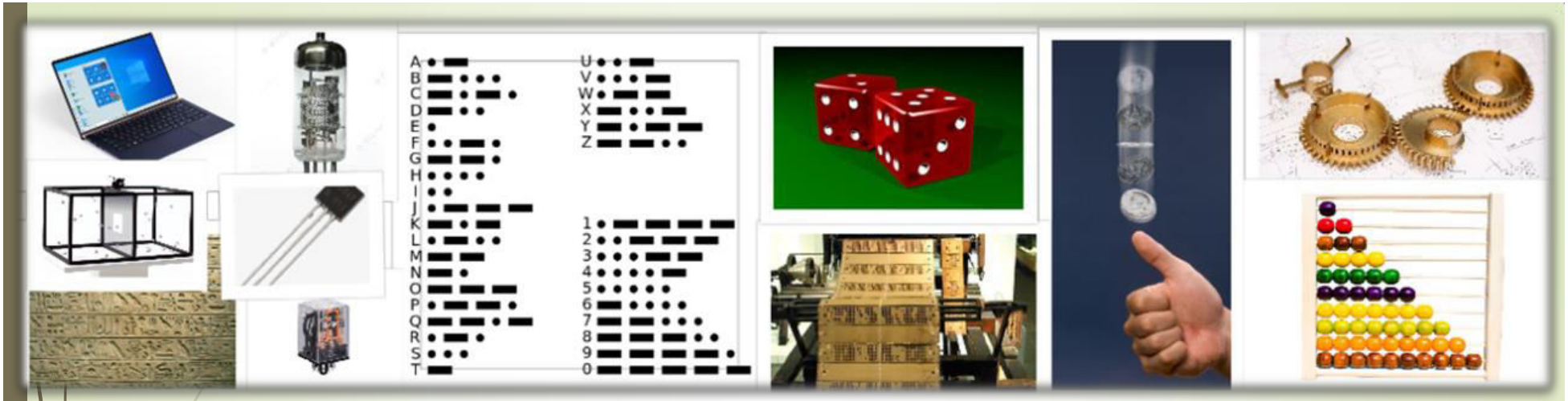


Computer Architecture (CS F342)

Design and Analysis of Instructions

Design of Control Unit for Reduced Instruction Set Computer (RISC)

Performance improvement of CPU

- To improve the performance of any processor by analyzing
  - Not only the Control Unit
  - But also the data-path
  - How does one analyze that?

How does one analyse the data-path?

- Consider the different stages of instruction cycle
- Identify the components

How does one analyse the data-path?

- Consider a few of the MIPS [*] instructions and design the data path
- Data instruction of type register or R-type instruction
- Memory type or M-type instruction or immediate type or I-type instruction
- Branch type or B-type

[*] Microprocessor without Interlocked Pipeline Stages (MIPS) and RISC-I to RISC-V are the example of RISC-style processor

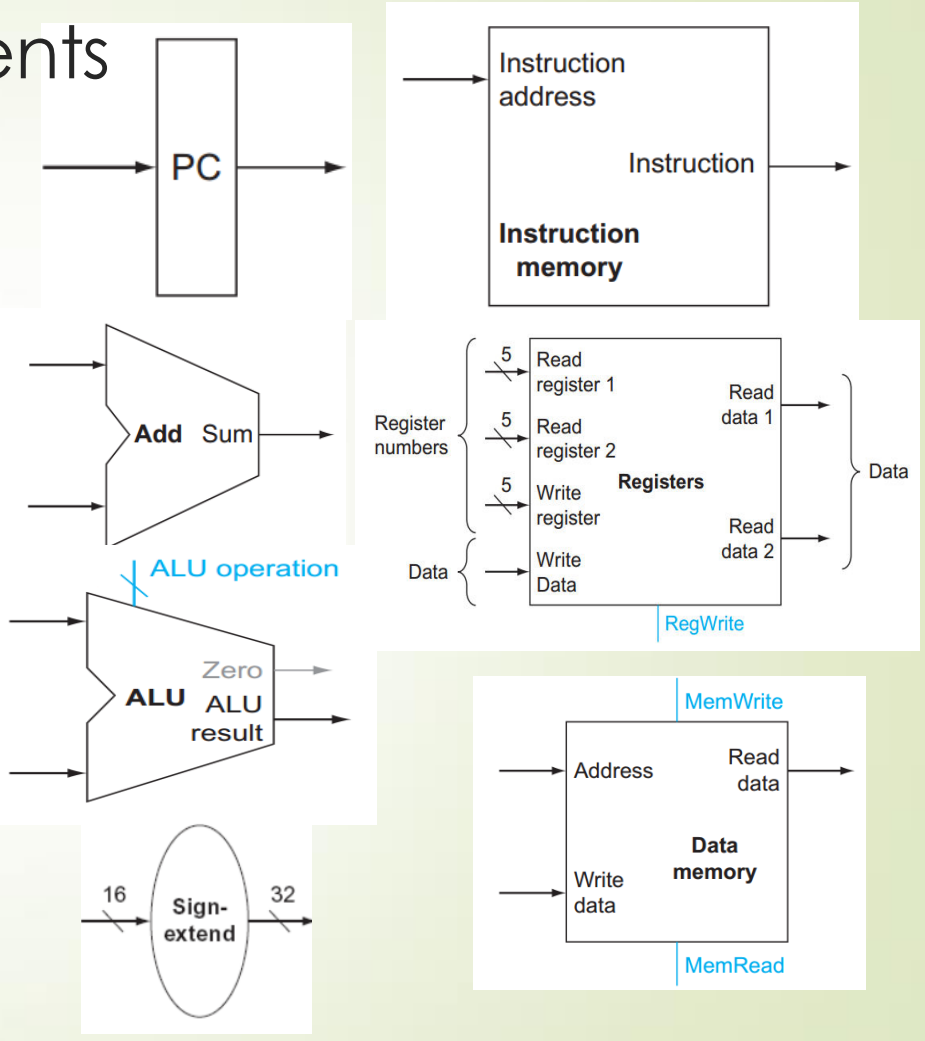
Instruction format for MIPS-based processor

op	rs	rt	rd	shamt	funct
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)

- *op:* Basic operation of the instruction, called the opcode
- *rs:* The first register source operand
- *rt:* The second register source operand
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount. (to be used for shift instructions. Otherwise, the field contains zero in this section.)
- *funct:* Function. This field, often called the *function code,* selects the specific variant of the operation in the op field

Data-path elements

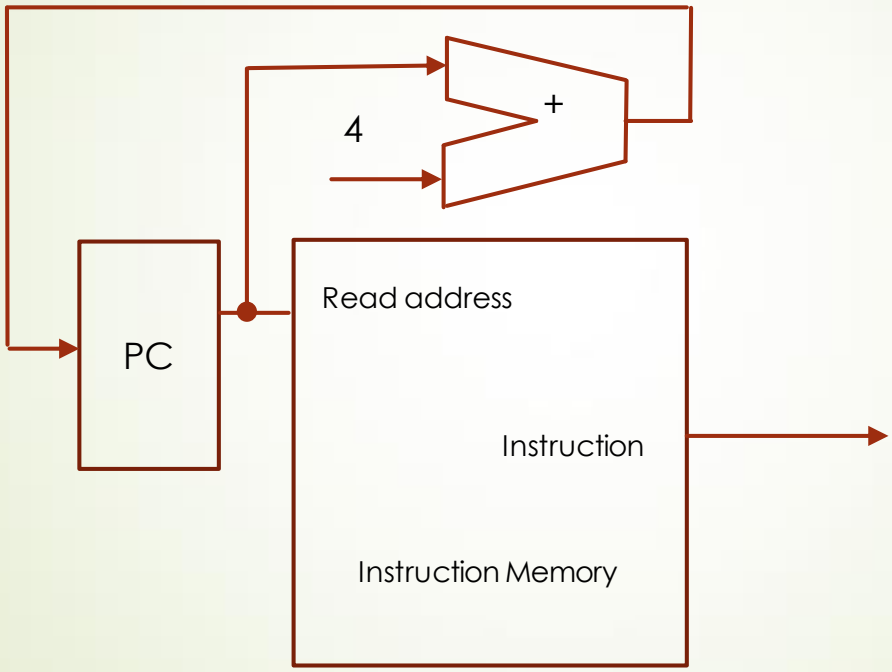
- Program counter (PC)
- Instruction memory
- Adder
- Register file
- ALU
- Signed-extension unit
- Data memory



Program Counter stores the address to the instruction location.
Instruction memory reads this address and stores the instruction.
After every instruction, **+4** is added to PC to get to the next instruction.

6

Analysis of data-path for Fetch stage



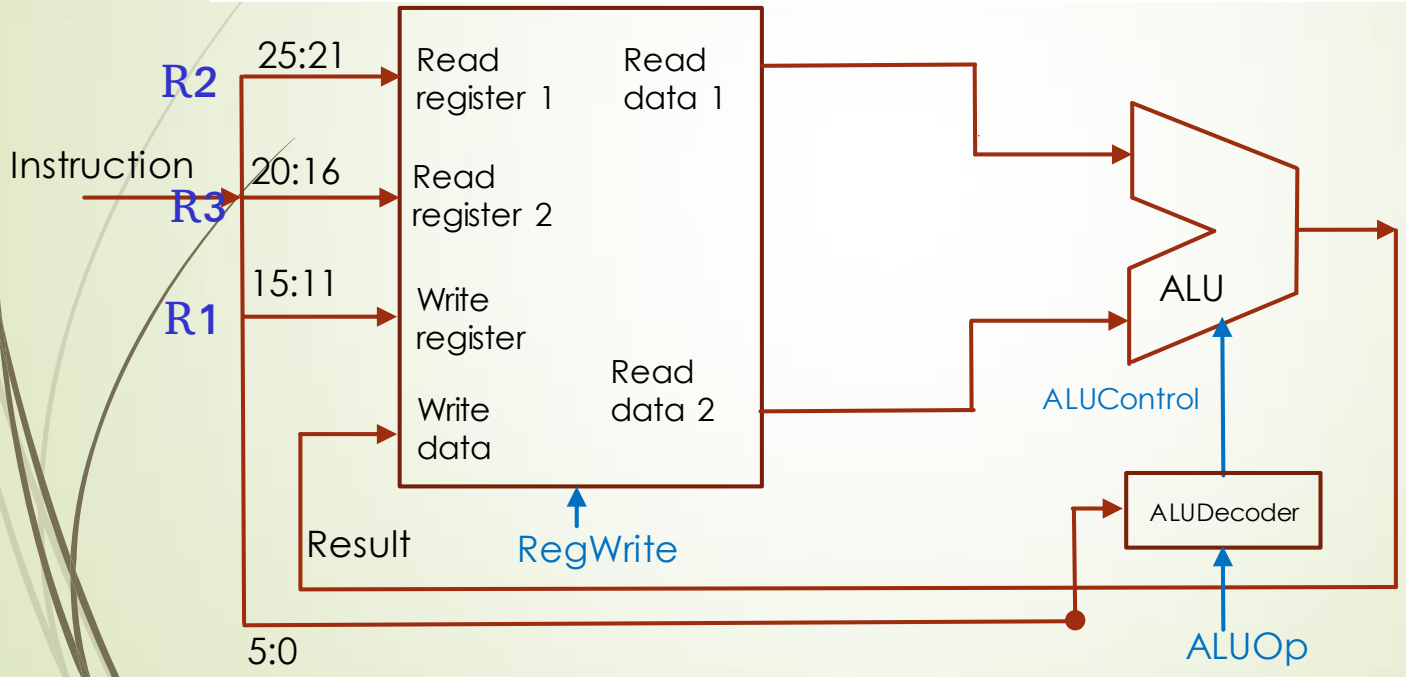
rs, rt, rd will store the register number (assigned to each register)

7

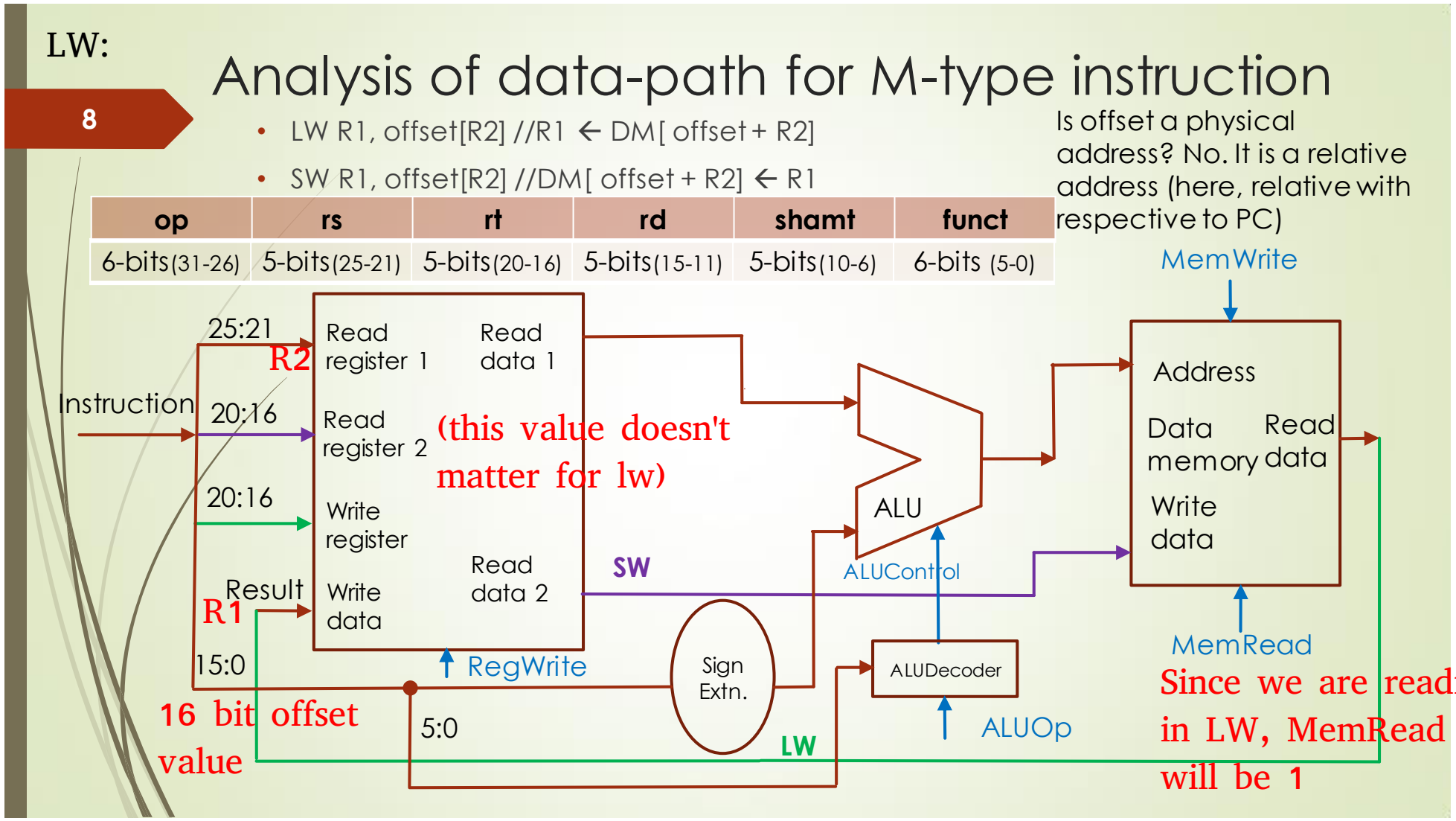
Analysis of data-path for R-type instruction

- ADD R1, R2, R3

op	rs	rt	rd	shamt	funct
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)



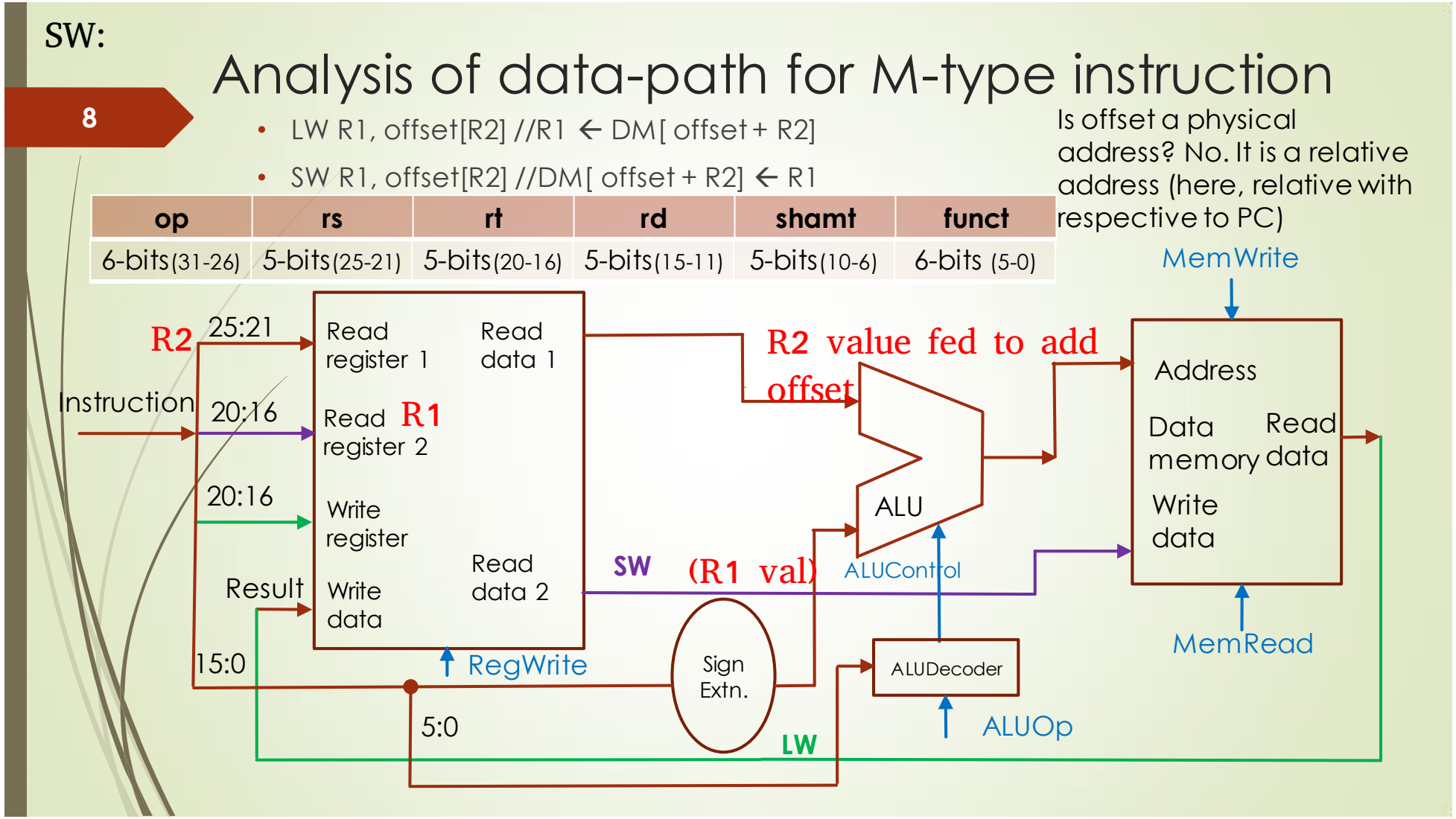
Load Word copies data from memory locn to register. So, it only has to write to the register R1 and reads only the value in R2 which has the base address stored in it.



