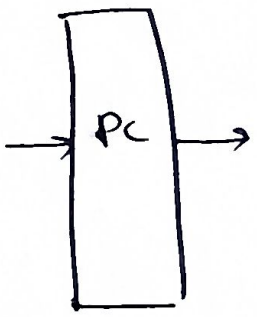
Components

(1) Instruction memory:



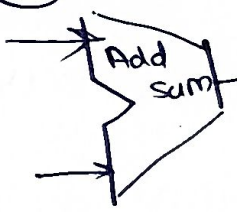
Stores instructions of a program & supplies instructions given an address

(2) Program Counter.

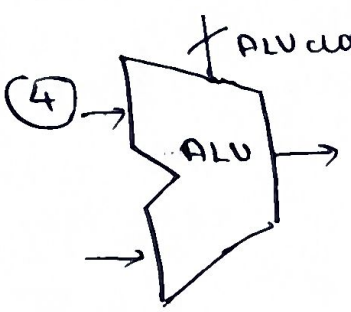


holds address of the next executable instruction

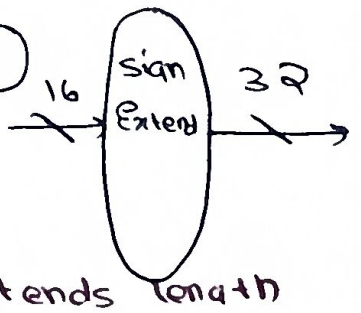
(3) Adder → increments PC to the address of the next instruction, can be built from ALU, by wiring control lines such that the ALU performs only add operations



(4)

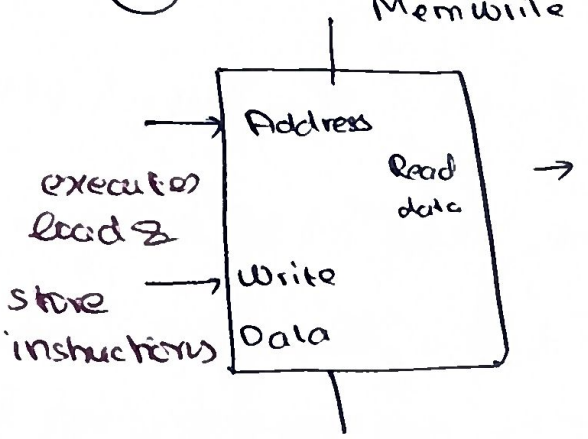


(5)

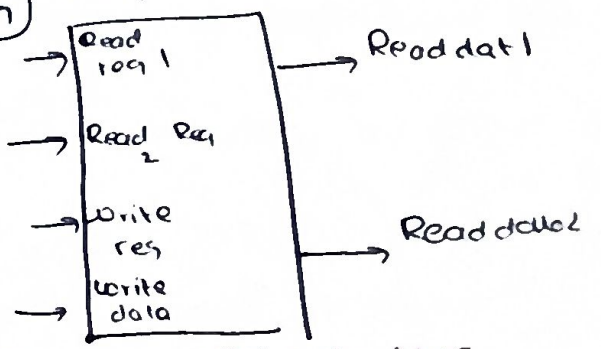


extends length from 16 - 32 bits

(6) Data Memory unit



(7) Register File



general purpose register

Stage 1: Instruction Fetch

→ Transfer contents of PC to MAR

→ Choose a particular memory location

→ Read instructions present in memory

→ Load instructions into instruction memory

→ Also increment the PC by 4 to execute the next instruction

Stage 2: Decode

→ Instruction present in IR will be decoded

→ If data available for the operation is available in the register, it performs the operation

Stage 3: Performing the Operation

This stage depends on the nature of instruction being executed.

Type 1: R-Type Operations (add, sub, AND, OR, slt)

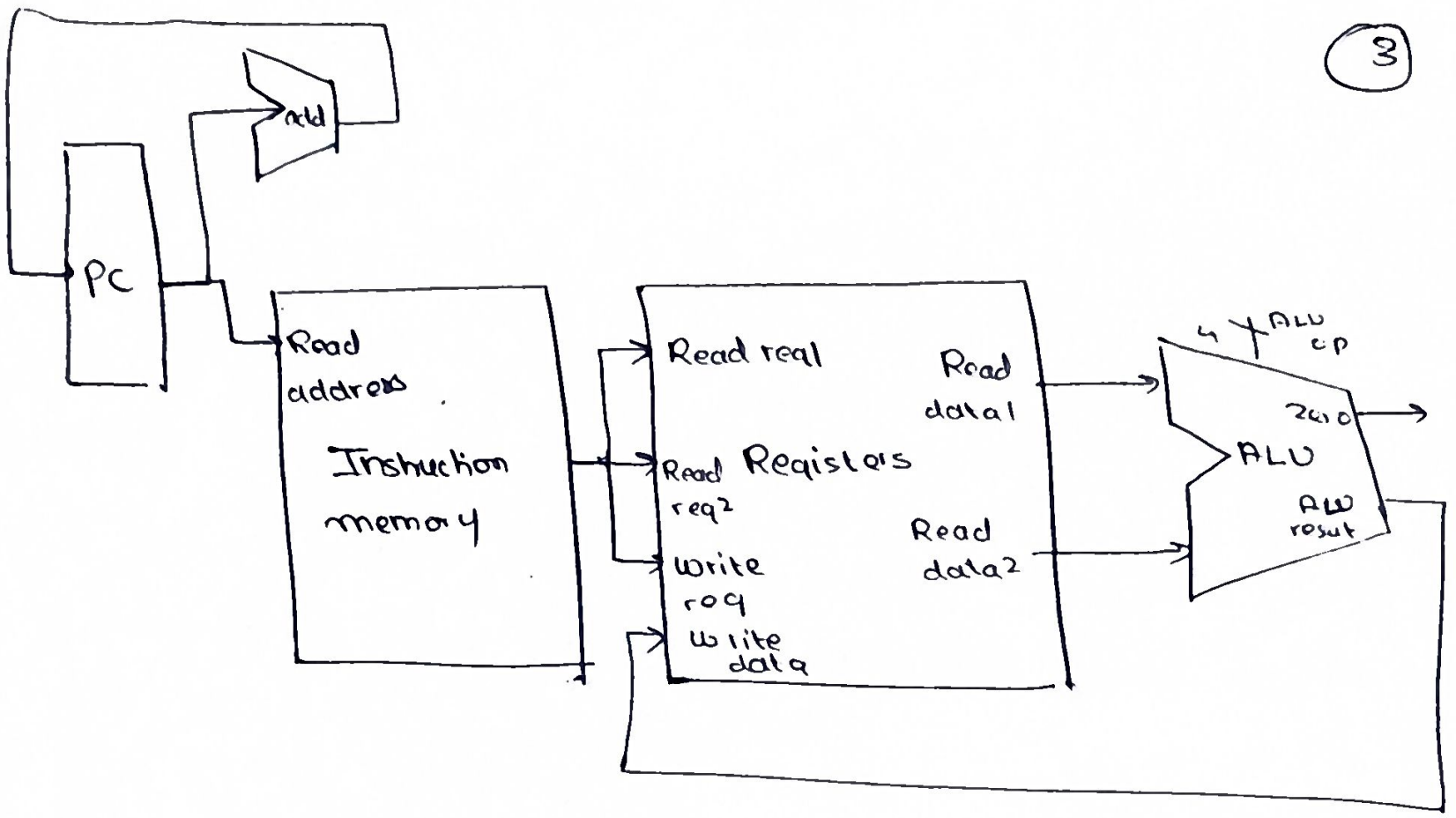
eg. add $t1, $a0, $s2

→ pick ~~instruction~~ source operands from instructions, specifically the corresponding reg. no

→ Read the corresponding data from the 2 source registers

→ Take read values to ALU

→ Take result of ALU to the <write data> portion of the register file. Also extract the write register no from the instructions, to know which register to write into