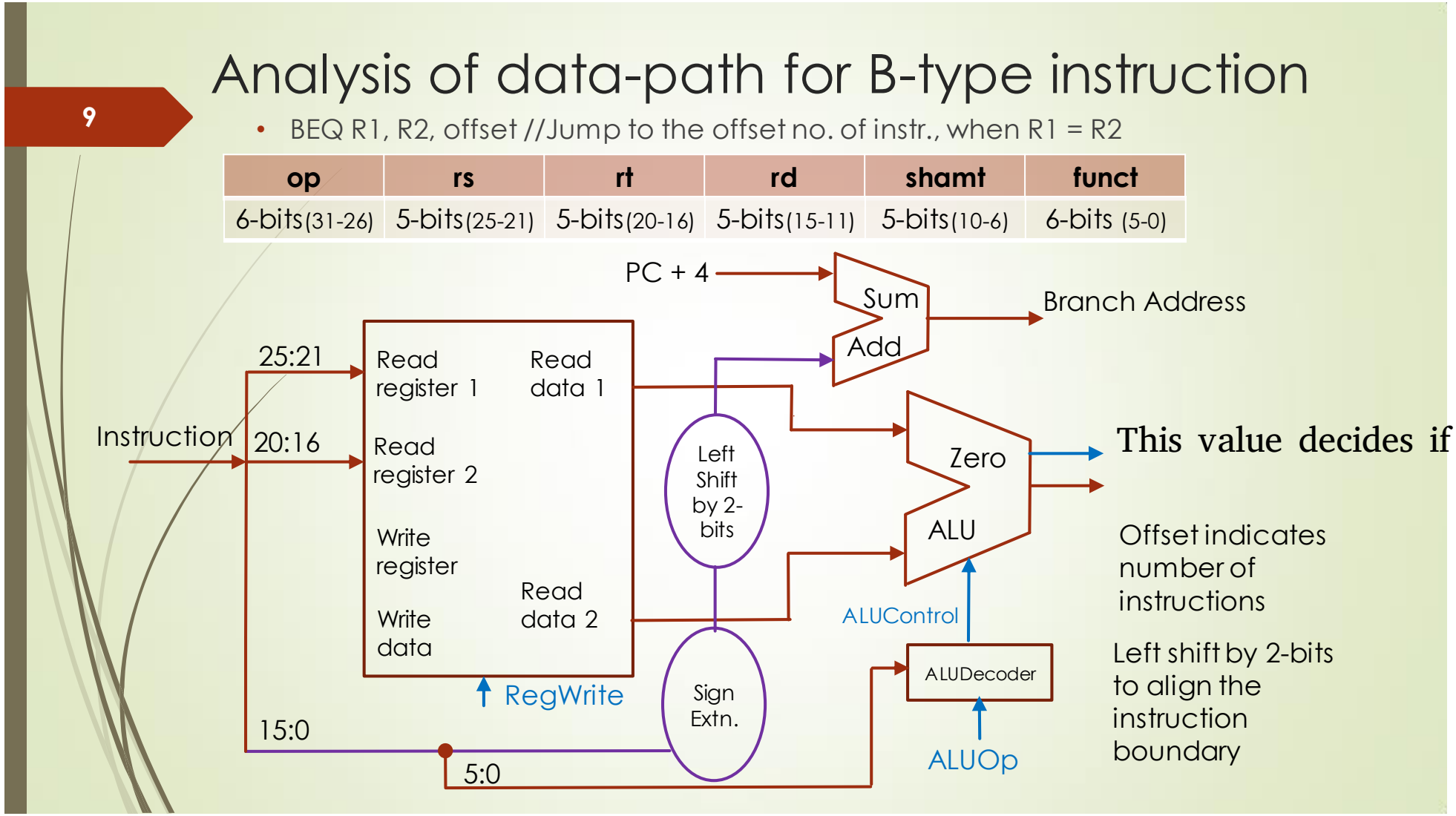
16 bit offset values has to be converted to 32 bit so sign extended.

Also, the last 6 bits which denote the funct doesn't matter here because ALUOp will only generate for Addition of offset to value in R2

SW will copy the register value in R1 to a memory location at addresss -> [offset + R2]



Conditional jump. Reads the value R1 and R2. Offset value is read (basically the label used in code). This address is added to the program counter value to get the new address of the branch.

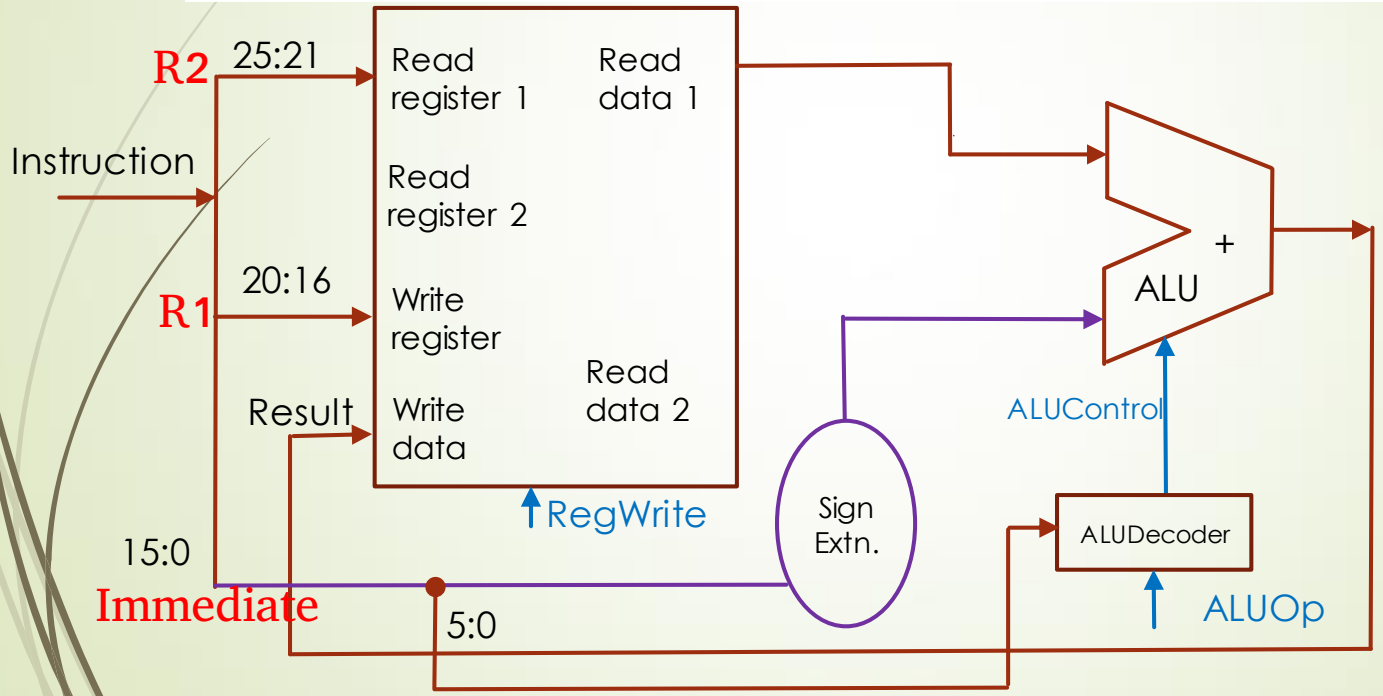


Note that here, 20:16 gives the register to be written rather than read.

Analysis of data-path I-type instruction

- ADDI R1, R2, -12 //R1 ← R2 + (-12)

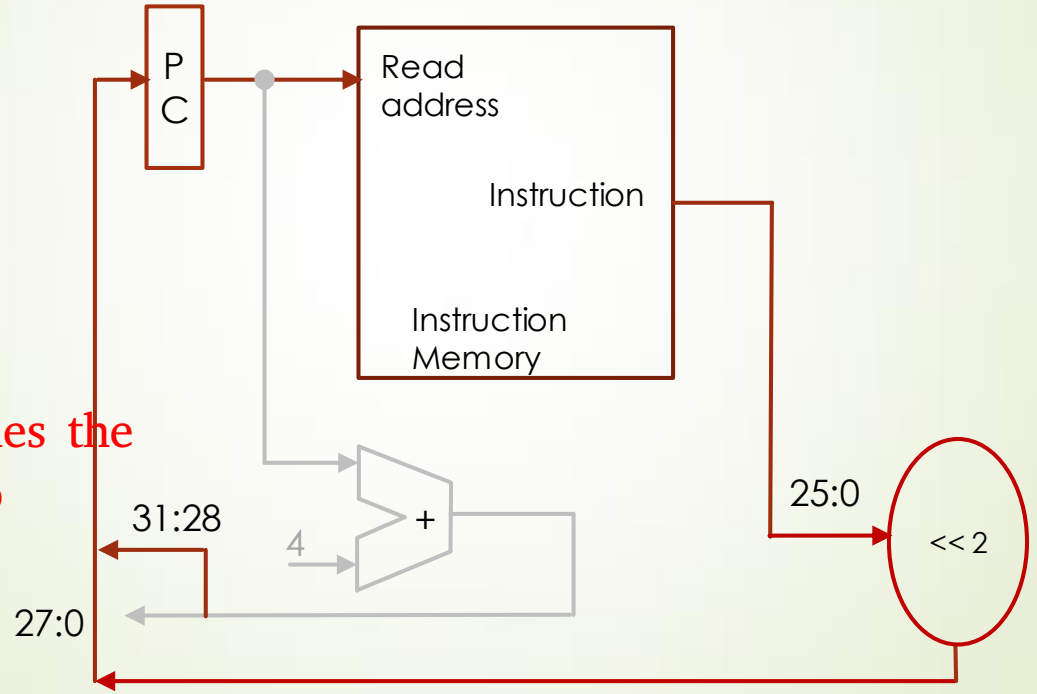
op	rs	rt	rd	shamt	funct
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)



Analysis of data-path j-type instruction

- J addrs //PC ← PC[31:28]addrs[27:0]

op	rs	rt	rd	shamt	funct
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)

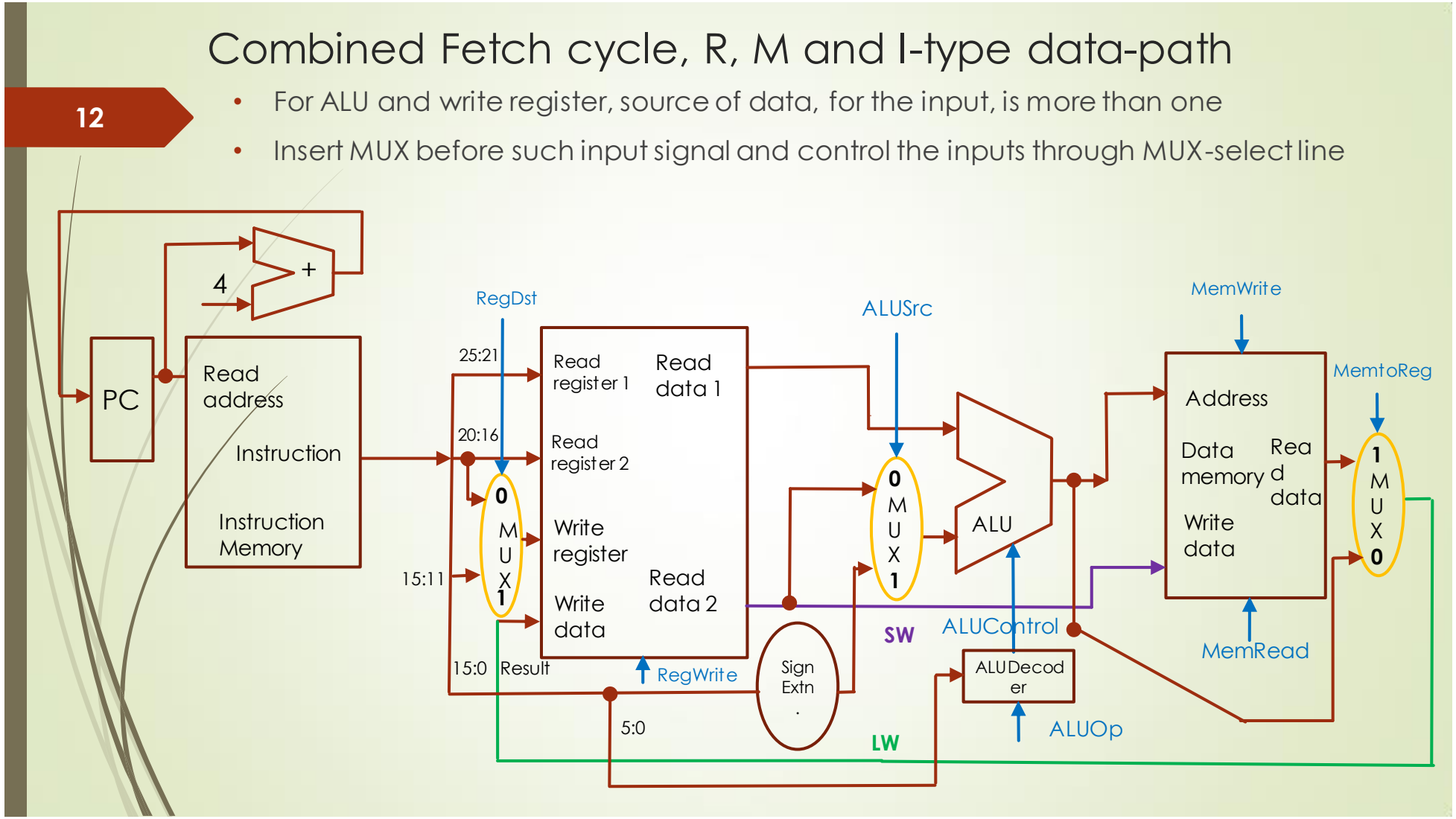


27:0 decides the actual jmp address.

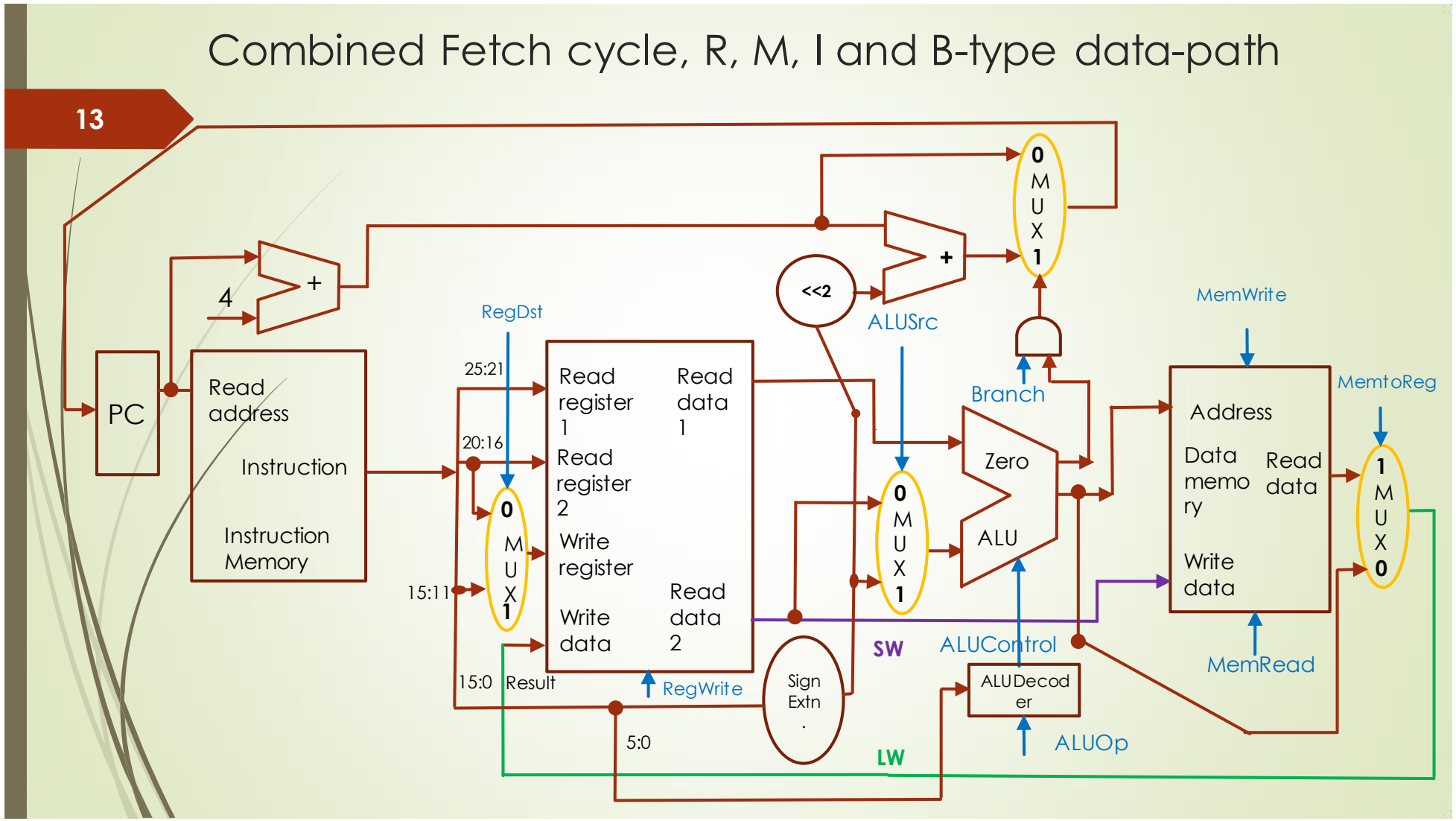
Making it multiple of 4 so that it is byte addressable

The 4 most significant bits are from the previous address value

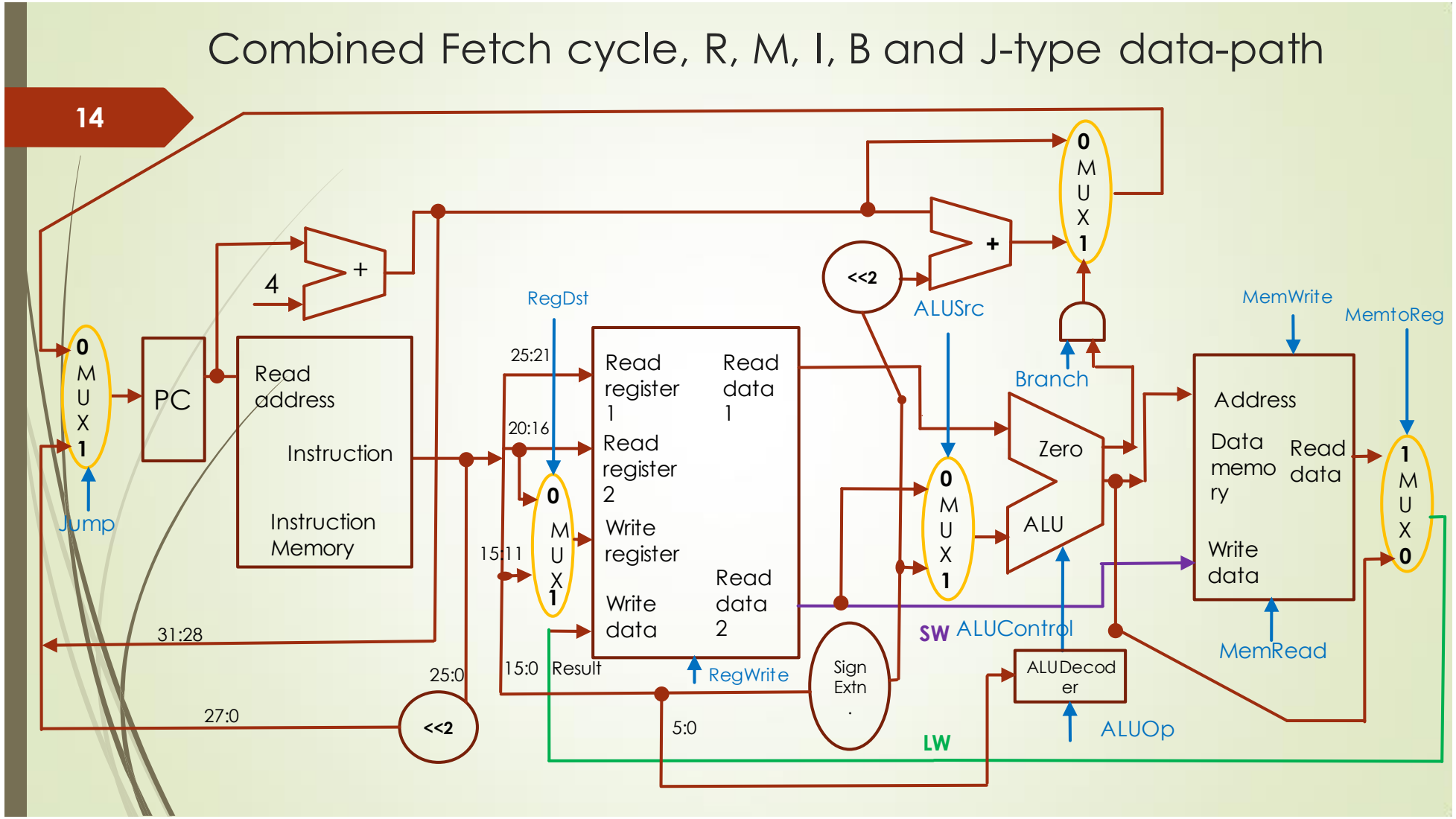
RegDst signal decides whether register **20:16** is being written or **15:11** is being written
ALUSrc decides whether Immediate value should be used or the read data **2** value.
MemToReg is set if register is being written



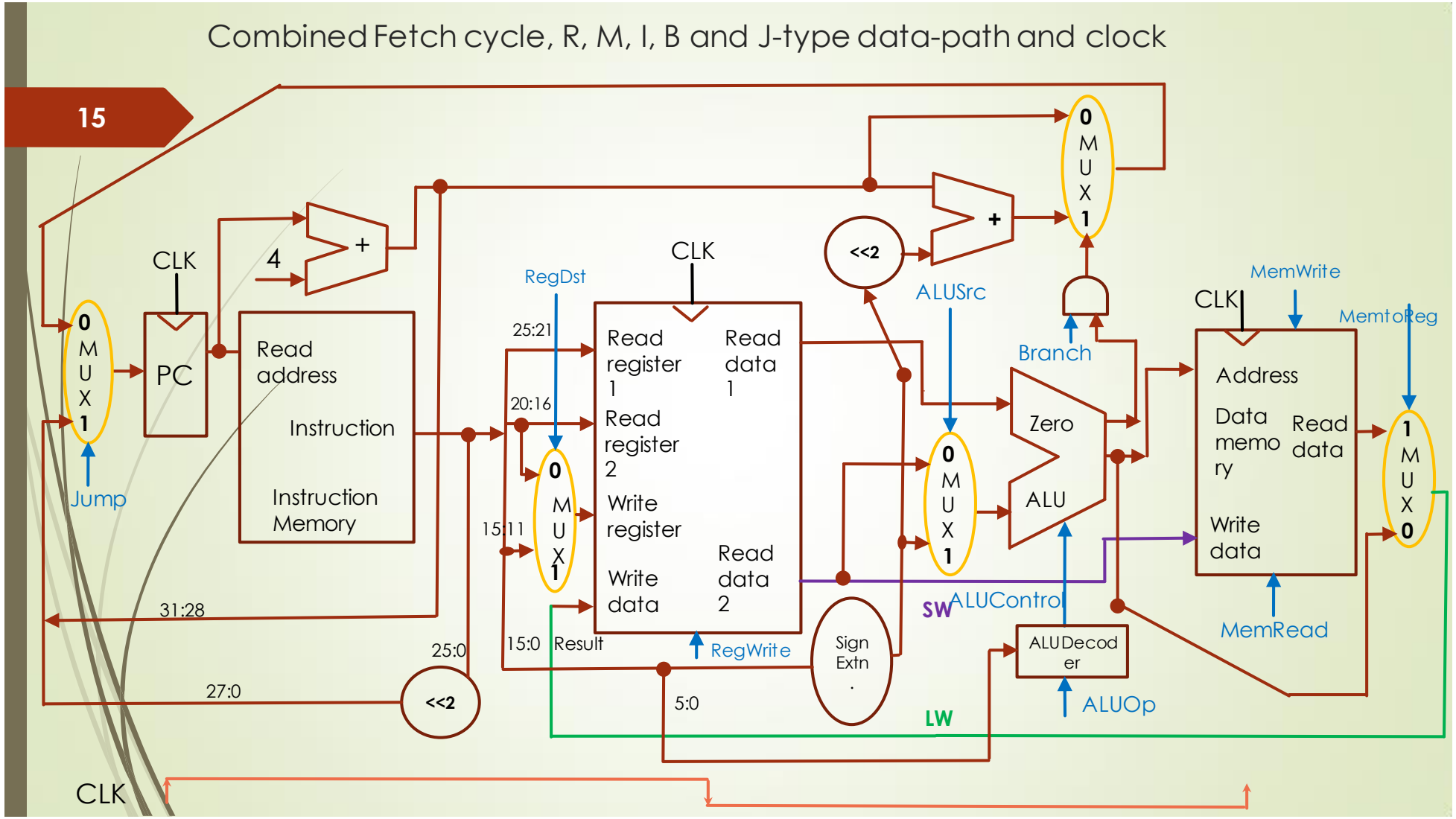
The branch signal decides whether the next address is PC+4 or the value PC+4+offset.
The zero value from the ALU must also be set (that is the comparator should satisfy the branch condition)



Jump signal decides if PC **+4** or the jump address.



Combined Fetch cycle, R, M, I, B and J-type data-path and clock



Identify the control signals

- Jump
- RegDst
- RegWrite
- ALUSrc
- Branch
- ALUOp
- MemRead
- MemWrite
- MemtoReg

Generation of Controls

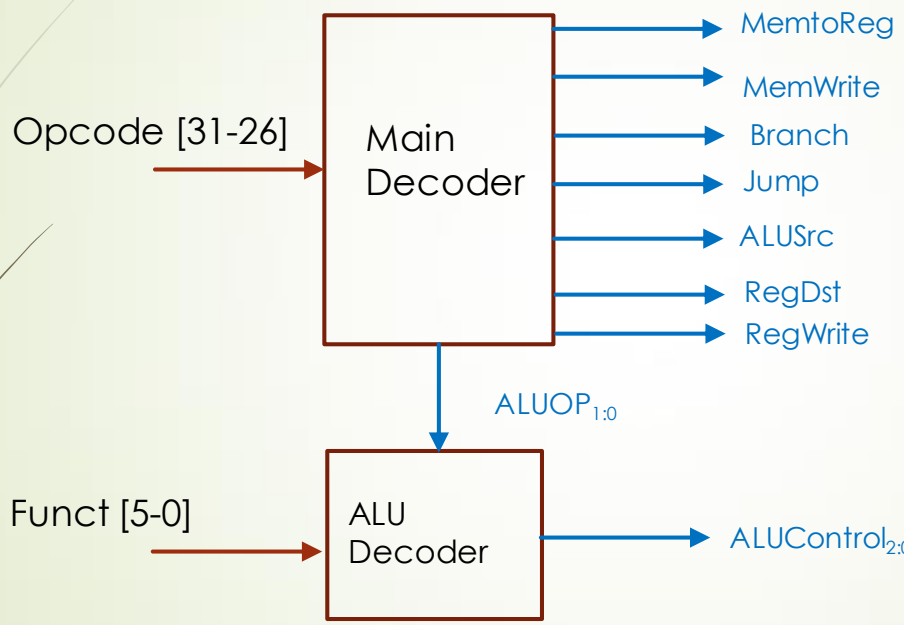
Inputs to the control unit: op-code part [31:26] and funct part [5:0] of the instruction

ALUOp	Meaning
00	add
01	subtract
10	Look at *funct* field
11	n/a

Output of the control unit:

Instr.	Jump	RegDst	RegWrite	ALUSrc	Branch	ALUOp1	ALUOp0	MemRead	MemWrite	MemtoReg
R-type	0	1	1	0	0	1	0	0	0	0
lw	0	0	1	1	0	0	0	1	0	1
sw	0	x	0	1	0	0	0	0	1	x
addi	0	0	1	1	0	0	0	0	0	0
B-type	0	x	0	0	1	0	1	0	0	x
J-type	1	x	0	x	x	x	x	0	0	x

Control Unit



Uses ALUop signal and funct to decide which function to actually invoke

Generation of Controls

Inst. opcode	ALUOp	Instr. operation	Funct field	Desired ALU action	ALUControl
100010 (LW)	00	load word	xxxxxx	add	0010
100011 (SW)	00	store word	xxxxxx	add	0010
000100 (BEQ)	01	branch equal	xxxxxx	subtract	0110
000000 (R-type)	10	add	100000	add	0010
R-type	10	Subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111
001000 (addi)	00	Immediate	xxxxxx	add	xxxx
000010 (j)	xx	jump	xxxxxx	jump	xxxx

Single-cycle implementation

- The previous design is called single-cycle implementation
- The instruction memory, register file and data memory are all read combinationally
- The new instruction appears to output of instruction memory after some propagation delay, if the address changes
- Operations are done on rising edge of the clock
- The single-cycle microarchitecture executes an entire instruction in one clock cycle
- Simple control unit (why?)

Performance analysis of Single-cycle implementation

- Need some quantity (or metric) for comparison of two design
- How does one measure the effectiveness of new design?

Performance analysis of Single-cycle implementation

- Execution time of a program is a metric
- $Execution\ Time = (\#instructions)\left(\frac{Cycles}{instruction}\right)\left(\frac{Seconds}{cycle}\right)$
- #instructions or length of a program depends on ISA
- Complicated Vs. Simple ISA
- The number of cycle per instruction (on an average) is called *CPI*
- Throughput = 1/CPI
- Assumption: an ideal memory model
- The number of seconds per cycle is the clock period (?)

Performance analysis of Single-cycle implementation

- CPI = 1, for single-cycle implementation
- lw-instruction decides Critical path (T_c)
- $T_c = t_{pcq\_PC} + t_{mem} + \max\{t_{RFread}, t_{select} + t_{mux}\} + t_{ALU} + t_{mem} + t_{mux} + t_{RFwrite}$
- Register read takes longer time than mux selection
- $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFwrite}$

Performance analysis of Single-cycle implementation

- XYZ-organization is contemplating building the single-cycle MIPS processor in a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compute the execution time for a program with 100 billion instructions.

Parameter	Delay (ps)
t_{pcq_PC}	30
t_{mem}	250
t_{RFread}	20
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20

Performance analysis of Single-cycle implementation

- XYZ-organization is contemplating building the single-cycle MIPS processor in a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compute the execution time for a program with 100 billion instructions.
- $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFwrite}$

  = 30 + 2(250) + 150 + 200 + 25 + 20 = 925
- The total execution time =

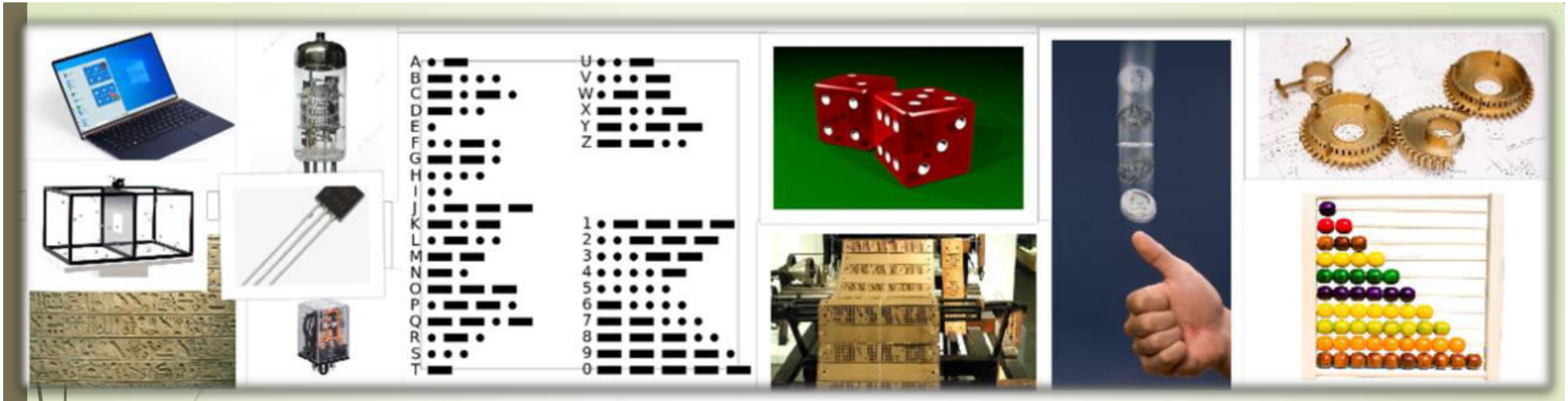
  (100 * 10^9 instrs.) * (1 cycle/instrs.) * (925 * 10^{-12} s/cycle)

  = 92.5 seconds

Parameter	Delay (ps)
t_{pcq_PC}	30
t_{mem}	250
t_{RFread}	150
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20

Summary

- Disadvantages of CISC-style processor
- RISC-style processor organization
- Design of datapath for Single-cycle processor
- Design of controls for Single-cycle processor
- Performance analysis of Single-cycle processor



Computer Architecture (CS F342)

Design and Analysis of Instructions

Design of Control Unit for Reduced Instruction Set Computer (RISC)

Problems of Single-cycle Datapath Design

- Single-cycle design works well but inefficient design
- Clock length (worst-case delay) is same for all instructions
- It is not a balanced design
- CPI is 1
- Use more resources:
  - Adder
  - Memory
- Necessity of balanced datapath design technique by focusing on common-case design & analysis principle

Balanced datapath design: Multi-cycle approach

- Instruction execution can be broken down to smaller steps
  - Is it similar to CISC-style approach?
- Simple instruction can complete the execution earlier than the complex instructions
- One can design the multi-cycle datapath as similar in single-cycle
  - Connecting architectural elements with the memory using combinational logic
  - Next, design the controller

Each clock cycle is now dedicated to perform a single task:
IF,ID, Execute (ALU), Data Memory, Write Back

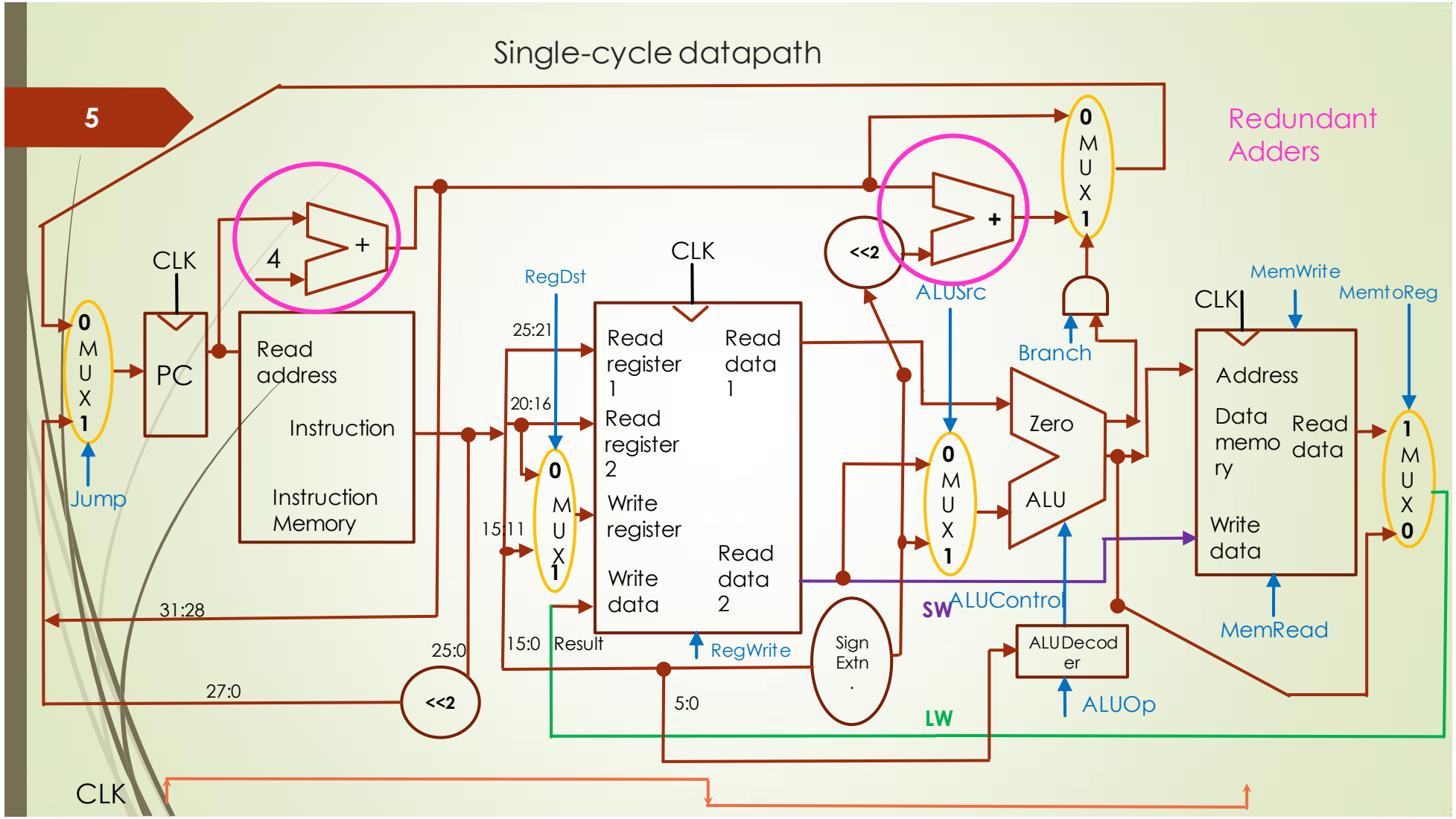
3

Balanced datapath design: Multi-cycle approach

- The key difference is
  - Controller produces different signals on different steps/states
  - A finite state machine approach as in CISC-style