Type 2: Immediate Instructions

eg. addi $t0, $s0, 4

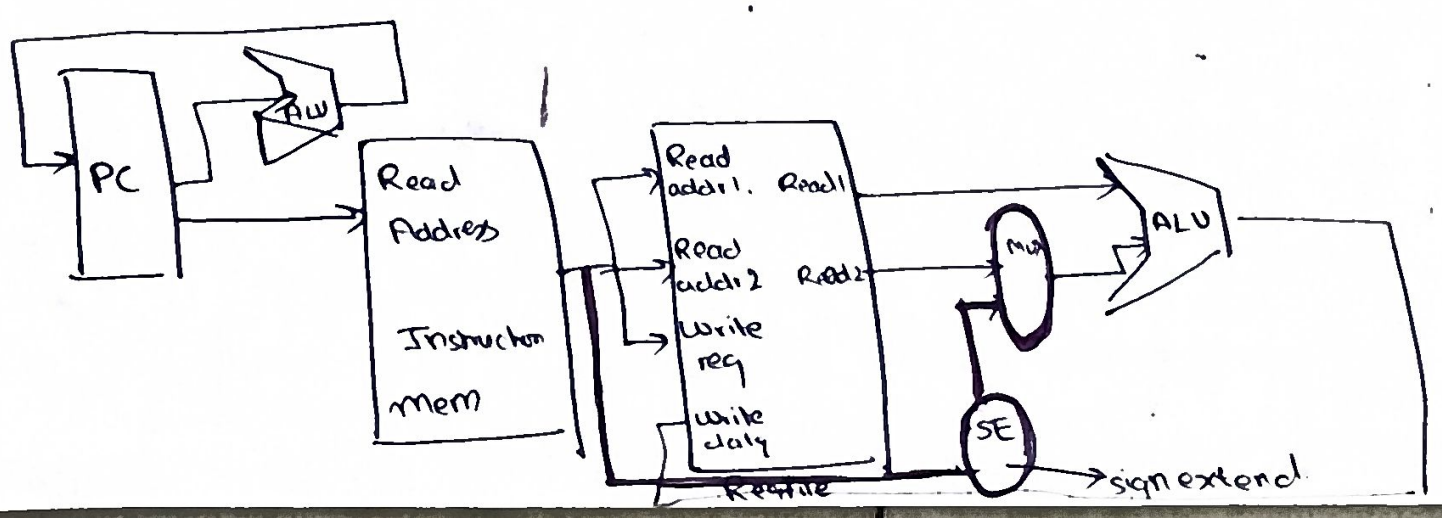
→ Ininally, the same as the R type instruction

→ This time, take the constant value from the instruction memory

→ Include a mux between the 2nd port to read data & the ALU.

→ This allows to choose between a value from the register and a constant directly from the IR

→ While taking the constant from the IM, and before passing to the ALU, include a sign-extender, to extend length from 16-32 bit



(3) Load Instructions

Step4: Reading / Storing in Data memory

eg. lw $s1 s0 , 8($t0)

offset

→ Initially, the base address (here $t0) and the offset 8 have to be added.

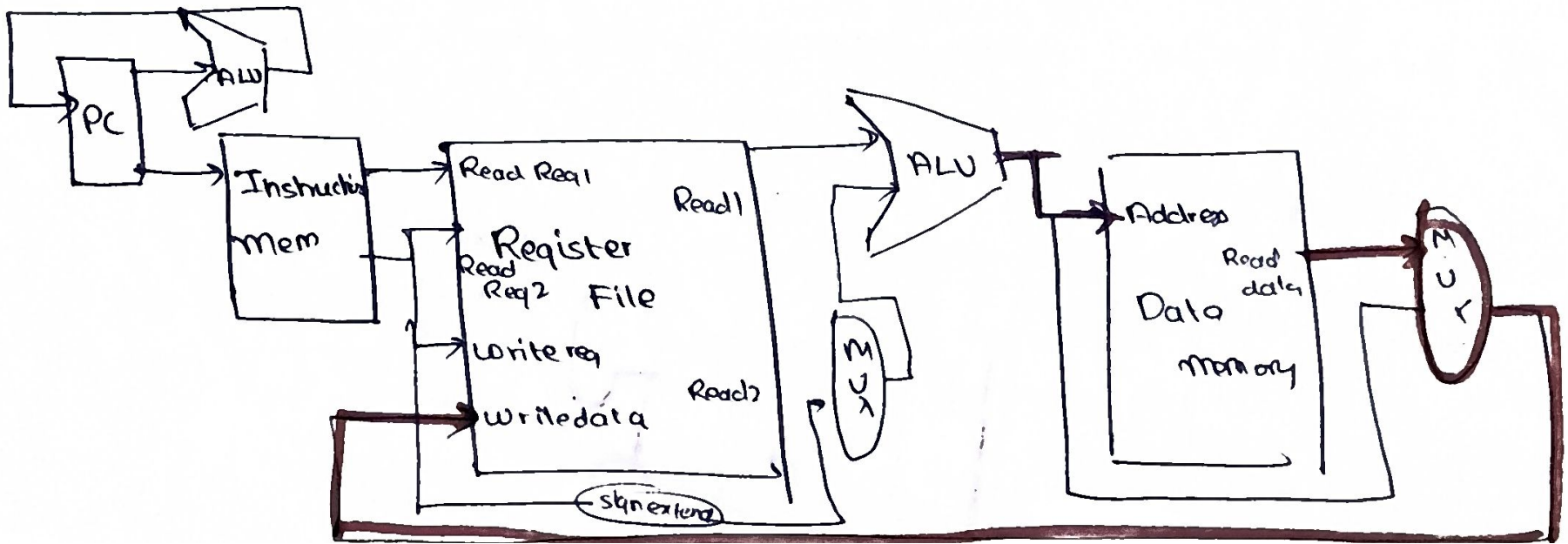
→ The base address will be present in the register files and the offset comes from the immediate portion (along w/ the MUX) ~~that col~~ portion

→ The ALU is used to add the offset to the value in the base register to find the load address. It is now present at the ALO result section

→ Pass the address from the result section to the data memory to load the corresponding value at the other end.

→ Add a MUX after the data memory. This is so that it chooses what comes from the ALU result in case of arithmetic operations, & and chooses the value after the offset in case of load instructions