Balanced datapath design: Multi-cycle approach

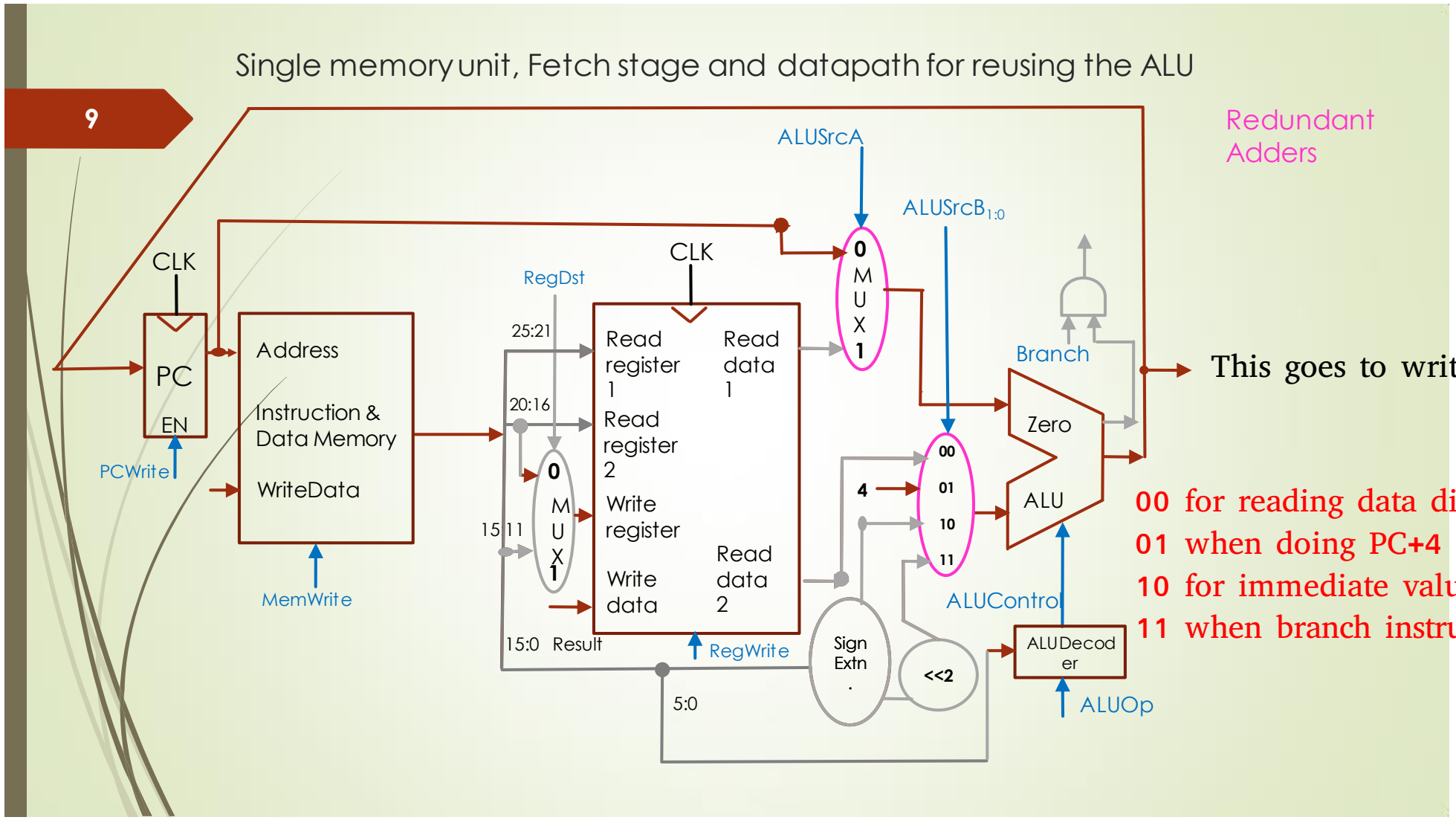
- Combined the instruction and data memory
- Remove the redundant adders
- Incorporate the non-architectural state elements


Single-cycle datapath
5
Redundant Adders
CLK
4
+
0
M
U
X
1
PC
Jump
Read address
Instruction
Instruction Memory
31:28
27:0
25:0
<<2
RegDst
25:21
20:16
15:11
15:0
Result
CLK
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
RegWrite
Sign Extn
5:0
<<2
+
ALUSrc
Zero
ALU
Branch
MemWrite
MemtoReg
CLK
Address
Data memory
Read data
Write data
MemRead
SW
ALUControl
ALUDecoder
ALUOp
LW
CLK

Balanced datapath design: Multi-cycle approach

- Remove the redundant adders
- Where to place this operations?
- How does one control such operations?

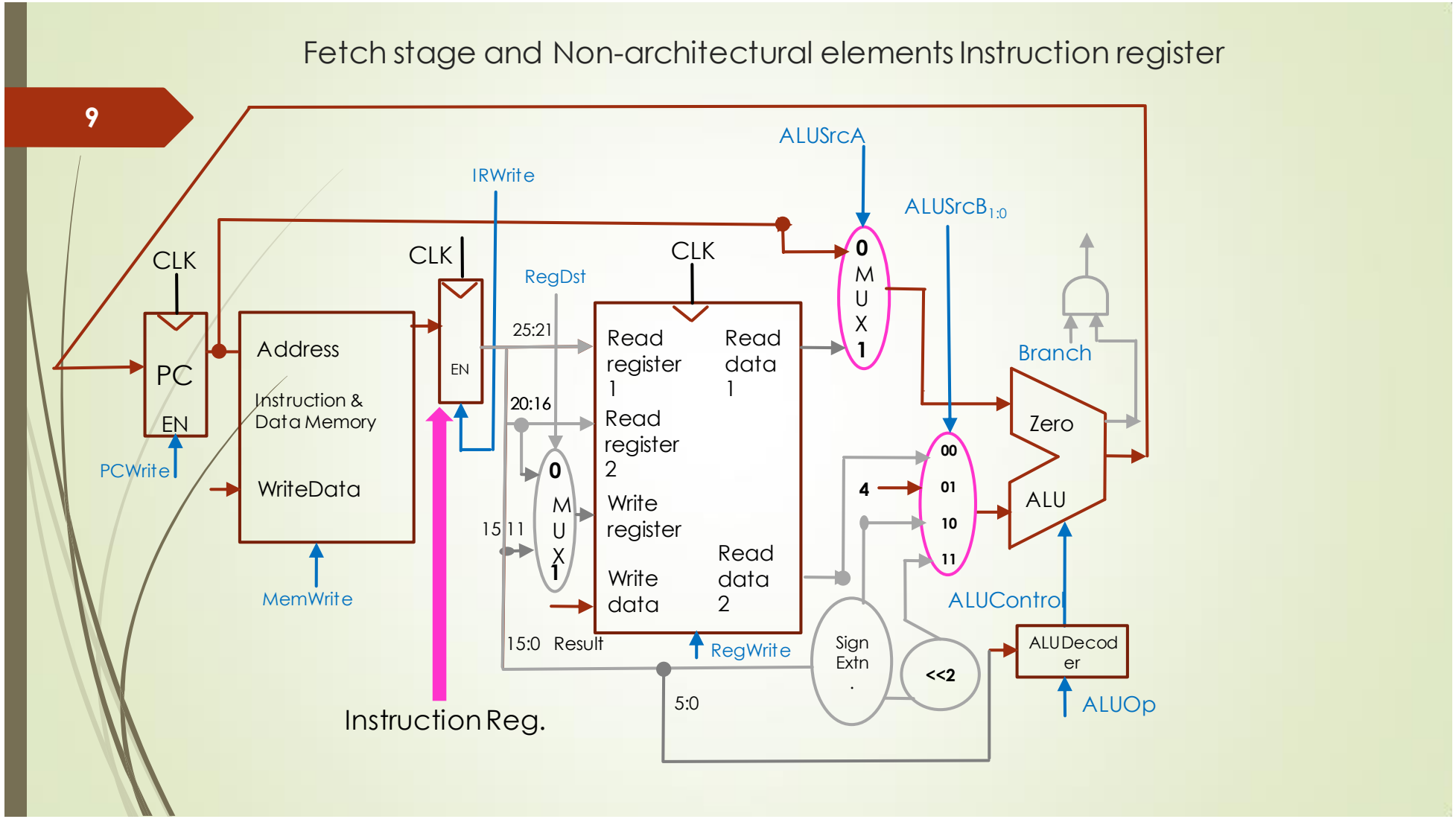
PCWrite used to control if PC is being incremented or not.
Also note that here instruction and data memory are merged into one



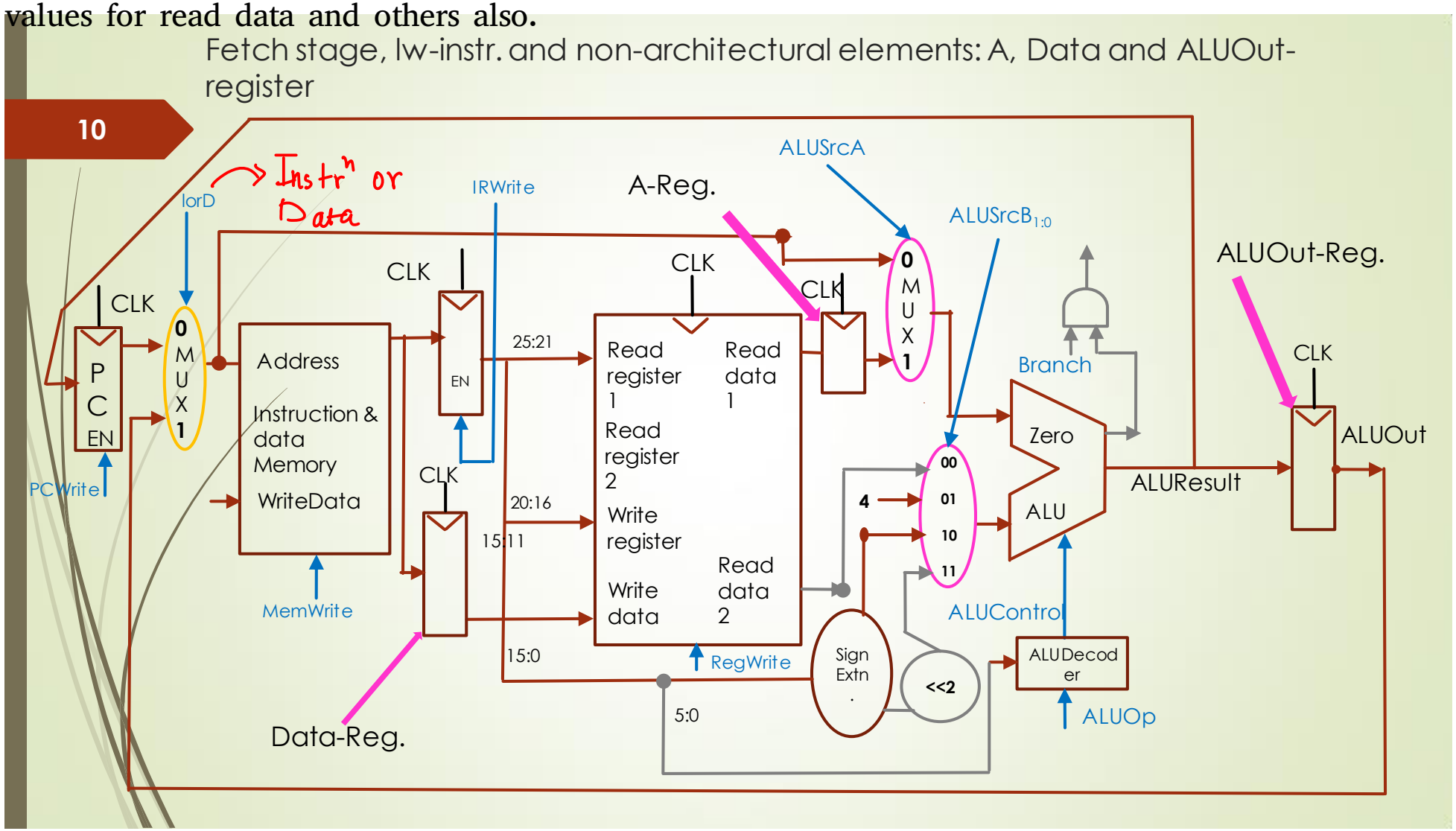
Balanced datapath design: Multi-cycle approach

- Combined the instruction and data memory
- Remove the redundant adders
- Incorporate the non-architectural state elements

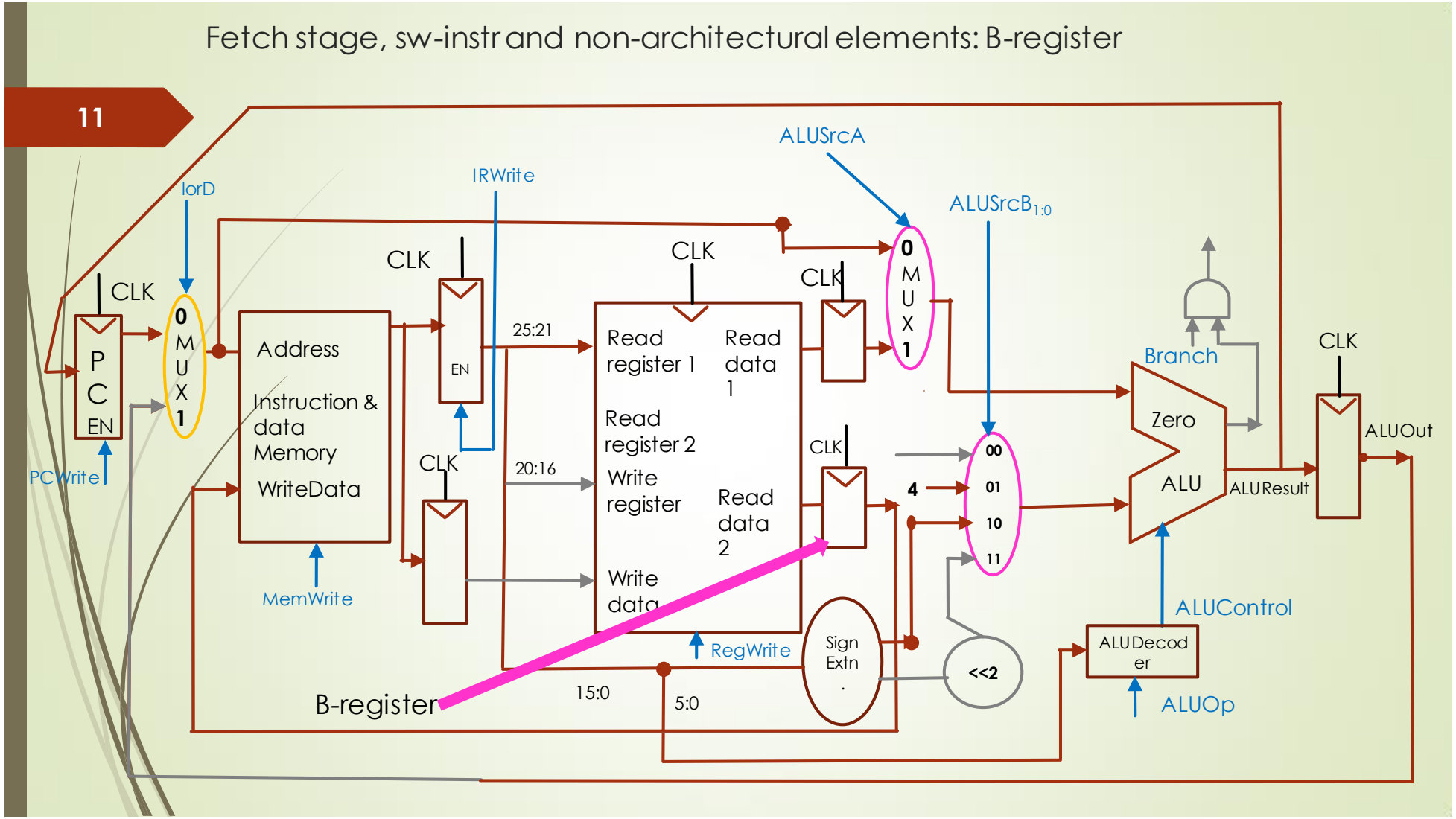
During the IF cycle: IRWrite is set. So the instruction stored in the address corresponding to PC is stored inside the Instr Reg. After this the IRWrite is unset and PCWrite is set. So the PC becomes PC+4 ready to get to next instruction when the full execution of this one is finished.



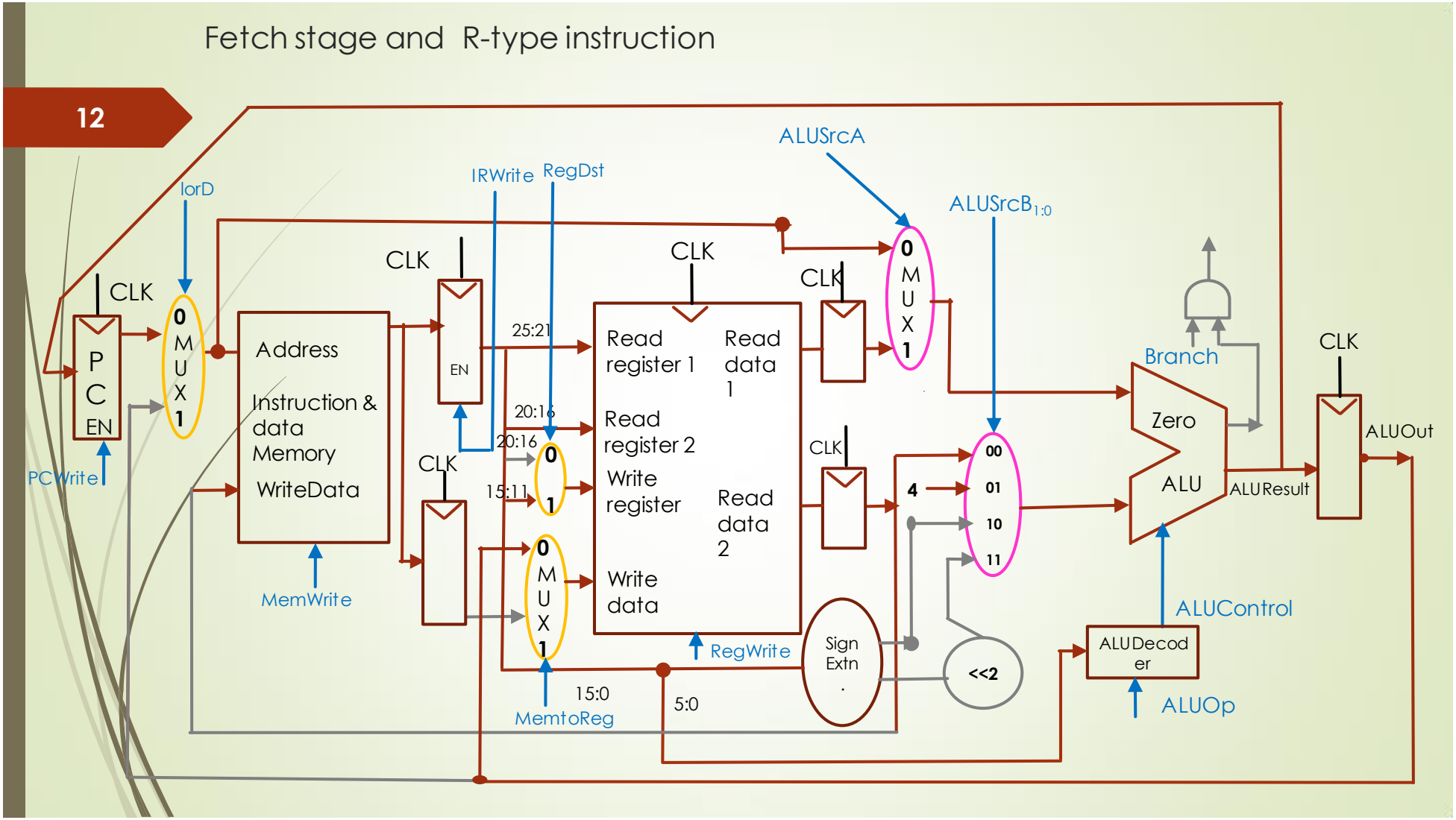
A data register is also needed so as to distinguish clearly when the mem element is acting like data or instr memory. We basically require a register to save information from the previous clock cycle. Each of them has an enable signal also. Here we have used A-Reg to store the data 1 which we can use in the exec cycle later. If this isn't done, then there is a possibility that some other things change in the previous wires so as to give wrong values for read data and others also.