→ Write back the final result into the register file

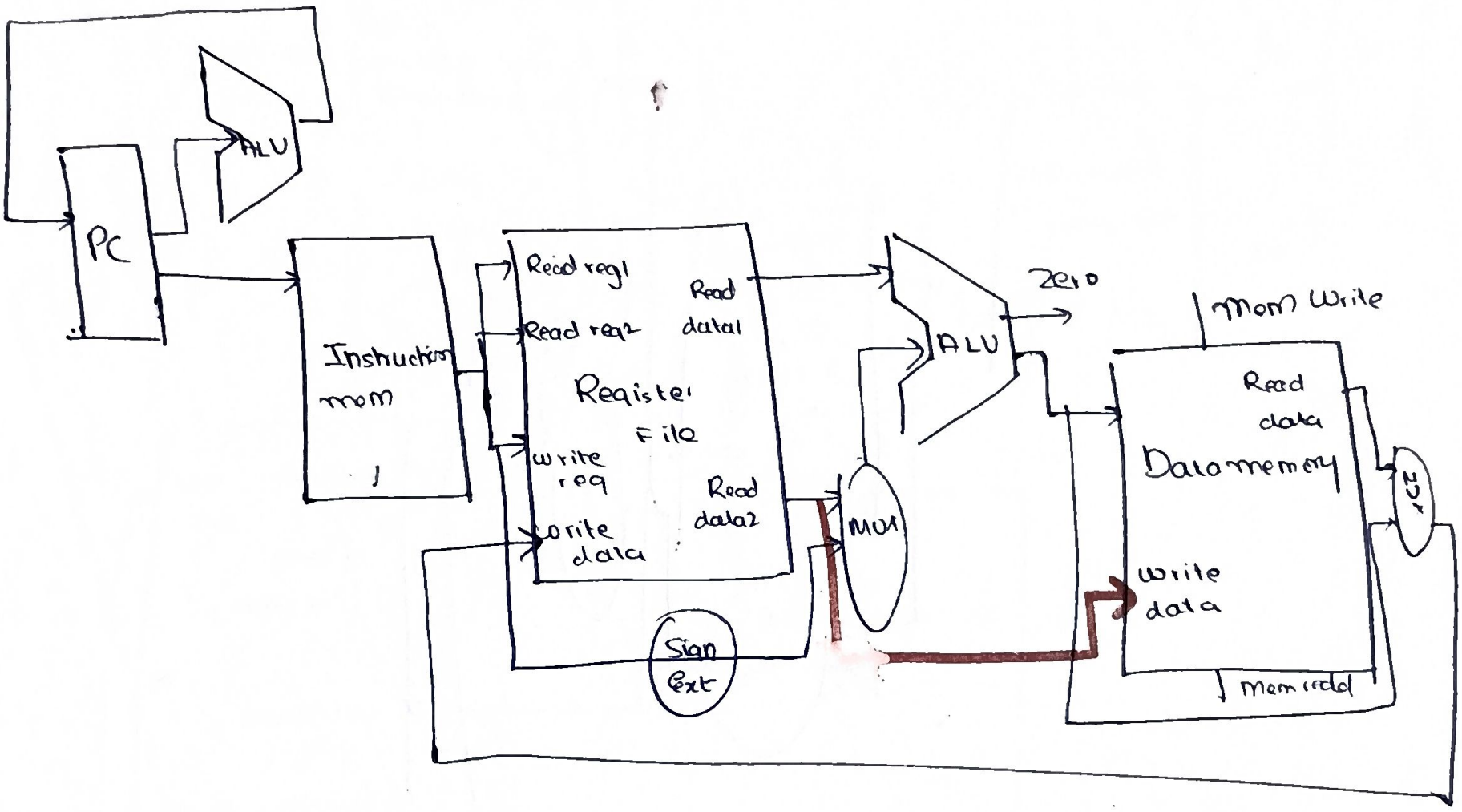


Step 4: Pt 2 Store word

eg. sw $a0, 8($sp)

→ Use ALU to add the offset to the value in the base register to find the store address.

→ Pass the value of the second register as the data to be written



Type 3: Branch Instructions

eg. beq $t0, $s0, offset

→ Use the ALU to check if the 2 source registers are equal or not equal (subtract & check if result is 0)

→ Find target address = PC + 4 + offset × 4

→ Offset available, since it is constant (i type)

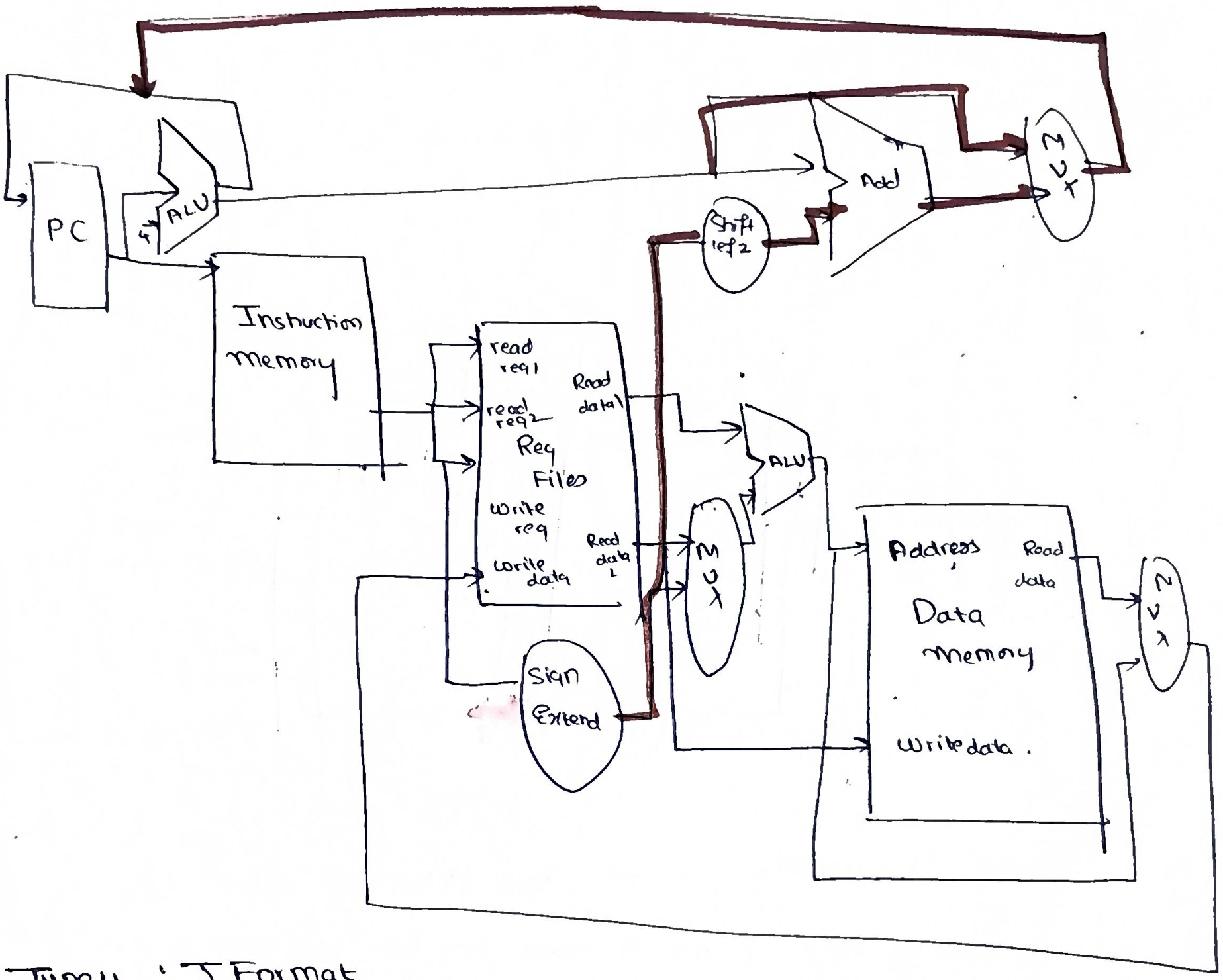
→ Multiply offset by 4 (aka shift left by 2)

→ Pass this, and the shift left result to an adder

→ The result has to be sent back to PC ⇒ if beq occurs, then send (PC+4) + (offset × 4), otherwise send PC+4

→ This is decided using a MUX .

Final Diagram



Type 4 : J Format

eg. j R1

→ There are 6 bits in the opcode

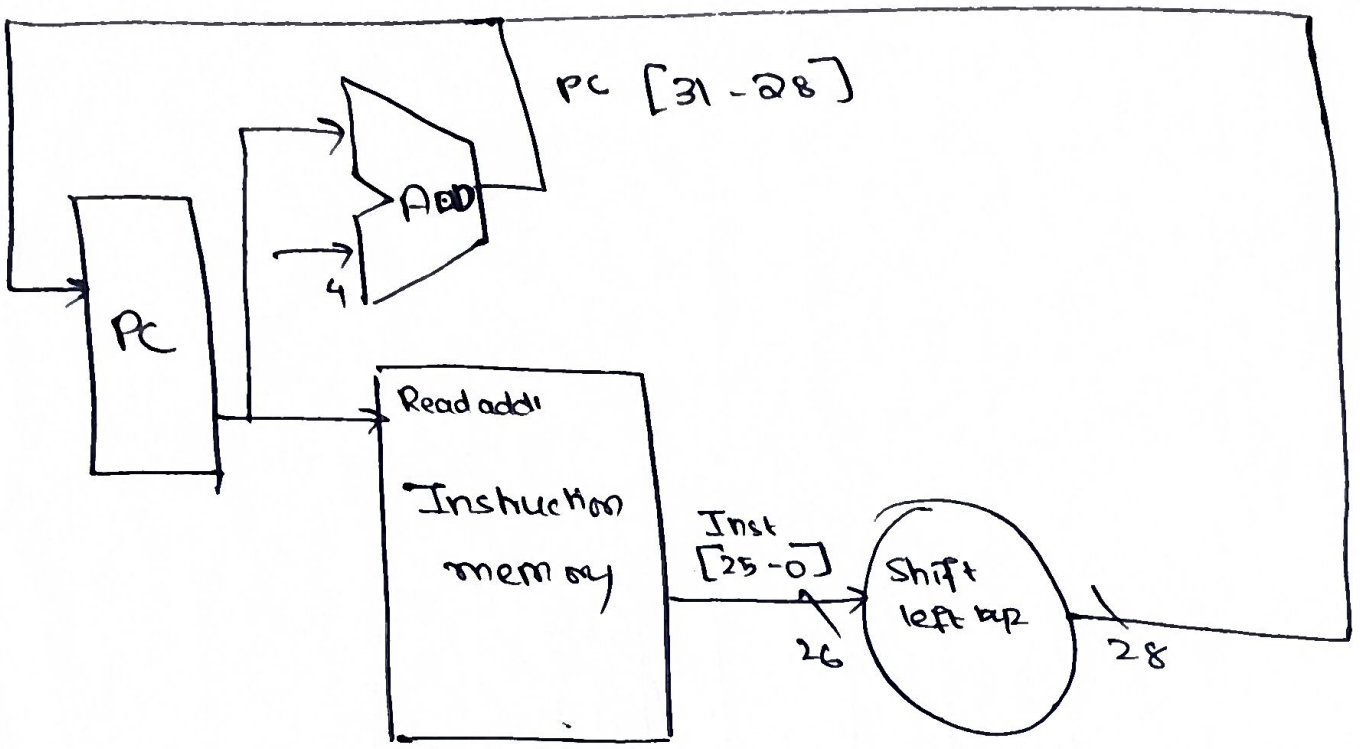
→ There are 26 bits in the target address