Similar to A-reg, B-reg also needed



Fetch stage and R-type instruction

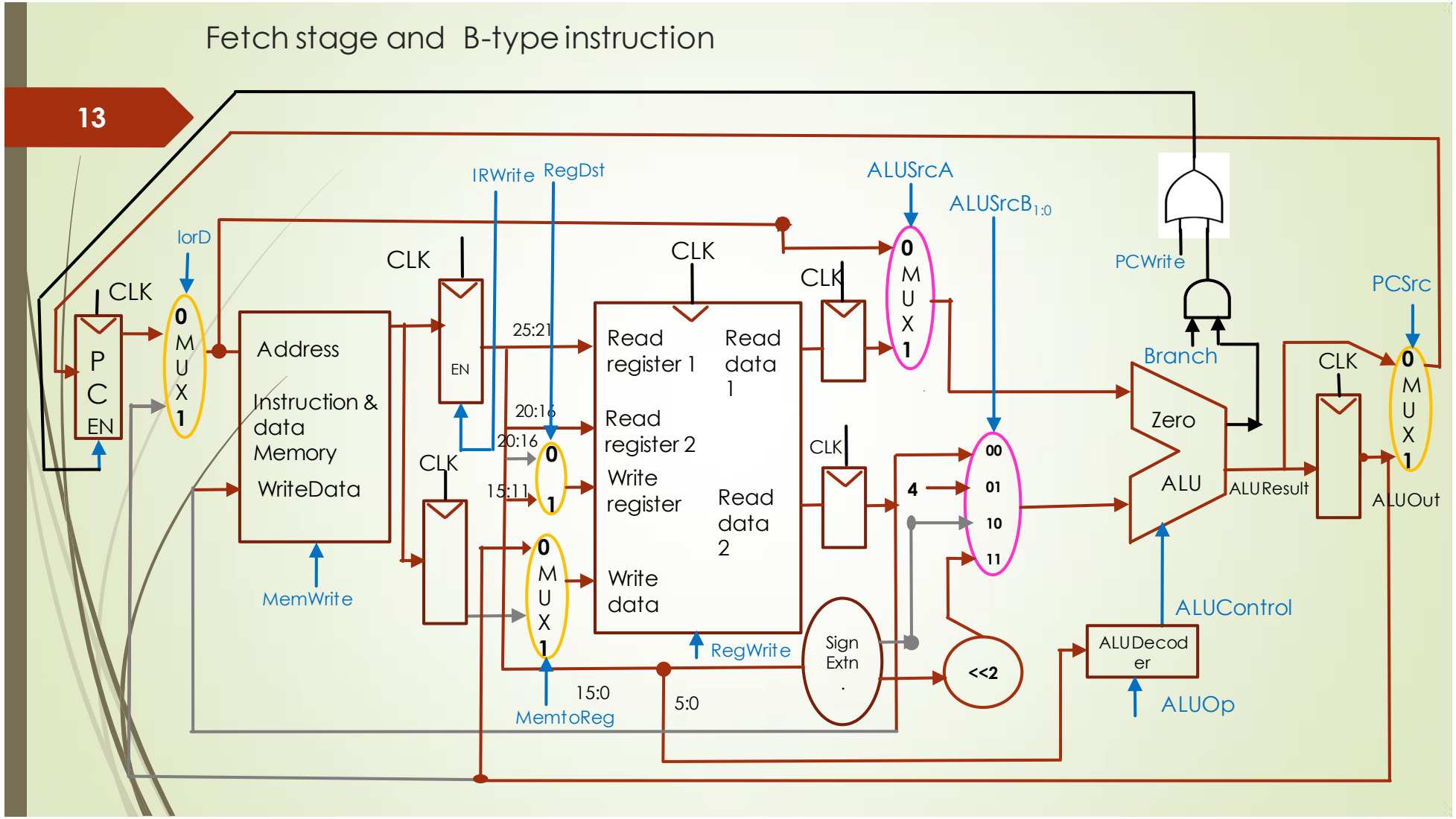


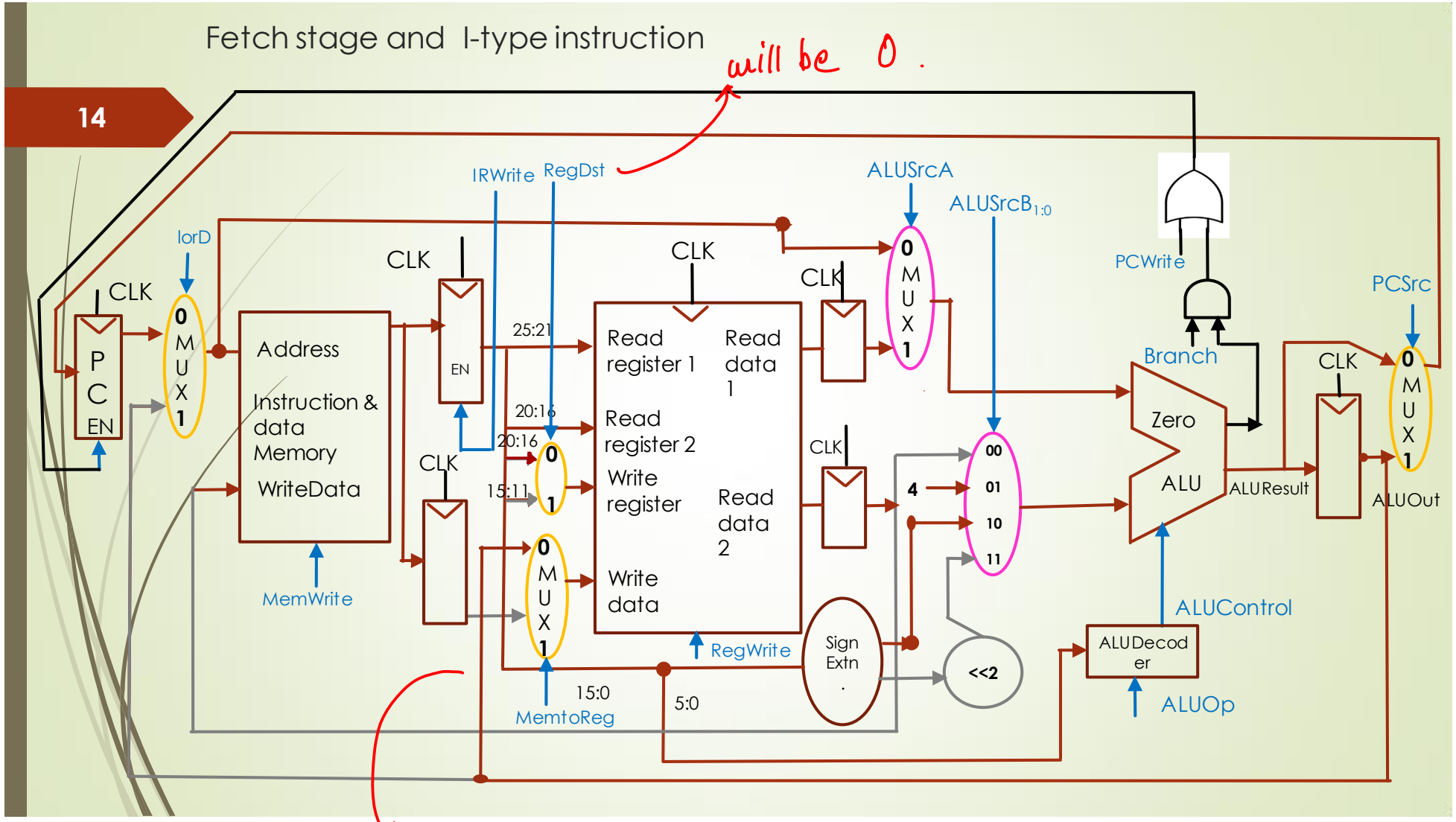
When branch instruction is called, the PC should be enabled. So, either PCWrite will decide or Branch ∧ Zero.

Steps: 1. IF

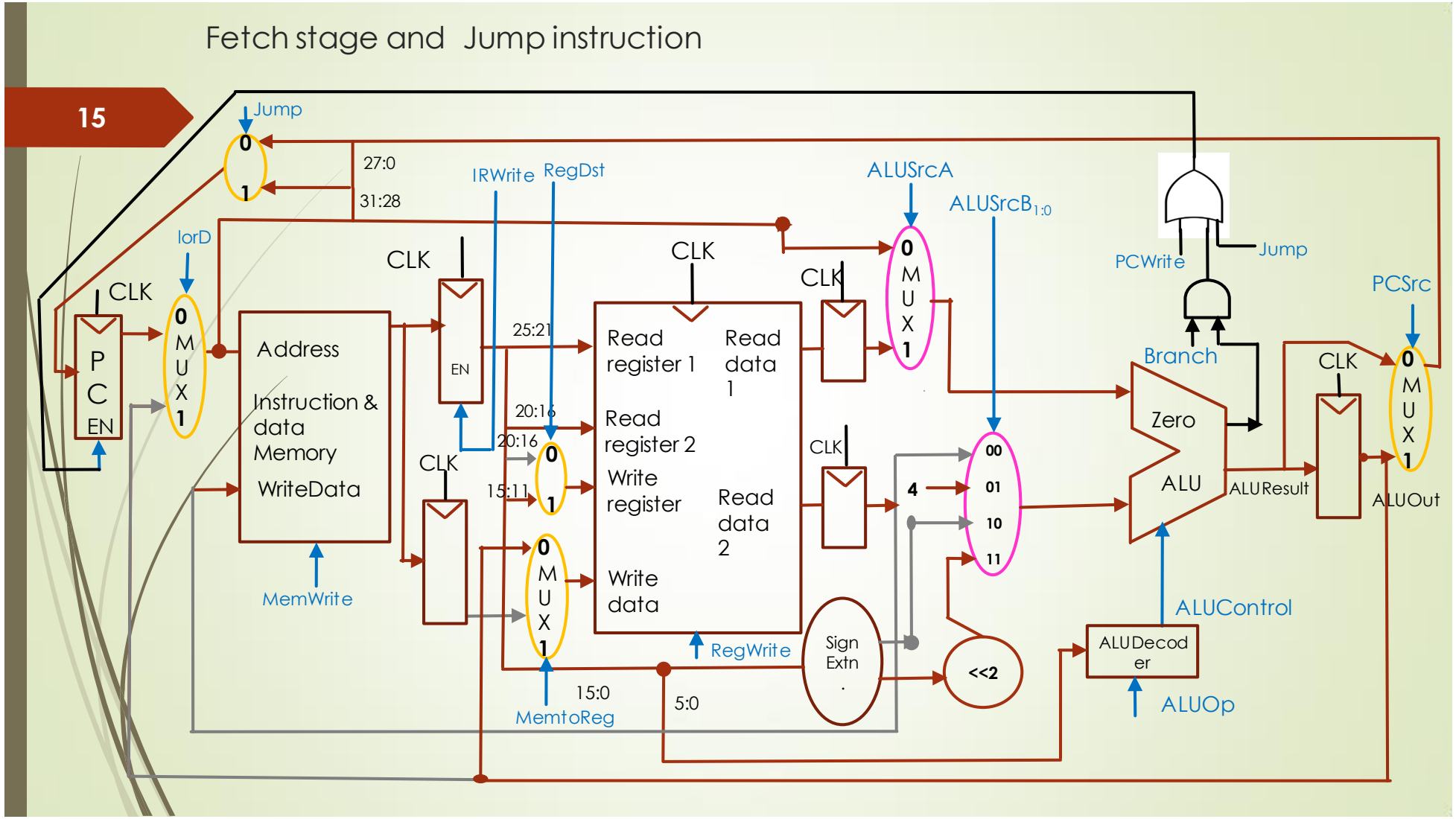
2. ID: In this stage itself ALUResult = PC+4+offset is calculated and stored in ALUOut

3. the A-reg and B-reg will be compared and accordingly PC will be enabled or disabled. PCSrc =1 for branch so, the next address is taken from the ALUOut




Fetch stage and I-type instruction
14
will be 0.
IRWrite
RegDst
ALUSrcA
ALUSrcB1:0
IorD
CLK
PCWrite
PCSrc
Branch
PC
EN
0
M
U
X
1
Address
Instruction &
data
Memory
WriteData
MemWrite
EN
25:21
20:16
20:16
15:11
Read
register 1
Read
register 2
Write
register
Write
data
Read
data
1
Read
data
2
RegWrite
Sign
Extn
.
<<2
4
00
01
10
11
Zero
ALU
ALUResult
ALUOut
ALUControl
ALUDecod
er
ALUOp
15:0
MemtoReg
5:0
This wire will carry the final data and write.

Fetch stage and Jump instruction



see this

0 ⟸ 31:0 (ALU out)

1 ⟸ 31:28 (PC+4) + 27:0 (ALU out)

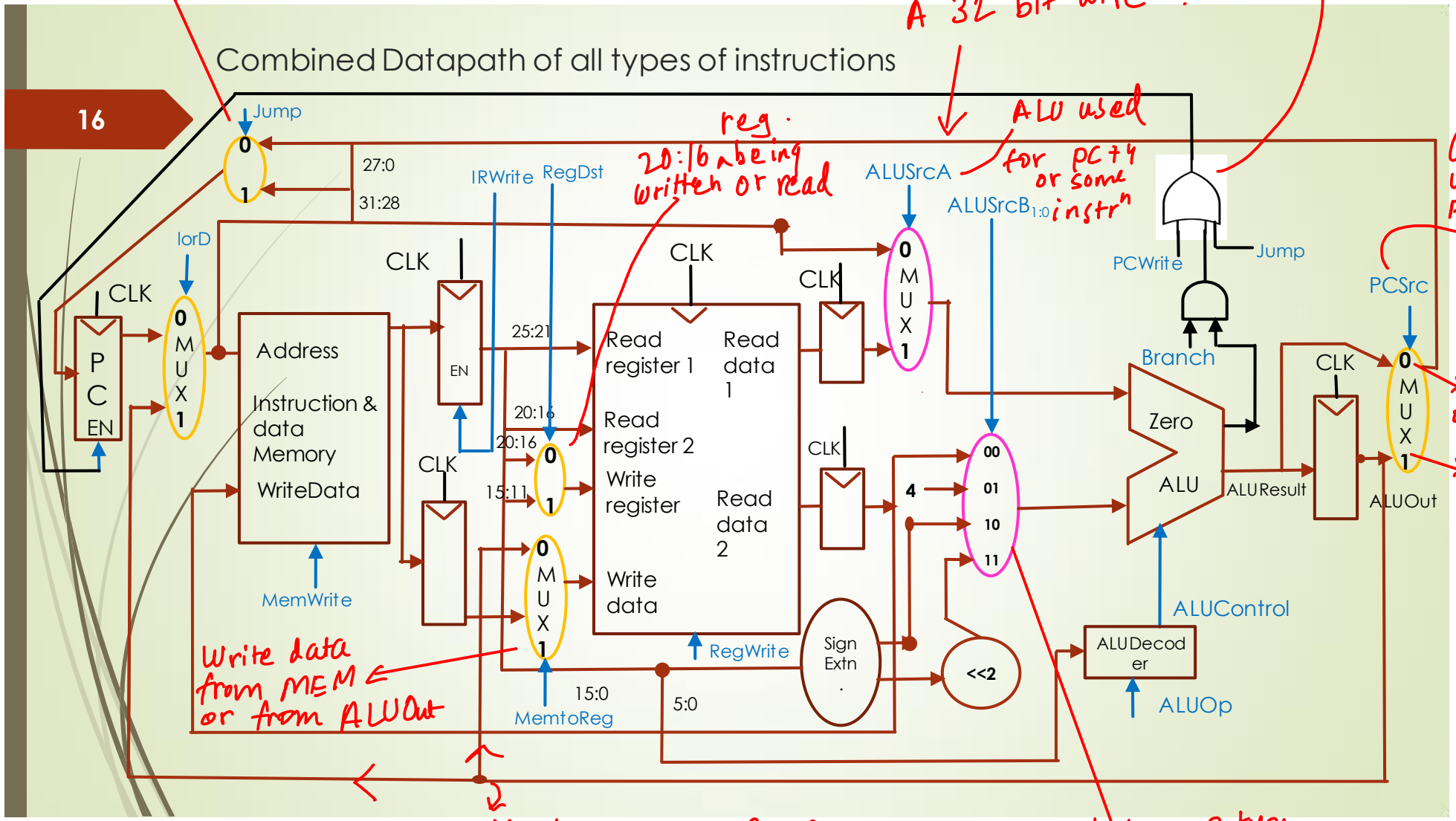
→ PC+4

{IF, Jump, Branch}

Enabler for PC

https://www.cs.fsu.edu/~zwang/files/cda3101/Fall2017/Lecture6_cda3101.pdf

A 32 bit wire.



Two paths. Either write to memory (sw) or write to register (ADD) or read data from mem and then write (LW) to reg.

00: R type
01: PC+4
10: Immediate values
11: Branch

Control signals

- IorD
- Jump
- Memwrite
- IRWrite
- RegDst
- MemtoReg
- RegWrite
- ALUSrcA
- ALUSrcR1:0
- PCWrite
- Branch
- PCSrc
- ALUOp
- ALUControl

Fetch stage

Machine state	Operation	Control signals
T0	InsR ← M[PC]; PC ←PC+4	IorD=0, IRWrite=1, ALUSrcA=0, ALUSrc=01, ALUOp=00, PCSrc=0, PCWrite=1

Decode stage

Machine state	Operation	Control signals
T1	(PC+4) + SigExtn(offset)	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00

Useful for BRZ instruction

LW type instruction

Machine state	Operation	Control signals
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	Data ←M[A+sigEx(off)]	IorD=1
T4	RF[dest] ←Data	RegDst=0, MemtoReg=1, RegWrite=1, **T0**

SW type instruction

Machine state	Operation	Control signals
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	M[A+sigEx(offset)] ← B	IorD=1, MemWrite=1, **T0**

R-type instruction

Machine state	Operation	Control signals
T2	A Op B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=00
T3	RF[dstn] ← A Op B	RegDst=1, MemtoReg=0, RegWrite=1, **T0**

B-type instruction

Machine state	Operation	Control signals
T1	(PC+4) + SigExtn(offset)	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00
T2	A-B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=01, Branch=1, **T0**

Decoding stage

ADDI instruction

Machine state	Operation	Control signals
T2	A OP SigExtn(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00
T3	RF[Destn]← A OP SigExtn(offset)	RegDst=0, MemtoReg=0, **T0**

Jump instruction

Machine state	Operation	Control signals
T2	A OP SigExtn(offset)	Jump=1, **T0**

Clock-cycle needed for the instructions

Instructions	Clock-cycle
LW	5
SW	4
R-type	4
BEQ	3
ADDI	4
J	3

All
IF, ID, Execute, Mem. , Write Back
" " " , Write Back
" " "
" " " , write Back.
" " "

What is next?

- All the ALU operations (ALUOp) similar as in Single-cycle approach
- Generate the control signals using:
  - Hardwired-based approach
  - Microprogrammed-based approach
- Inputs of the control unit similar as in Single-cycle approach

Performance Analysis

- The program consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this program.

Performance Analysis

- The program consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this program.
- The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used
- Average CPI = (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12
- For Single-cycle approach, Avg. CPI is 1

Performance Analysis

- Each cycle involved one ALU operation, memory access, or register file access
- Assumptions:
  - the register file is faster than the memory and
  - writing memory is faster than reading memory
- Datapath has two possible critical paths:
- $T_c = t_{pcq\_PC} + t_{mux} + \max\{t_{ALU} + t_{mux}, t_{mem}\} + t_{registerRead}$

Performance Analysis

- XYZ-organization is contemplating building the multi-cycle MIPS processor instead of the single-cycle processor. For both designs, the organization plans on using a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compare each processor's execution time for a program with 100 billion instructions

Parameter	Delay (ps)
t_{pcq_PC}	30
t_{mem}	250
t_{RFread}	20
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20

Performance Analysis

- XYZ-organization is contemplating building the multi-cycle MIPS processor instead of the single-cycle processor. For both designs, the organization plans on using a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compare each processor's execution time for a program with 100 billion instructions
- $T_c = t_{pcq\_PC} + t_{mux} + \max\{t_{ALU} + t_{mux}, t_{mem}\} + t_{registerRead}$
- T_C = 30 + 25 + 250 + 20 = 350 ps
- Execution time =

  (100 * 10^9 instrs.) * (**4.12** cycle/instrs.) * (350 * 10^{-12} s/cycle)

  = 133.9 seconds

Parameter	Delay (ps)
t_{pcq_PC}	30
regSetup	20
t_{mem}	250
t_{RFread}	20
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20

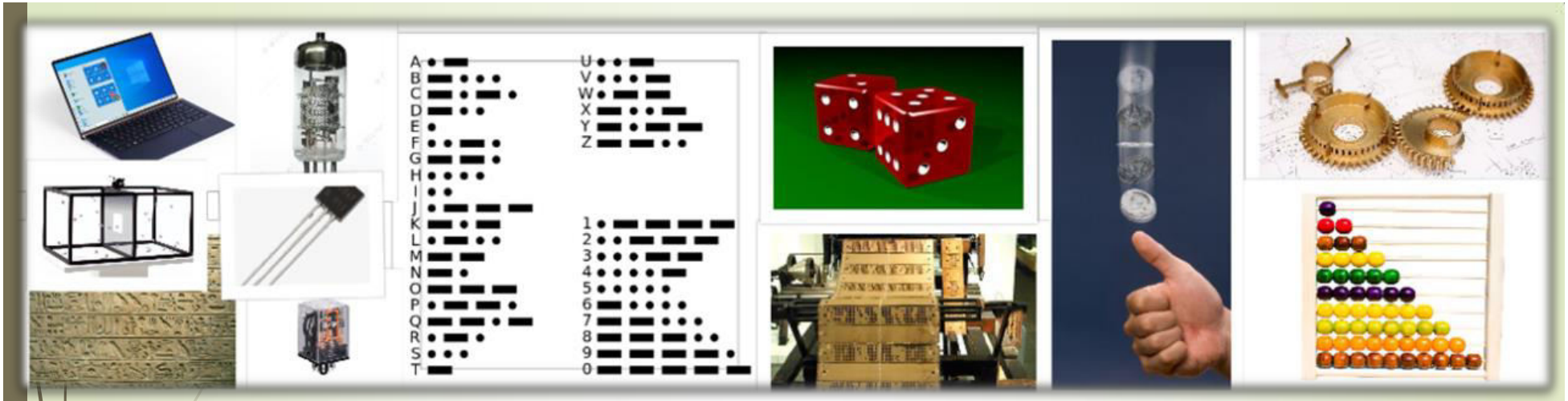
Performance Analysis: A Comparison

- For multi-cycle, T_C = 350 ps and CPI = 4.12
- For single-cycle, T_C = 925 ps and CPI = 1
- For multi-cycle, execution time = 133.9 seconds
- For single-cycle, execution time = 92.5 seconds
- This example shows multi-cycle processor is slow than the single-cycle processor; why is it so?
- Multi-cycle processor is less expensive
- It has 5-nonarchitectural elements

Parameter	Delay (ps)
t_{pcq_PC}	30
regSetup	20
t_{mem}	250
t_{RFread}	20
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20

Summary

- Disadvantages of Single-cycle processor
- Necessity of balanced design approach and multi-cycle processor
- Datapath design of multi-cycle processor
- Design of Controls for multi-cycle processor
- Performance analysis of multi-cycle processor
- Comparison between single-cycle and multi-cycle processor
- Disadvantages of multi-cycle processor



Computer Architecture (CS F342)

Design and Analysis of Instructions

Pipelined-based Design Fundamentals

Reason: We have to take the cycle completion time to be = critical stage (generally which is the memory stage)

It does increase but not 5 times

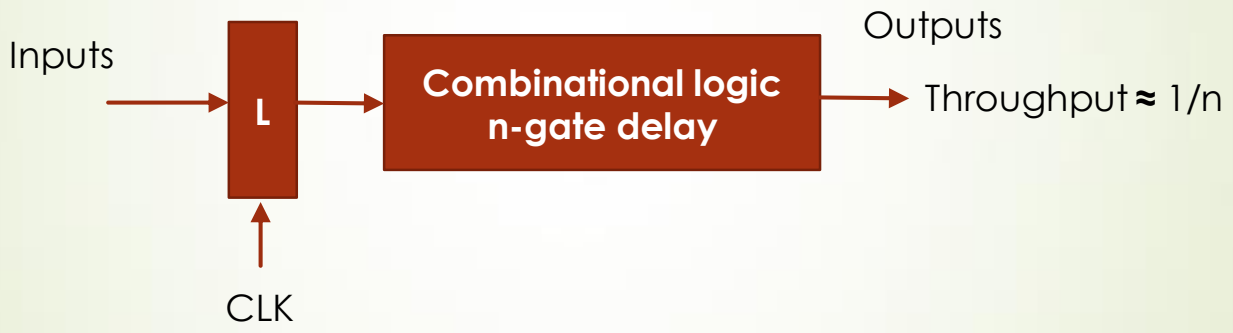
Problems of Multicycle Processor

1

- The fundamental problem
  - Split the slowest instruction, lw, 5-steps
  - Processor clock cycle time does not improve 5-times
- The steps take unequal length of time
- 5-non-architectural register and a additional multiplexer

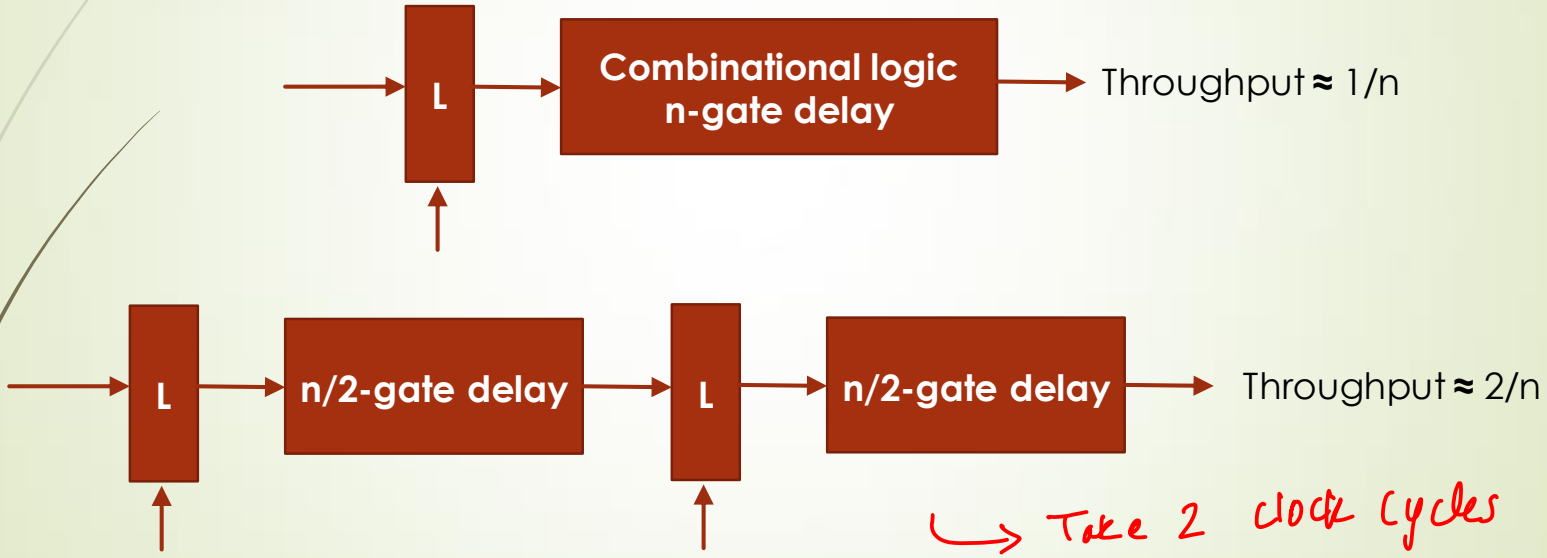
How to improve the processor performance

- Measure throughput (output/unit-time)
- For example



How to improve the processor performance

- For example

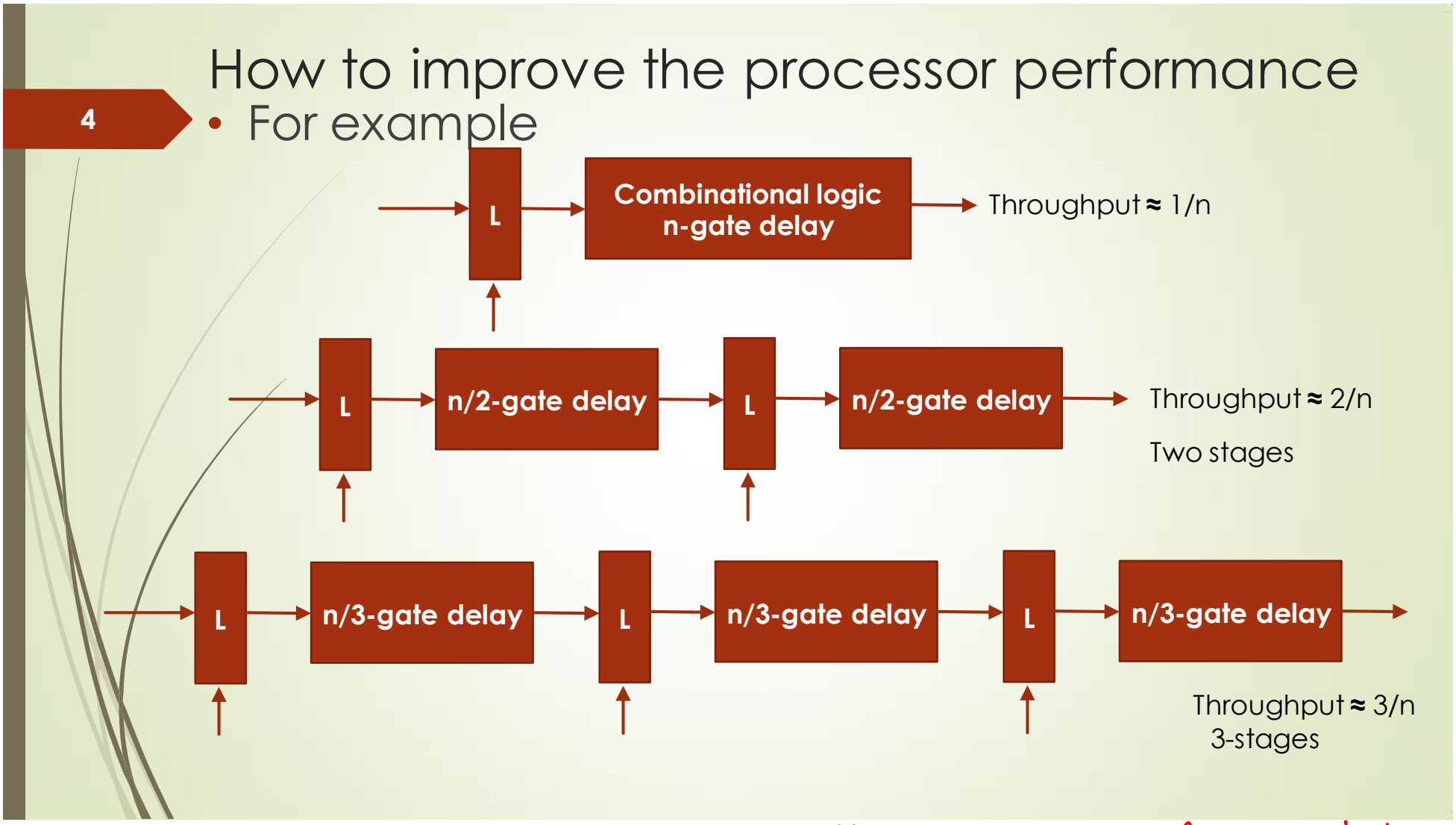


In 1st cycle, Ins1 goes to second latch.
In 2nd " , Ins2 goes to " " and Ins1 goes to dest^n.
Clock Time Period set here = $\frac{n}{2}$ gate delay

How to improve the processor performance

4

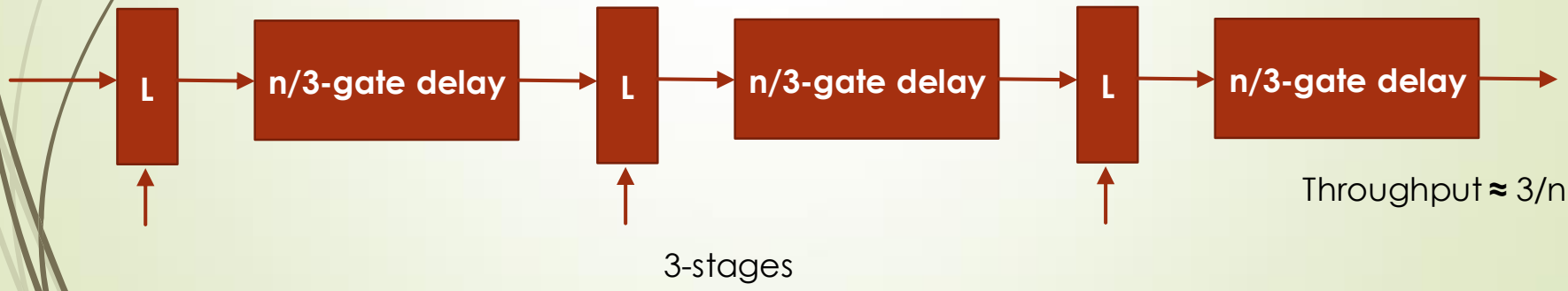
- For example



- Increasing such pieces would increase the no. of instructions processed per unit time
- However, Latch cost and delay would be there too.

Pipelined-based Design Methodology

- k-fold increase in throughput
  - Increase in performance
    - Partitioning the logics
    - Adding new buffer
  - Inputs are overlapped in execution



Limitations of Pipelined-based Methodology

- Assumed inter-stage buffers does not introduce additional delay
- Increase in performance as the stages increase
- What if the stages increase to infinite

Limitations of Pipelined-based Methodology

- What if the stages increase to infinite
- Constrains
  - Clocking
  - Physical limitation on partitioning the logics
- Cost

Minimum clock period in Pipeline-based Systems

- Pipelined-based design
  - Combinational logic (F)
  - Latch (L)
- Max. propagation delay in F: T_M
- Min. propagation delay in F: T_m
- Proper latching delay: T_L

Minimum clock period in Pipeline-based Systems

- Consider the 2-scenarios
- Case-1:
  - Inputs x_1 applied at the stage at time T_1
  - Outputs of F must be valid at $T_1 + T_M$
  - Latching at L of the outputs must be valid until: $T_1 + T_M + T_L$

Minimum clock period in Pipeline-based Systems

- Case-2:
  - Inputs x_2 applied at the stage at time T_2
  - Effect of the outputs can be found at least at $T_2 + T_m$
  - Condition of 2-nd set of signals does not overrun the 1-st set: $T_2 + T_m > T_1 + T_M + T_L$
- Clock period (T): $T_2 - T_1 > T_M - T_m + T_L$
- Max. clocking rate cannot exceed 1/T

Earliest time at which the second instr reaches the latch must be more than the time it takes for the latch to be free of the first instr.

Minimum clock period in Pipeline-based Systems

- Clock period has two parts
  - T_M-T_m
  - T_L
- T_M-$T_m \approx 0$
- T_L:
  - feedback loop and stabilizing of the signal
  - worst-case clock skew

This can effectively be made 0. But T_L would always be there.

Tradeoff between Cost and Performance

- Cost of non-pipelined design: G
  - Gate count
- Cost of adding a latch: L
- Cost of k-stages pipelined design (C): G + k * L
- Cost of pipeline design increases linearly w.r.t depth of pipeline

Total gate count is same => cost also same

Tradeoff between Cost and Performance

- Consider the latency in the non-pipeline design: T
- Performance or throughput: 1/T
- Throughput of pipelined design (P): 1/(T/k + S)
- The additional delay S because of latches
- P is a non-linear function of k

Time taken to process 1 instⁿ in pipeline architecture.