→ There is not enough space in the instruction to specify a full target address

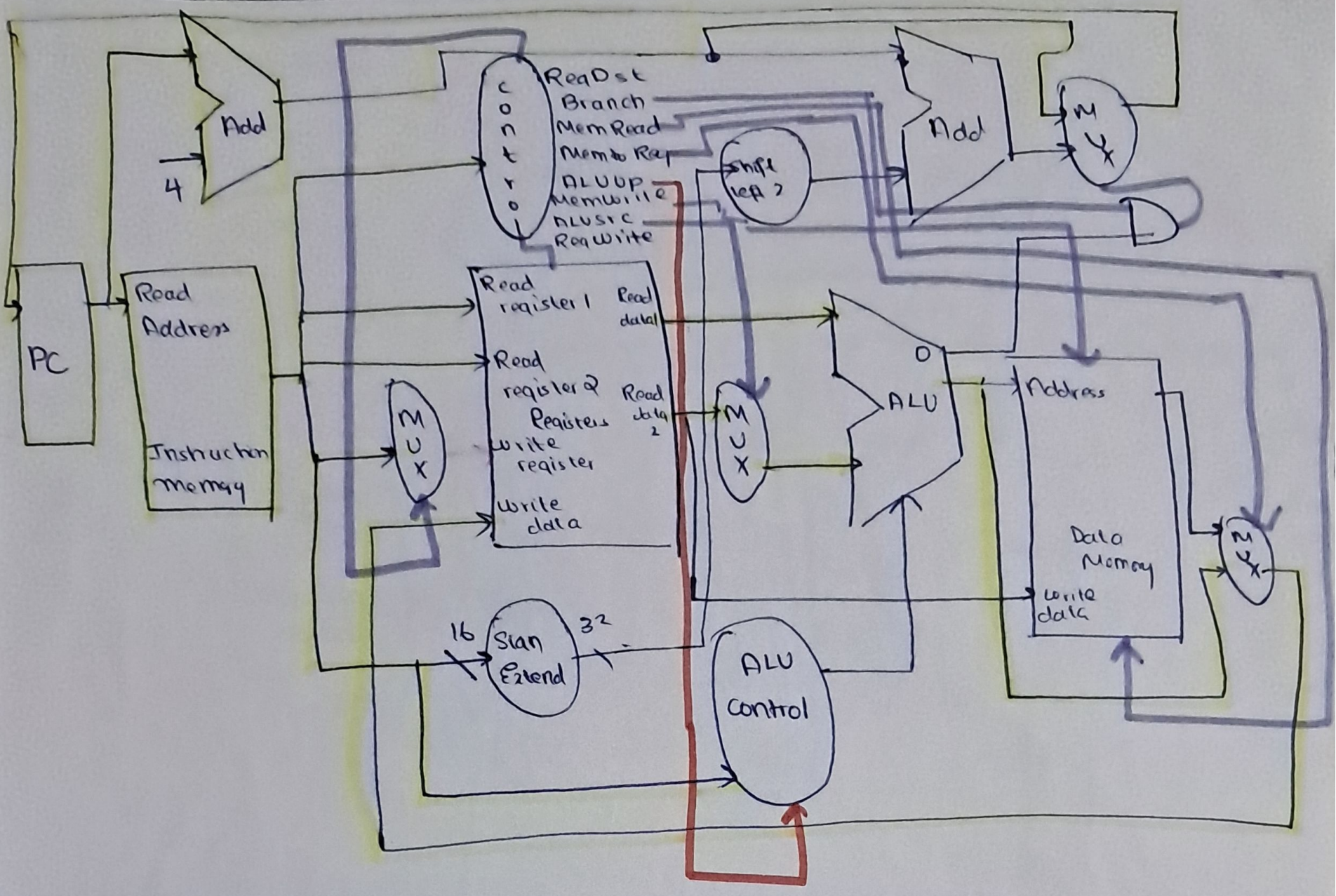
Solution

→ Take the 26 bit target address field, left shift by 2

→ concatenate with the result of the upper 4 bits of PC + 4.



Implementing Control Paths for different Instruction Types



(A) R-Type Instruction (add, sub, AND, OR, slt)

eg. add $t1, $t2, $t3

Step1: Fetch

→ Transfer contents of PC to MAR

→ Choose a particular memory location

→ Read instruction

Step2: Decode → Instruction present in IR is decoded

→ ~~Instruction~~ Increment PC by 4

Step3: Execute

→ Pick source operands' reg. no from instruction

→ Read corresponding data from source registers

→ The main control unit computes the setting of control lines

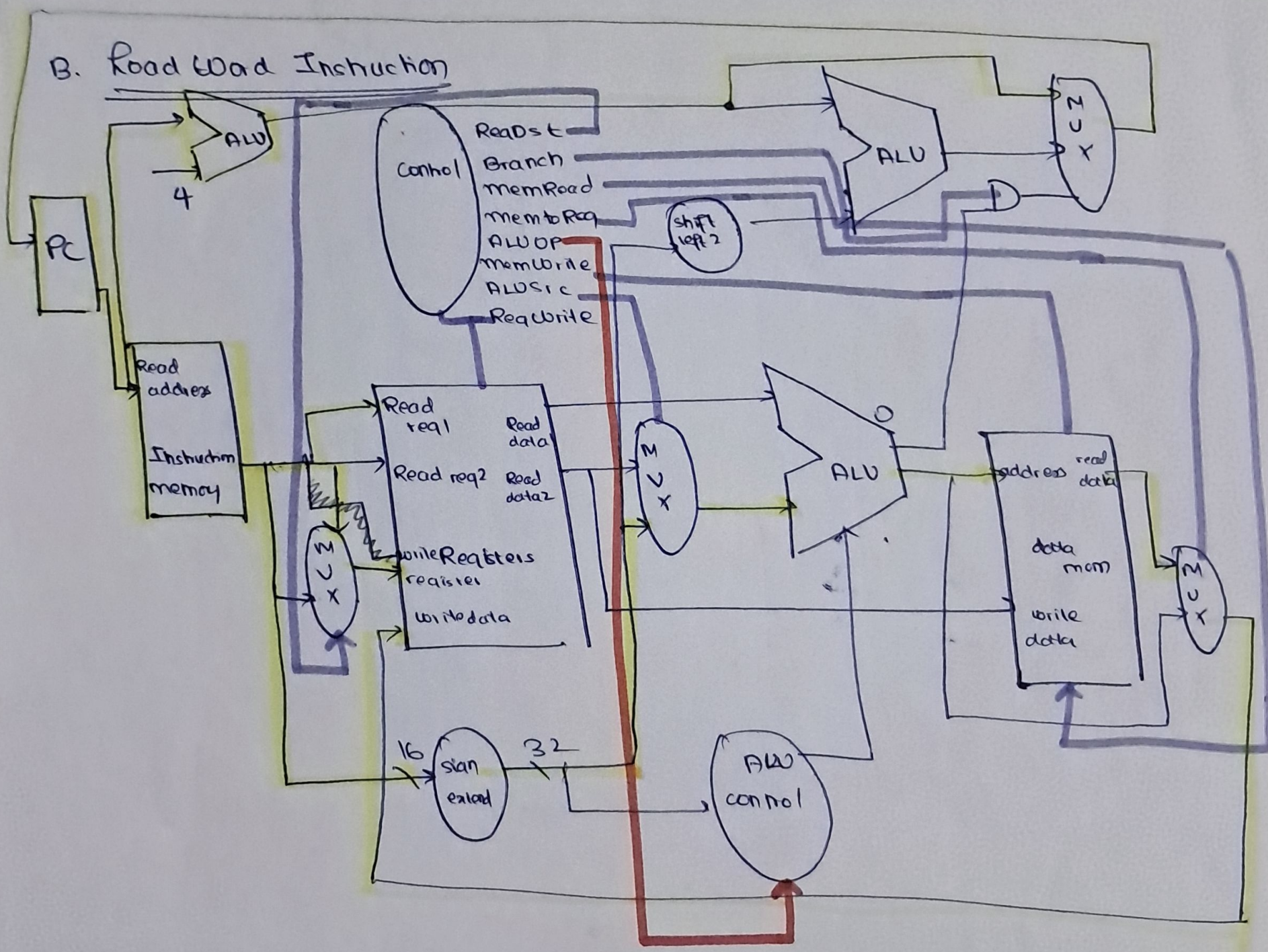
→ The ALU operates on the data read from the register file, using the funct field.

Step 4: Write back

→ The write register no. from the instructions is extracted to know which reg. to write into.

→ The result from the ALU is written into the register file (bits 15:11), after selecting the destination register.

B. Load Word Instruction



Step 1 & 2 — Write from R-type

→ A register value is read from the register file

→ The ALU computes the sum of the value read from the reg. file & the sign extended, lower 16 bits of the instruction (offset)

→ The sum from the ALU is used as the address for the data memory.

→ The data from the memory unit is written into the register file, the register destination is given by bits 20:16.

c. Branch Instructions

* Why is a single-cycle implementation not used today?

Ans. → In single-cycle design, the clock cycle must have the same length for every instruction

→ The clock cycle is determined by the longest possible path in the processor (load instruction, using 5 function units in series- the instruction memory, the register file, the ALU, the data memory and the register file)

→ The clock cycle becomes too long, performance is very poor.

Solution - use a multicycle implementation

each step in the execution will take 1 clock cycle

allows a functional unit to be used more than once per instruction

Pipelines

→ Pipelining is a process of decomposing a sequential process into sub operations, with each sub operation being executed in a separate dedicated segment that operates concurrently with all other segments

→ involves the overlapping of multiple instructions

→ lowers total execution time by increasing instruction throughput

→ It does not reduce the execution time of an individual instruction.

→ Each step in the pipeline completes a part of an instruction.. Each step is called a pipeline stage or pipeline segment.

→ In an ideal case, one instruction can finish executing on every clock cycle.

Potential speedup = no. of pipeline stages

* Pipeline Throughput = no. of instructions completed per second

* Pipeline Latency = how long does it take to execute a single instruction in the pipeline

* Pipeline Speed up

If all the stages are balanced (each stage takes the same amount of time)

Time taken to execute an instruction in pipeline = $\frac{\text{Time taken to execute an instruction sequential}}{\text{no. of stages}}$

→ If not balanced, speedup is less.

* Pipeline Performance

Let there be k segments in a pipeline.

There are n tasks to be performed

The cycle time is tp

No. of cycles to get the first output from the pipeline = k

Time needed for the first task = ktp

The remaining tasks can be performed in $(k-1)$ cycles.

$\therefore$ No. of cycles needed for n tasks $= k-1+n$

$= k+n-1$

Time needed for performing n-tasks using the pipelined processor

$= \underline{(k+n-1) * tp}$

Time taken by a non-pipelined processor

$= tn \quad \Rightarrow$ for n tasks $= \underline{ntn}$

$\therefore$ The speedup of the pipelined processor over non-pipelined processor is

$$S = \frac{ntn}{(k+n-1) * tp}$$

when n is large $k+n-1 \longrightarrow n$

$$\Rightarrow S = \frac{ntn}{ntp} = \frac{tn}{tp}$$

If we assume that the cycle time of a pipelined & non-pipelined processor is tp

$\Rightarrow tn = k * tp$ (1task's $tn_{seq} = k$ cycles $* tp_{pipeline}$)

$\Rightarrow \frac{tn}{tp} = S = k$

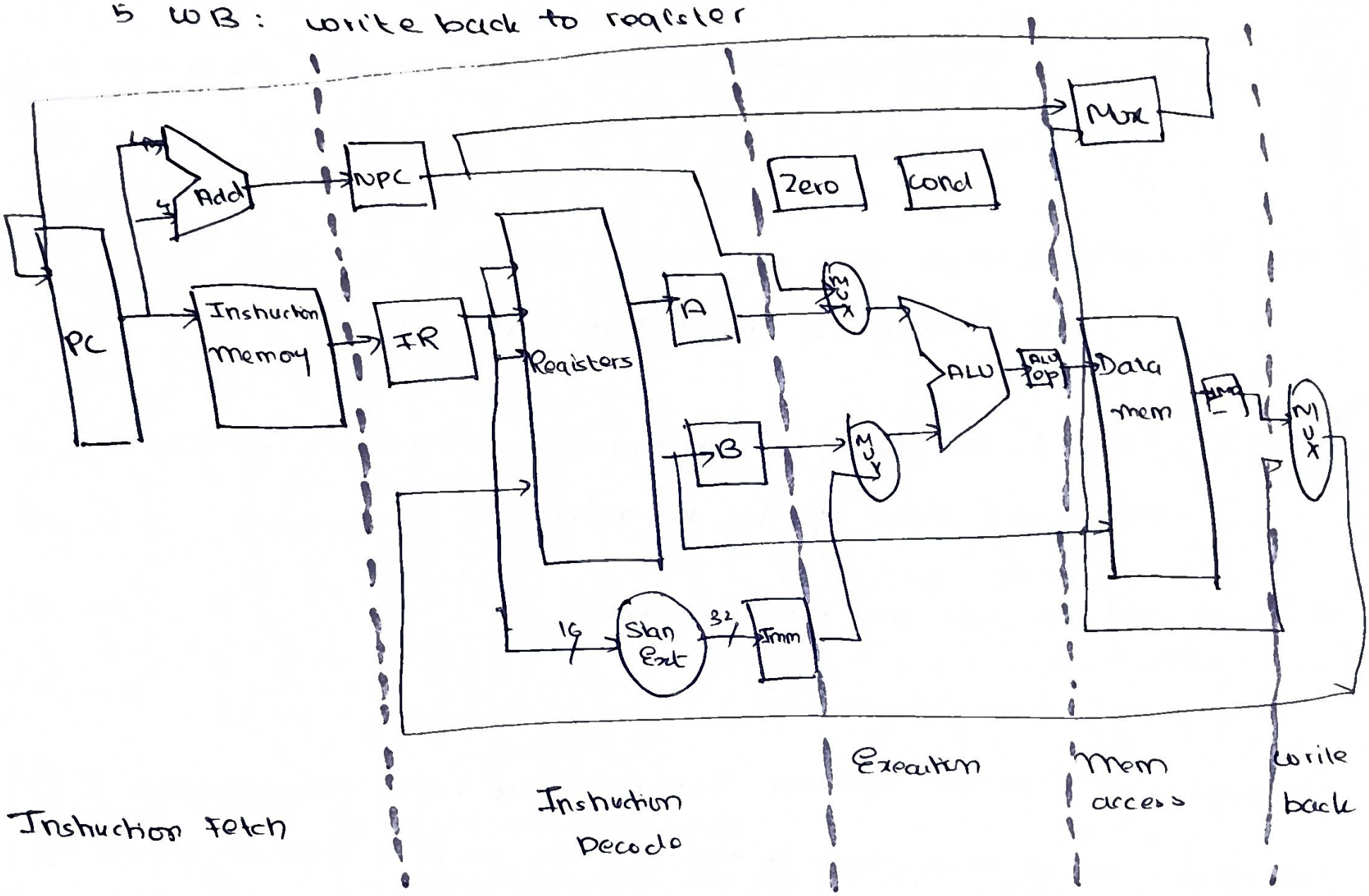
$\boxed{S = k}$

$\Rightarrow$ In ideal cases, the maximum speedup that can be provided by the pipeline = depth of the pipeline

* Pipelining in MIPS

→ There are 5 stages, one step per stage

1. IF : Instruction fetch from memory
2. ID : Instruction decode
3. EX : Execute operation
4. MEM : access memory operand
5. WB : write back to register



* Pipeline Hazards

→ Situations that prevent the next instruction in the instruction stream from being executed during its designated clock cycle.