Tradeoff between Cost and Performance

- Cost/performance ratio

$$\frac{C}{P} = \frac{G + k * L}{\frac{1}{(\frac{T}{k} + S)}}$$

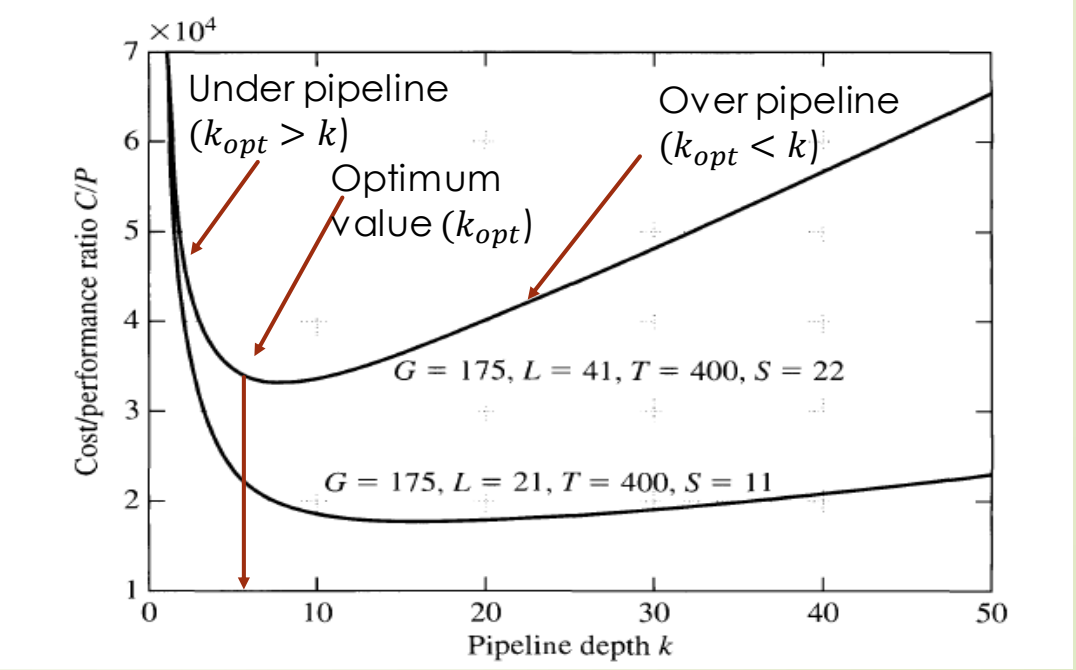
$$= LT + GS + LSk + \frac{GT}{k}$$

- Find minimum cost/performance ratio

Tradeoff between Cost and Performance

- Find minimum cost/performance ratio
- First derivative (w.r.t k)

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

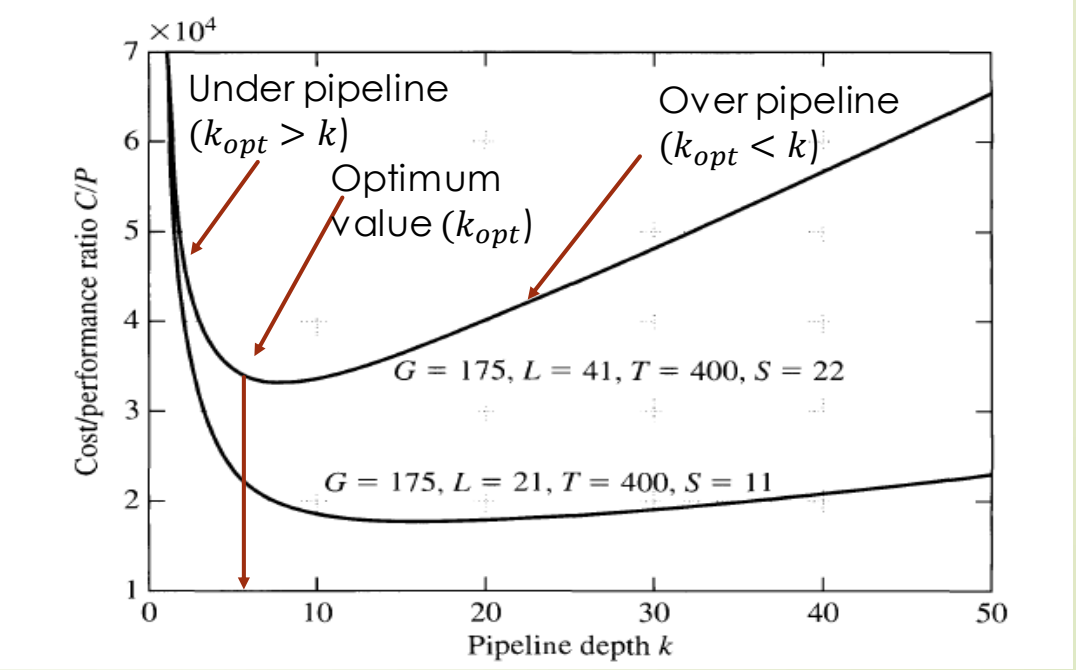


Tradeoff between Cost and Performance

- Find minimum cost/performance ratio
- First derivative (w.r.t k)

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

- **No consideration on dynamic behavior or runtime**



Pipeline Idealism

- Motivation: k-stages pipeline increases k-fold increase in throughput
- In reality this is difficult to achieve
- Are there hidden assumptions?

Pipeline Idealism

- Are there hidden assumptions?
- Yes, 3-assumptions, called pipeline idealism
  - Uniform sub-computations
  - Identical computations
  - Independent computations

As if division of components is possible

As if each unit takes same delay

As if the components are not interconnected (like || mein hota then ?)

Pipeline Idealism: Uniform sub-computations

- The computation can be evenly partitioned into uniform-latency sub-computations
  - No (minimize) internal fragmentation
  - No (minimize) additional delay by inter-stage buffer & clocking

Pipeline Idealism: Uniform sub-computations

- The computation can be evenly partitioned into uniform-latency sub-computations
  - No (minimize) internal fragmentation
  - No (minimize) additional delay by inter-stage buffer & clocking

Pipeline Idealism: Identical computations

- The same computation is to be performed repeatedly for all instructions (or for all input data set)
  - No (minimize) external fragmentation
  - All pipeline stages are always be utilized

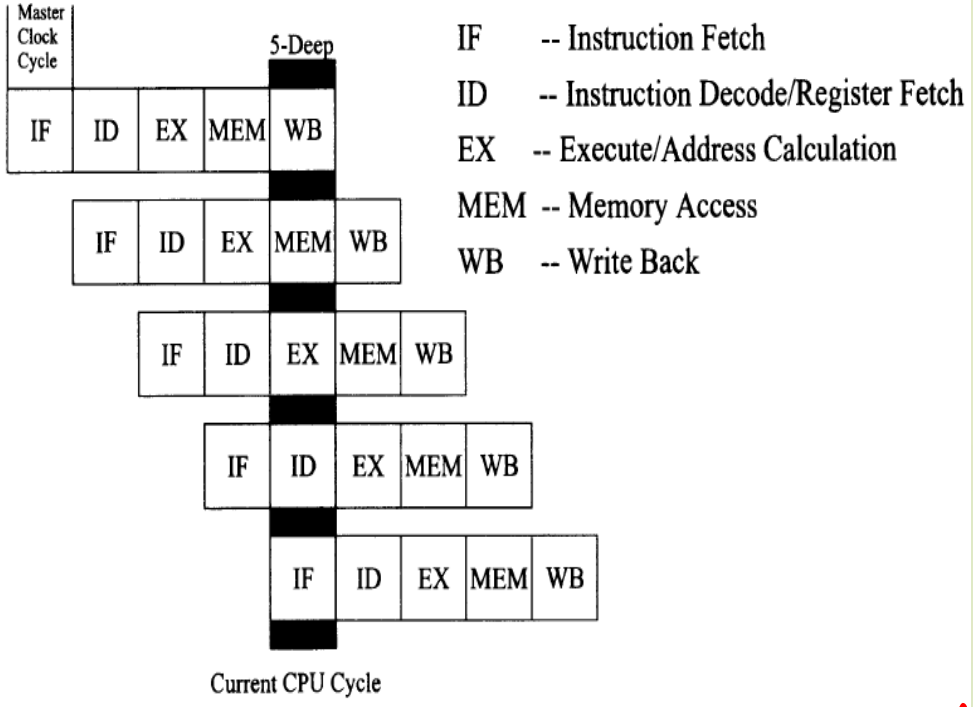
Pipeline Idealism: Independent Computations

- No data or control dependencies between any pair of computations
- Pipeline operates in streaming mode

Actual Implementation

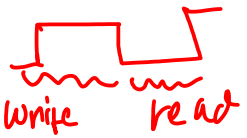
Instruction pipeline or pipelined processor

- An Implementation technique
  - Exploits parallelism among the instructions
  - Overlapping the execution
- Instruction cycle
  - A logical concept
- Machine cycle
  - A physical concept
- Fill time
- Drain time



Note: WB stage can coincide with ID → Reading and writing in Reg file both.

So, to avoid this, ← 1 clock cycle , divide into two parts.



The Pipeline architecture is kind of mix of both multicycle and single cycle. The architecture is same as single cycle. However the single cycle is broken down into 5 cycles for processing each stage.

Summary

- Problems of multicycle design methodology
- Pipelined-based design methodology
- Limitations and optimum value for depth
- Pipeline idealism
- Instruction pipeline technique

Computer Architecture (CS F342)

Design and Analysis of Instructions

Design of Datapath & CU for MIPS-based Pipelined Processor

MIPS-based pipelined processor

- Powerful way to improve the throughput
- Divide the single-cycle implementation
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- A commercial MIPS processor: R2000/R3000

Latency of each instructions is unchanged, but throughput is ideally 5-times better

MIPS-based pipelined processor

- Delay/slow elements
  - Reading & writing the memory
  - Register file
  - ALU operation
- Each stage takes almost same amount of time
  - Consists of one slow element

Comparison of timing diagram

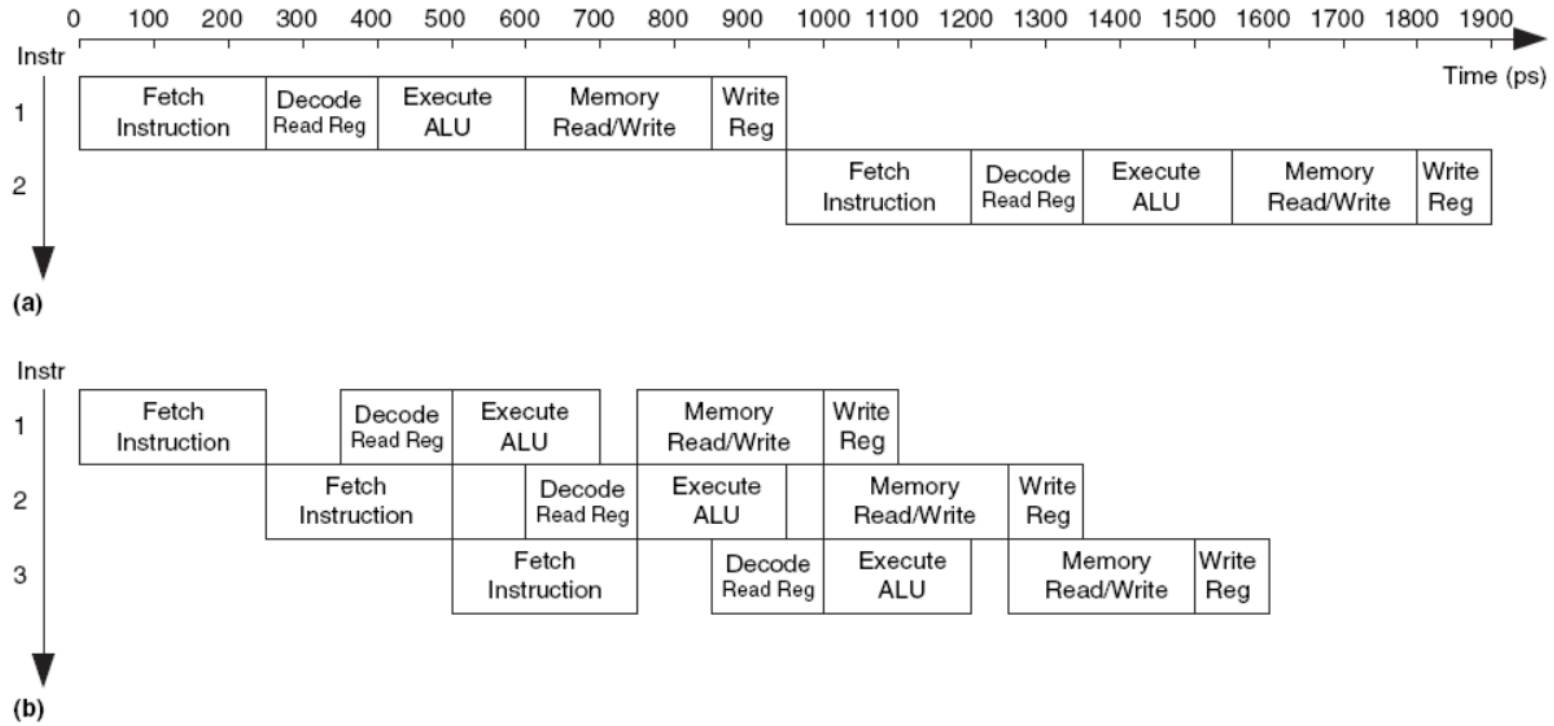
- Delay of the elements

Element	Parameter	Delay (ps)
Register clk-to-Q	T_{pcq}	30
Register setup	T_{setup}	20
Multiplexer	T_{mux}	25
ALU	T_{ALU}	200
Memory read	T_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

Critical Time
= Clock Time period

Comparison of timing diagram

- Delay of MUX & register is not included



Timing diagram of (a) single-cycle processor (b) pipelined processor

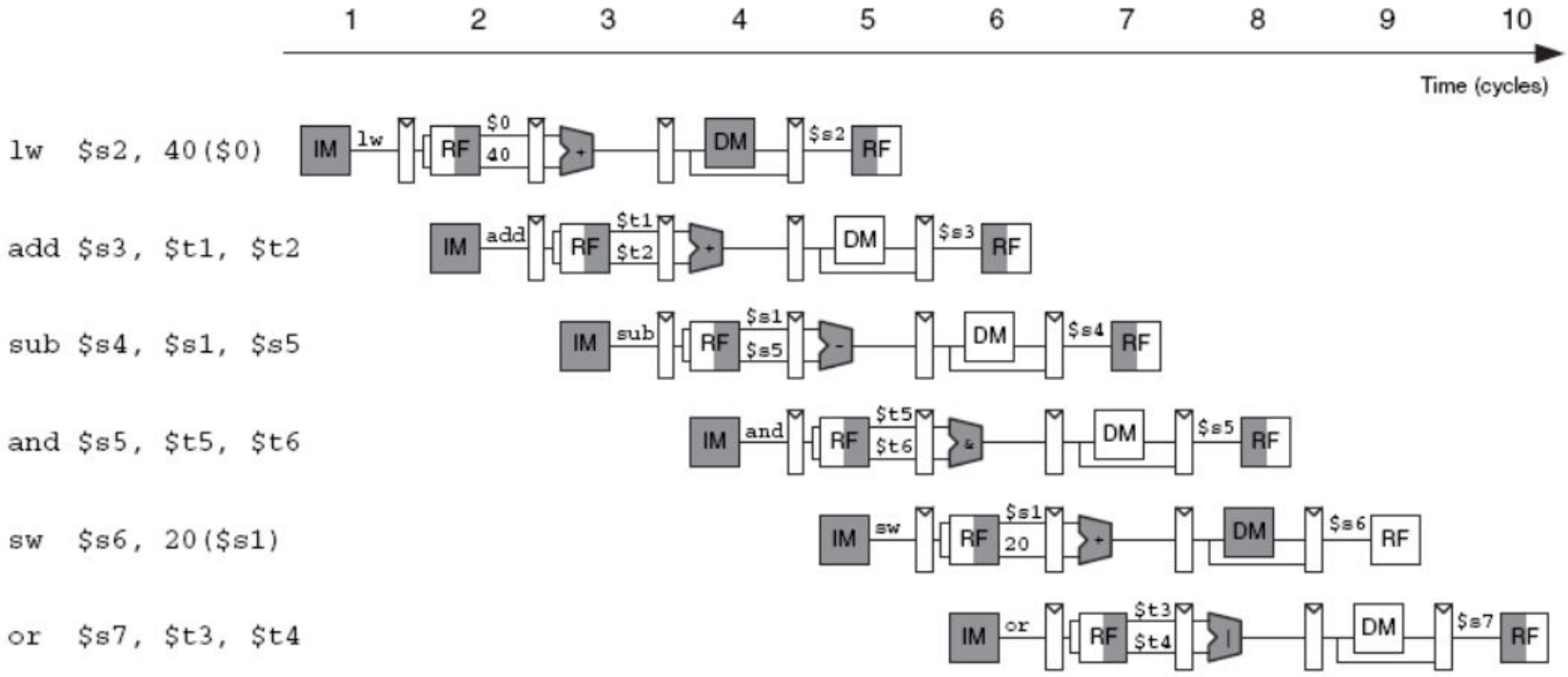
Comparison of timings

- **Single-cycle processor**
  - Instruction latency is 950 ps
  - Throughput 1 instruction per 950 ps
  - 1.05 billions instruction per second

- **Pipelined processor**
  - Length of pipeline stage is 250 ps (mem. access)
  - Instruction latency is 5*250 = 1250 ps
  - Throughput 1 instruction per 250 ps
  - 4 billions instructions per seconds

A view of pipeline in operation

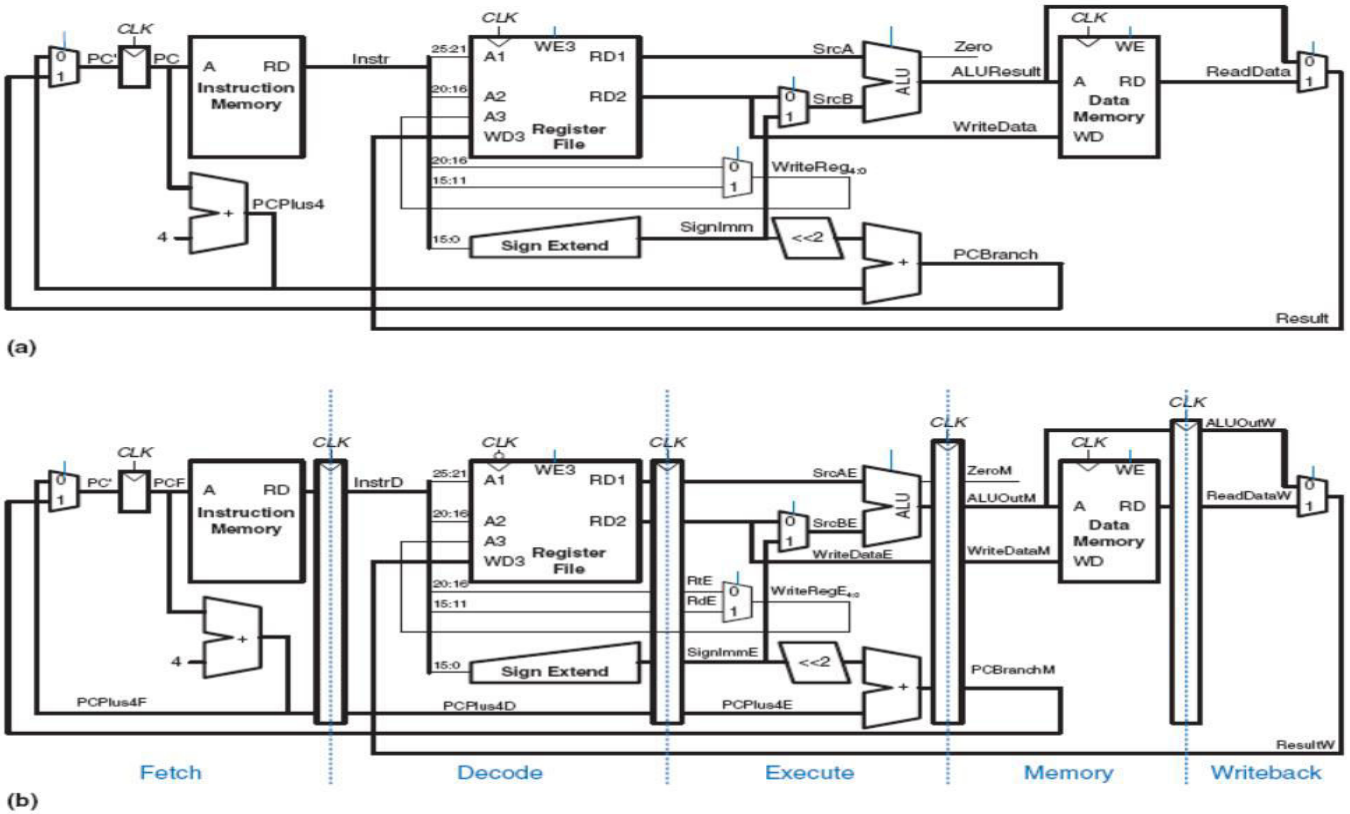
- Major component—instruction memory (IM), register file (RF) read, ALU execution and data memory (DM)
- Register file: write operation in the first part of a cycle and read in the second part



As we are breaking the cycle into multiple cycles, we would need latches. The latches store the intermediate values necessary to be taken forward.