→ reduce performance from the ideal speedup gained during pipelining

* Types of Hazards

(i) Structural Hazards

(ii) Data Hazards

(iii) Control Hazards

Structural Hazards → arise from resource conflicts

→ H/W cannot support all possible combinations of instructions

Data Hazards → occur when instruction depends on data from an instruction ahead of it in pipeline

Control Hazards → results from branches / other instructions that change the flow of the program

* Dealing w/ Hazards and the Consequences

→ pipeline must be stalled

→ stalling pipeline usually temporarily halts all instructions / lets some other instructions in the pipeline proceed, while the stalled inst waits for data, resources etc.

Consequences

→ stalls impede progress of pipeline & result in a deviation from 1 instruction executing / clock cycle.

CPI pipelined =

= ideal CPI + pipeline stall cycles per instruction

= 1 + pipeline stall cycles per instruction

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{pipeline stall cycles per instruction}}$$

(1) Structural Hazards

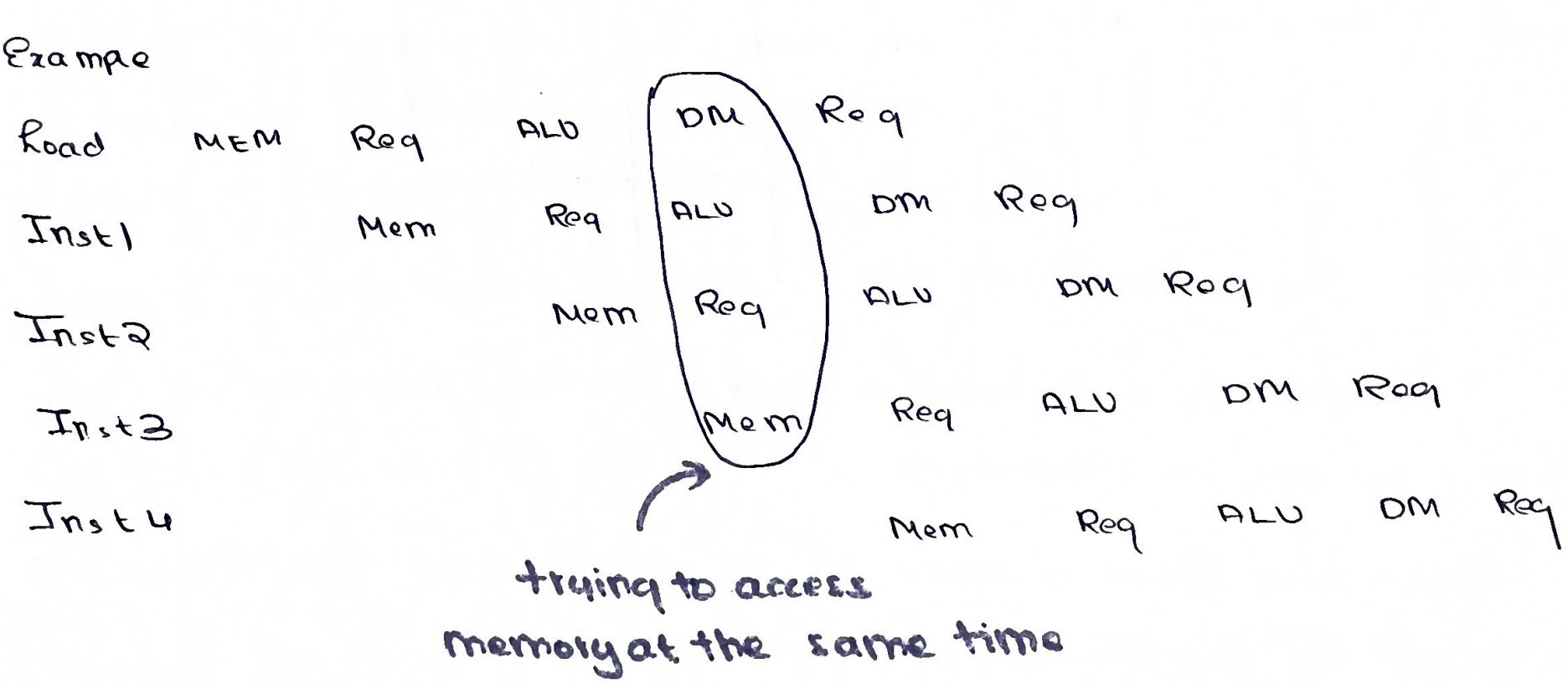
→ when not all possible combinations of instructions can be executed, structural hazards occur

→ Solution (i) duplicate resources

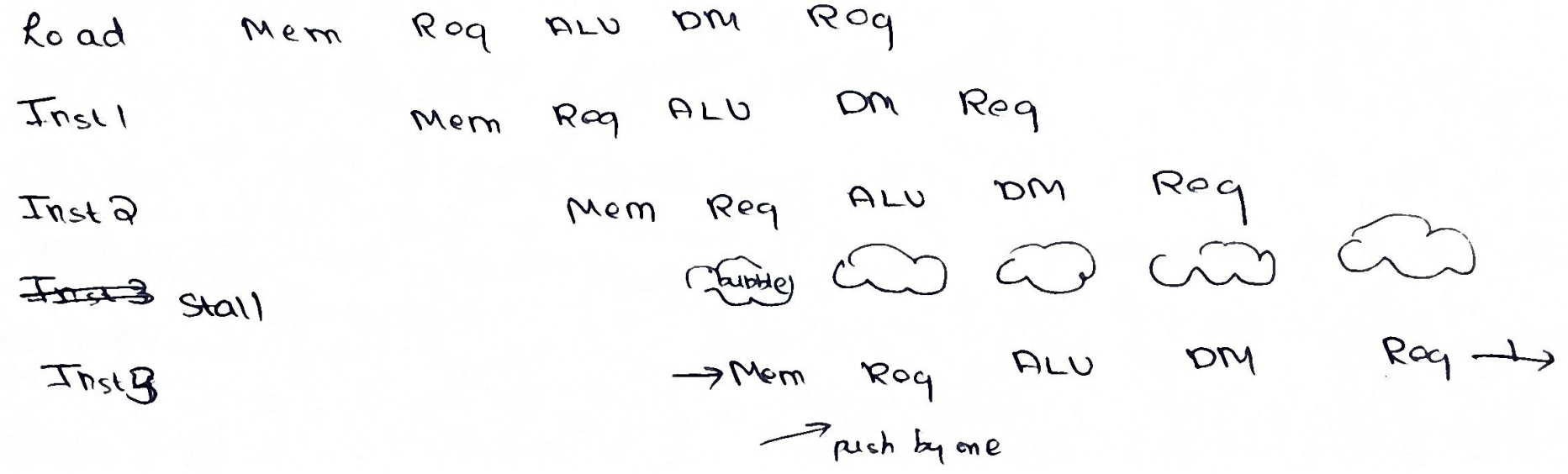
eg. an ALU to perform an arithmetic execution and an adder to increment the PC

(ii) introduce stalls / bubbles

Example



Solution: introduce a stall



* LOAD instruction steals an instruction fetch cycle, causing stall.

(2) Data Hazards

→ occurs when an instruction depends on data from an instruction ahead of it in the pipeline

Example

```
ADD  R1 , R2  R3
SUB  R4 , R1 , R5
AND  R6 , R1 , R7
OR   R8 , R1 , R9
XOR  R10 , R1 , R11
```

All instructions after ADD use the result of ADD

ADD writes into the register in the WB stage, but SUB needs it in the ID stage

⇒ data hazard

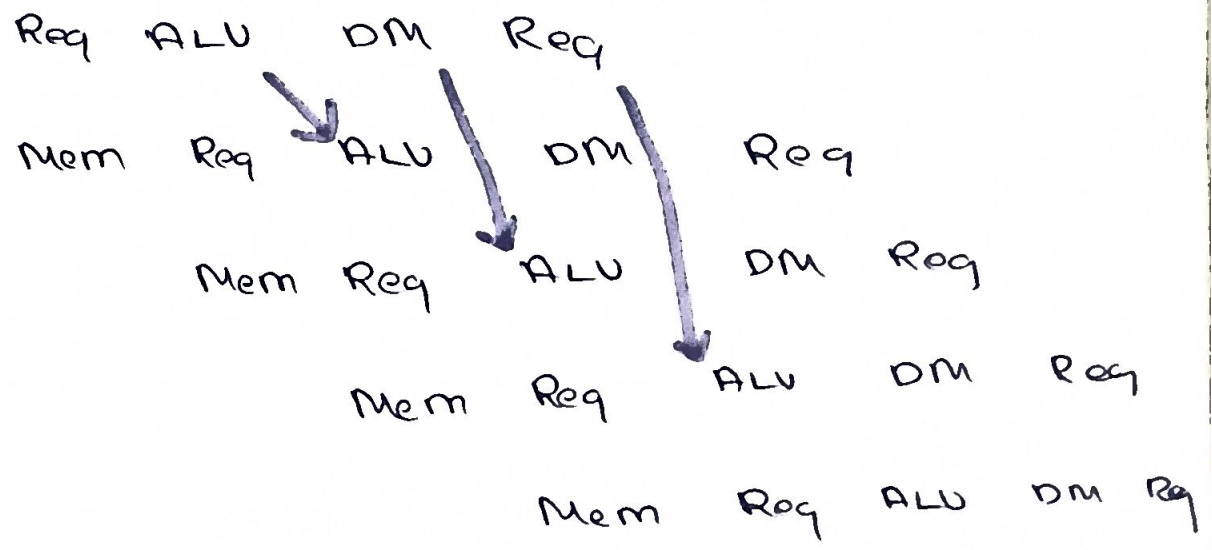
Solution: Forwarding

→ Forwarding occurs when a result is passed directly to a functional unit that requires it

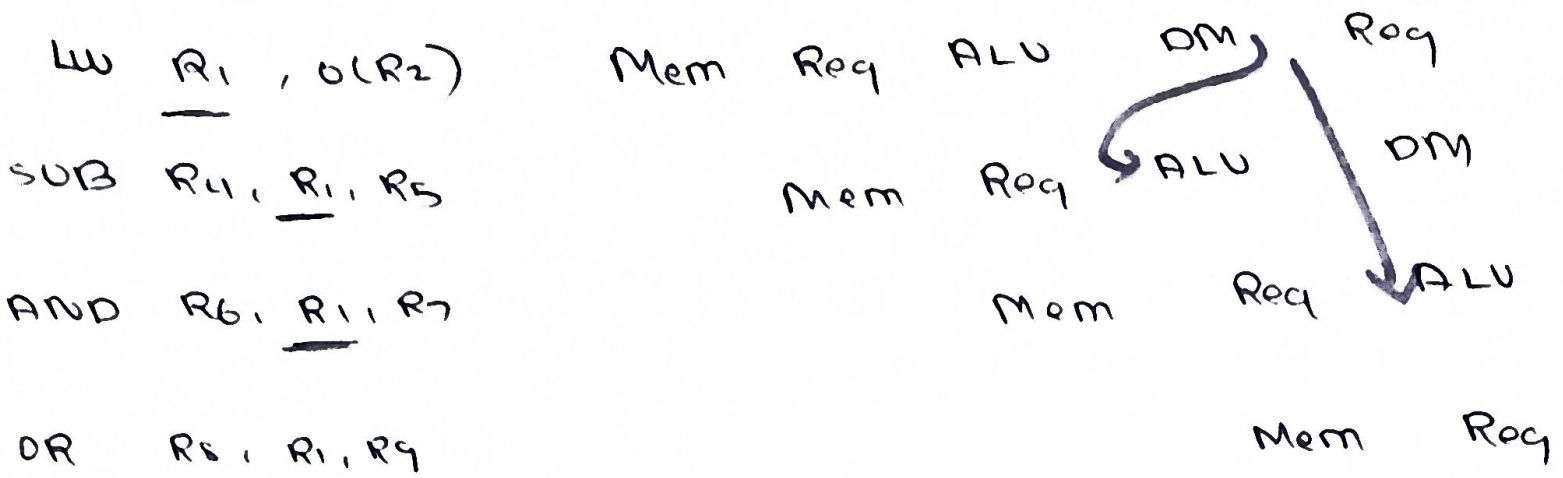
→ Result goes from the output of one unit to input of another

w/ forwarding, the example above becomes:

```
ADD  R1 , R2 , R3
SUB  R4 , R1 , R5
AND  R6 , R1 , R7
OR   R8 , R1 , R9
XOR  R10 , R1 , R11
```



Example (where forwarding doesn't work)



Here, can't get data to the sub instruction, because the result is needed at the beginning of clock cycle 4, but it is not produced until the end of clock cycle 4.

→ use a stall instead

* Types of Data Hazards

① RAW (Read after Write)

Instruction J tries to read data before instruction I writes it

I: $R_2 \leftarrow R_1 + R_3$

$R_4 \leftarrow R_2 + R_3$

② Write after Read (WAR)

Instruction J tries to write data before instruction I reads it.

I: $R_2 \leftarrow R_1 + R_3$

J: $R_3 \leftarrow R_4 + R_5$

③ Write after Write (WAW)

Instruction J tries to write output before instruction I writes it

I: $R_2 \leftarrow R_1 + R_3$

J: $R_2 \leftarrow R_4 + R_5$

Examples

(1)
```
LW  R1 , 45(R2)
ADD R5 , R1 , R7
SUB R8 , R6 , R7
OR  R9 , R6 , R7
```

=> requires 1 stall because of the use just before add

(2)
```
LW  R1 , 45(R2)
ADD R5 , R6 , R7
SUB R8 , R1 , R7
OR  R9 , R6 , R7
```

=> can use forwarding, result of LOAD can be forwarded to ALU before sub instruction

(3)
```
LW  R1 , 45(R2)
ADD R5 , R6 , R7
SUB R8 , R6 , R7
OR  R9 , R1 , R7
```

Ans

```
LW  R1 , 45(R2)
ADD R5 , R6 , R7
SUB R8 , R6 , R7
OR  R9 , R1 , R7
```

```
Mem  Reg  ALU  DM  Reg
     Mem  Reg  ALU  DM  Reg
          Mem  Reg  ALU  DM  Reg
               Mem  Reg  ALU  DM  Reg
```

no action required

read from Reg occurs in the second half while of R1 happens in the first half.

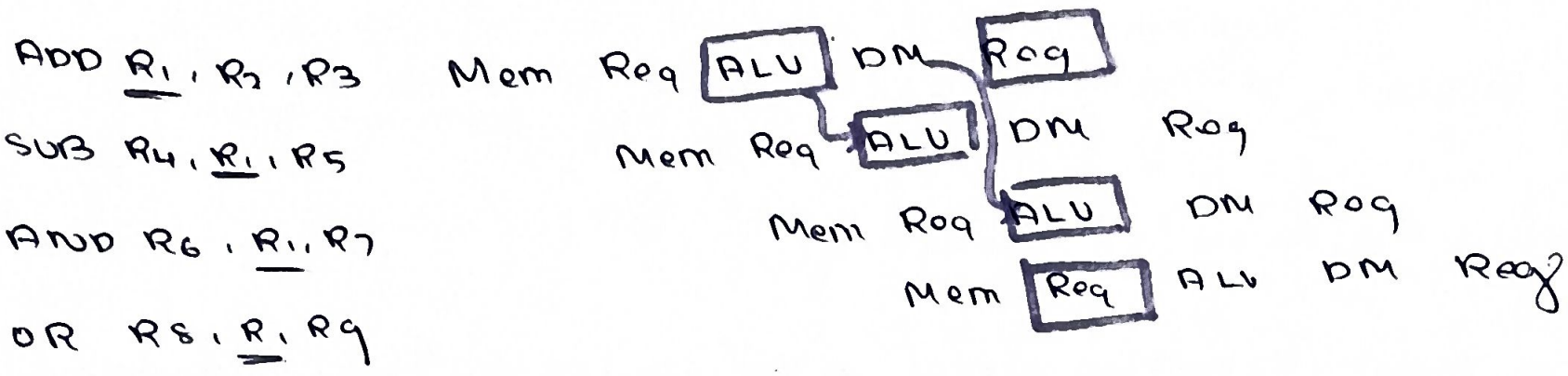
Note: Register writes happen in the first half of the clock cycle, read happens in the second half (can access memory twice in 1 cc)

(4) ADD R_1, R_2, R_3

SUB R_4, R_1, R_5

AND R_6, R_1, R_3

OR R_8, R_1, R_9



The req write of R_1 happens first in cc 5 and the req read of R_1 for the OR operation happens in the second half of the 5th cc.

(3) Control Hazards

→ Control hazards arise from the need to make a decision based on the results of one instruction, while others are executing (occur less frequently than data hazards)

Schemes to Resolve Control Hazards

A. Assume Branch not Taken

→ stalling until the branch is complete is too slow

→ An improvement over branch stalling is to predict that the branch will not be taken, and thus continue execution down the sequential instruction stream.

→ If the branch is taken, the instructions that are being fetched and decoded must be discarded.

→ If the branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

→ To discard instructions, change the original control values to 0.

B. Reducing the delay of Branches

→ Move branch execution earlier in the pipeline ⇒ fewer instructions need to be flushed.

* Dynamic Branch Prediction

→ Look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, begin fetching instructions from the same place as last time. This technique is called dynamic branch prediction.

Implementation

1-bit Branch prediction

→ use a branch prediction buffer or branch history table.

→ A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction

→ The memory contains a bit that says whether the branch was recently taken or not.

Shortcoming: Even if a branch is almost always taken, we predict incorrectly twice, rather than once.

eg. for . . . .

```
for (i = 0 ; i < a ; i++)
    a[i] = a[i] * 2;
```

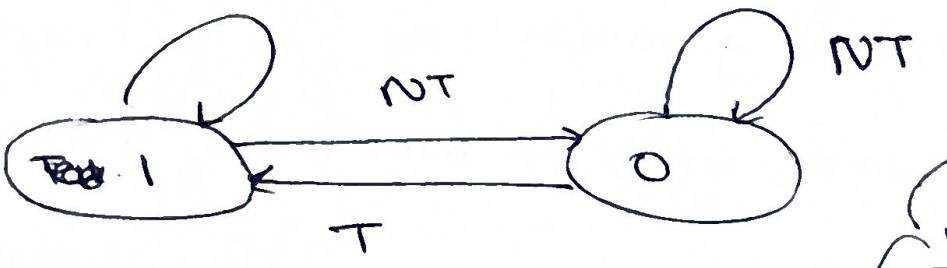
Mispredictions: first & last loop iterations

(i) last iteration - prediction bit will indicate taken, as the branch has been taken 9 times in a row.

(ii) first iteration: bit was flipped on prior execution of the last execution, since the branch was not taken on that exiting iteration.

accuracy: branch taken 90% of the time

accuracy = 80%. (2 incorrect predictions + 8 correct ones)



1-bit branch prediction

another method:
branch delay slot after delayed branch inst, is filled by an instruction that does not affect the branch.

* Exceptions

→ exceptions: unexpected events requiring change in flow of control

exception vs. interrupt

↓ ↓

arises within CPU (undefined opcode, overflow, syscall)

from an external I/O controller

* Handling Exceptions

→ In MIPS, exceptions are managed by a System Control Coprocessor

→ Save the program counter of the interrupted instruction
(In MIPS - Exception Program Counter (EPC))

→ Save indication of the problem

(i) the cause register

(ii) assume 1 bit: 0 → undefined opcode

1 → overflow

→ Jump to exception handler

Method 2: use vectored interrupts

→ the address to which control is transferred is determined by the cause of the exception

→ instructions deal with the interrupt / jump to real handler

~~* Errors~~

What does the handler do?

→ determines action to be taken

If restartable:

(i) take corrective action

(ii) use EPC to return to program

Otherwise

(i) terminate program

(ii) report error using EPC and cause

* Exceptions in a Pipeline

→ another form of a control hazard

eg. consider overflow on add in EX stage

add $1 , $2 , $1

Soln:

complete all previous instructions

flush add & subsequent instructions

set cause and EPC register values

transfer control to handler

* Exception Properties

① Restartable exceptions

→ flush inst. from pipeline

→ execute handler, return to inst, refetched & executed from scratch

② PC saved in EPC register

→ identifies causing instruction

→ handler must adjust as PC +4 is what gets saved

* Precise vs. Imprecise Interrupt

↓ always associated w/ correct instruction

↓ not associated w/ exact instruction that was the cause of the interrupt or exception

If there are multiple exceptions in a pipeline, due to overlap, deal w/ exception from the earliest instruction, flush subsequent instructions

Unit 3

Additional Numericals

(1) Consider the following code:

```
DIV   R2   R5   R8
SUB   R9   R2   R7
ADD   R5   R14  R6
MUL   R11  R9   R5
BEQ   R10  #10  R12
OR    R8   R15  R2
```

Identify all the RAW, WAR, WAW & Control Hazards.

Ans RAW

(i) DIV & SUB
(ii) SUB & MUL
(iii) DIV & OR
(iv) SUB & MUL

WAR

(i) DIV & ADD
(ii) DIV & OR

WAW → NONE

CONTROL → BEQ, OR

(2) An unpipelined system tasks 50ns to process a task. The same task can be processed in a 6 segment pipeline with a clock cycle of 10ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

$t_n = 50ns$

$t_p = 10ns$

$k = 6$

$n = 100$

$$S = \frac{n t_n}{(k+n-1) * t_p} = \frac{100 * 50}{(6+100-1) * 10}$$

$$= \frac{500}{105} = 4.76$$

$$\boxed{S = 4.76}$$

$$\text{max speedup} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

(3) Consider an unpipelined processor that has a 1ns clock cycle and it uses 4 cycles for ALU and branch operations and 5 cycles for memory operations. Assume the relative frequency of these operations are 40%, 20% & 40% respectively. Suppose that due to clock skew & setup, pipelining the processor adds 0.2ns of overhead to the clock, ignoring any latency impact, how much speedup in the instruction execution rate will we gain from the pipeline?

$T = CPI \times C \times I$

$= Ins * ((0.40 + 0.2) * 4 + 5 * 0.4)$

$= Ins * (4.4) = 4.4ns$

Pipelined Instructions:

$1 + 0.2 = 1.2$ ns

Speedup = $\frac{4.4}{1.0}$ = 3.76 times

(4) The time delay for the 4 segments in a pipeline are as follows: $t_1 = 50$ns, $t_2 = 30$ns, $t_3 = 95$, $t_4 = 45$ns. The interface register delay time $t_r = 5$ns

(a) How long would it take to add 100 pairs of numbers in the pipeline?

(b) How can we reduce the time to about one half of the time calculated in (a)?

Answer time taken = 100ns

pipelined time $= (k + n - 1) * t_p$

$t_p = 100$

$k = 4$

$n = 100$

$= (4 + 99) * 100 = 10,300$ ns

(b) Divide max segment into 2

50 + 5 = 55 ns

45 + 5 = 50 ns

⇒ new $t_p = 55$

$(k + n - 1) * t_p = (5 + 99) * 55 = 5720$ ns

(5) Assume that the individual stages of the datapath have the following latencies

IF	ID	EX	MEM	WB	
250	400	150	350	200	(in ps)

What is the clock cycle time in a pipelined & non-pipelined processor?

Ans

non-pipelined = 1350 ps

pipelined = 400 ps

(6) Assume that instructions executed by the processor are broken down as follows.

ALU	beq	lw	sw
40%	20%	25%	15%

What is the utilization of the write-register port of the Registers unit?

Ans>

ALU & lw use the write register port

=> 65%.

(7) Assume that the individual stages of the datapath have the following latencies

IF	ID	EX	MEM	WB
250ps	400ps	150ps	350ps	200ps

Split one stage into 2, and identify the new clock cycle time?

Ans. Split ID = 400 ps

New cycle time = 350ps

***

(8) Assume the following fragment of MIPS code:

```
lw $s0, 20 ($t1)
sub $t2, $s0, $t3
```

The individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps.

Identify the total latency (assuming there is no forwarding unit)

(9) What is the longest chain of dependent operations in the following program?

```
LD   r7, r8
SUB  r10, r11, r12
MUL  r13, r7, r11
ST   r9  r13
ADD  r13 r2, r1
LD   r5, r6
SUB  r3, r4, r5
```

Ans.

```
LD  r7 r8
MUL r13 r7 r11
ST  r9 r13
ADD r13 r2 r1
```

Role of control signals

(i) Branch → used to make with PcSrc = Branch + 0 from ALU

(ii) ALUSrc → selects second source operand

(iii) ALUOp → specifies what operation should be performed

(iv) MemRead, MemWrite → enables memory read, write & load instructions

(v) RegWrite → enables a write to one of the registers

(vi) RegDst → determines how the destination register is specified

(vii) MemtoReg → determines where the value to be written comes from.