Pipelined Datapath

- Datapath is formed by chopping the single-cycle datapath
  - Five stages separated by pipeline registers



Single-cycle

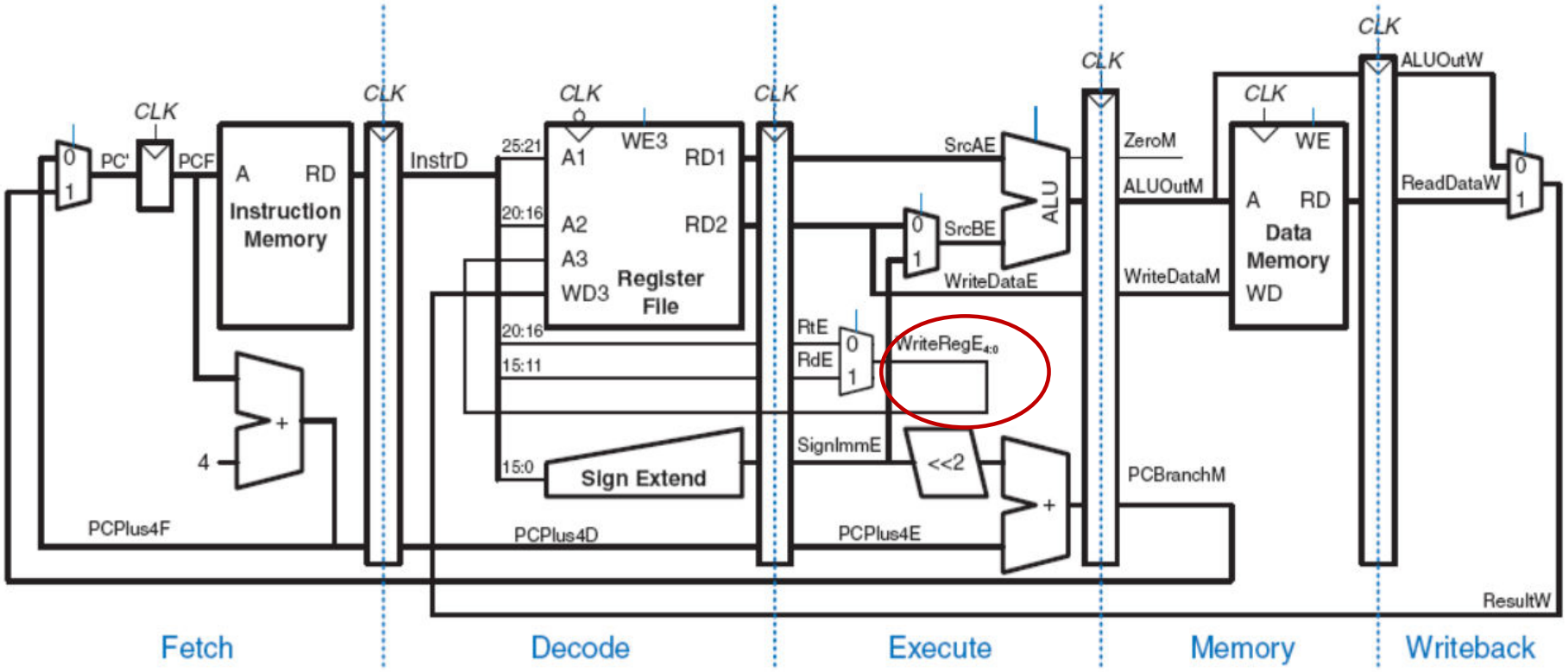
Pipelined

Is this datapath correct?

The data path is incorrect. The reg no. of the register to be written finally isn't forwarded till the 5th cycle. By that time, some other instruction in the pipeline might come and change the reg no. to be written. So, always forward all this info to the next latches until they are needed.

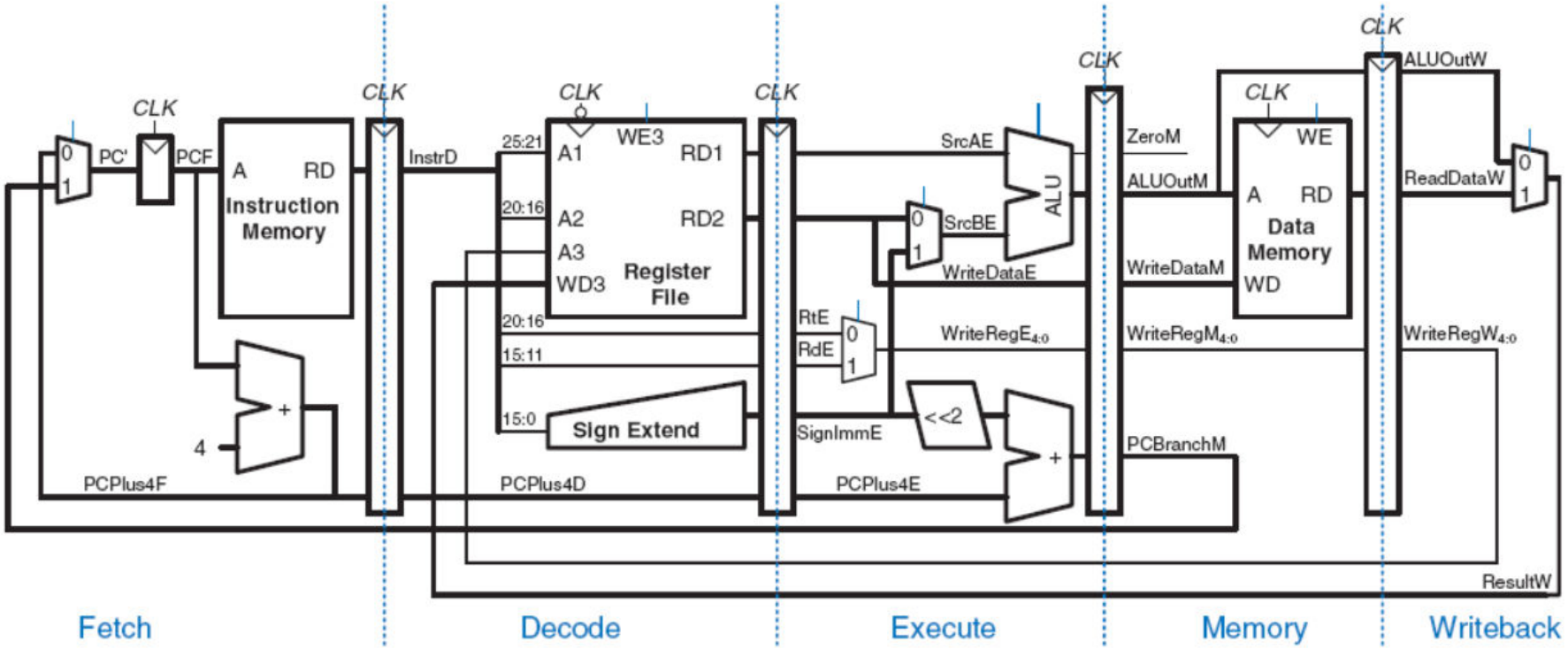
Pipelined Datapath

- Error in the datapath connection



Pipelined Datapath

- Modified datapath

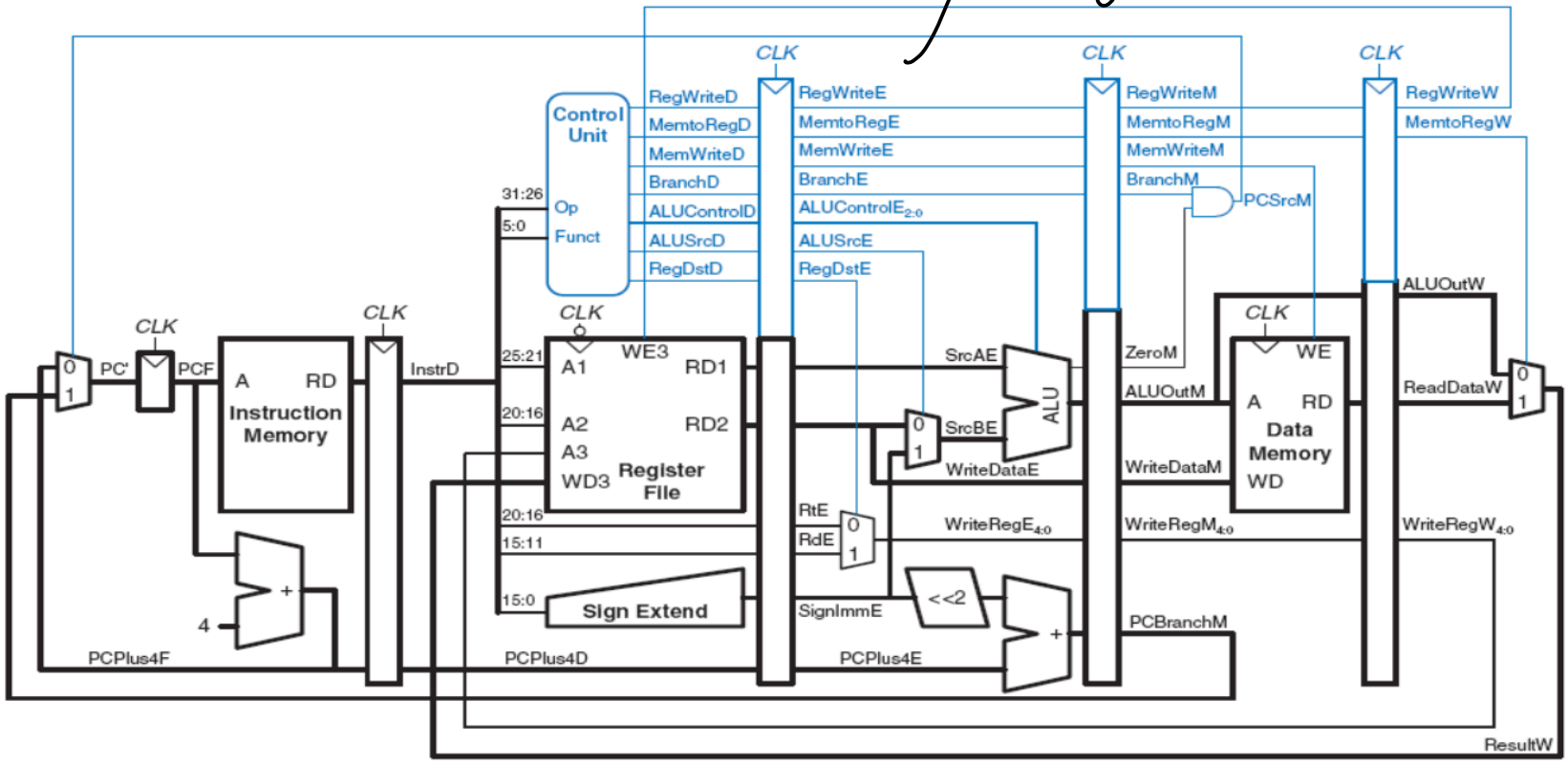


Is there any error in the datapath?

Pipelined Datapath & Control

- CU as in Single-cycle
- Control signals must be pipelined (remain synchronized with instruction)

All the control signals should also be carried in the latches



Summary

- Comparison between single-cycle and pipelined processor
- View of pipeline in operation
- Datapath and CU for pipelined processor

Computer Architecture (CS F342)

Design and Analysis of Instructions

Minimization of Structural & Data Hazards in Pipelined MIPS (RISC) Processor

Pipelined-based Processor

- Pipelining technique exploits parallelism
  - How?
- Can it faces any difficulties while doing so?

Hazards in Pipelined-based Processor

- Conditions that make pipeline to stall: Hazard
  - Structural Hazard
  - Data Hazard
  - Control Hazard

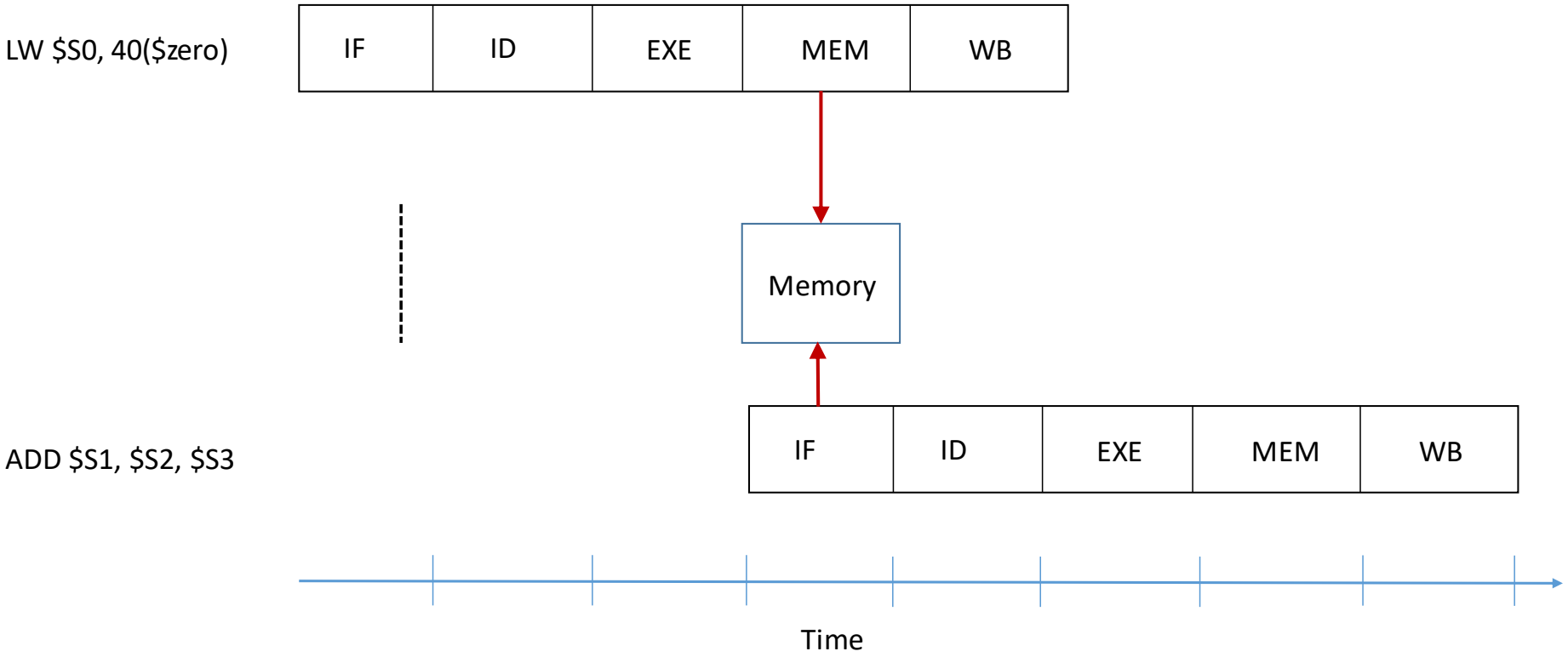
Structural Hazards

- Architectural component does not support parallelism, if we assume
  - Single memory unit
  - More than one instructions trying to write data in register file at the same time

We used this in multicycle
Here it wouldn't work.

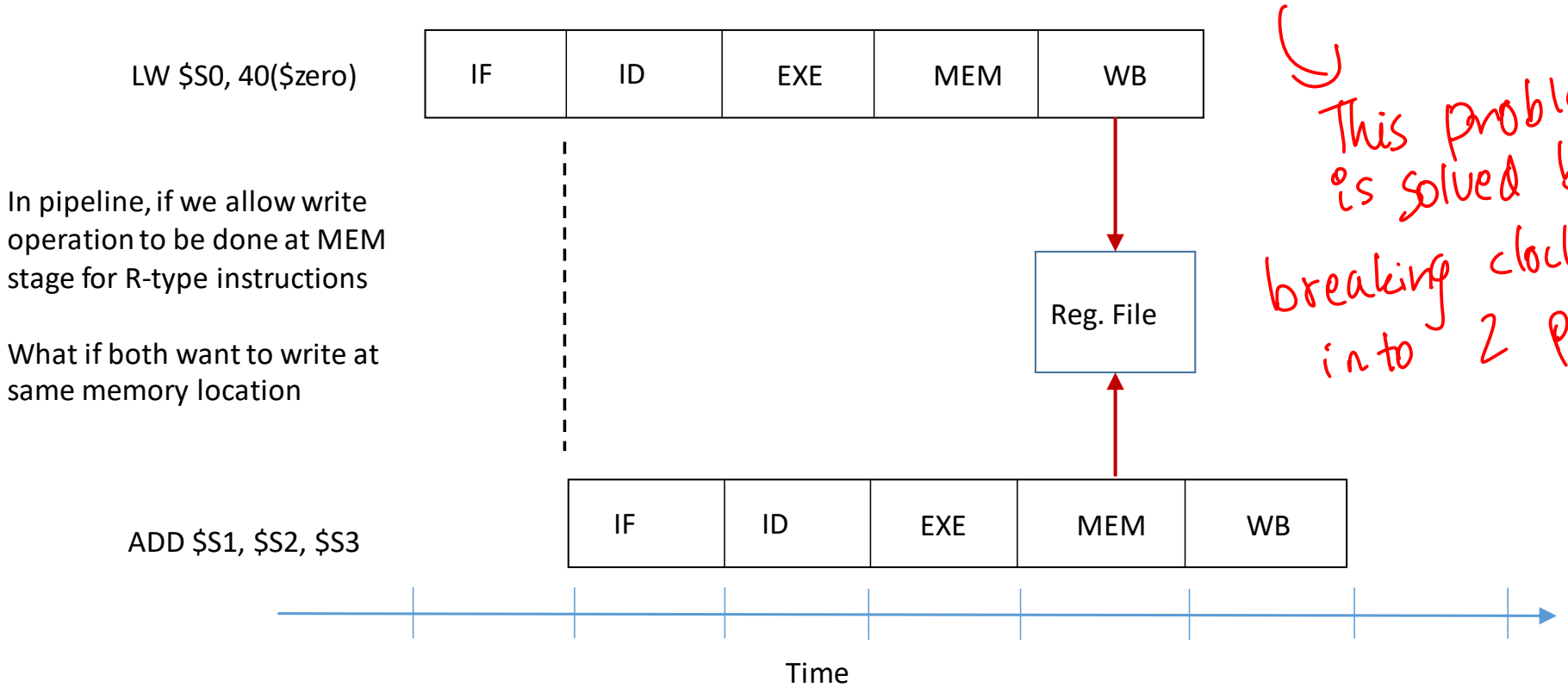
Structural Hazards

- Single memory unit



Structural Hazards

- Two instructions trying to write data in register file at the same time



This problem is solved by breaking clock cycle into 2 parts.

Solution for Structural Hazard

- Incorporate more resources
- Stall the operation
  - arbitration with interlocking

For memory access problem as memory is cheap.

Use this. Break cycle Stall reading.

Dependency between instructions/data

- What are the possible dependencies between two instructions: $Inst_1$ & $Inst_2$?

Dependency between instructions/data

- Assume: $Inst_1$ fetched prior to $Inst_2$
  - $Inst_2$ is data dependent on $Inst_1$
    - if $Inst_1$ writes its output in a register, Reg (or memory location)
    - $Inst_2$ reads as that as its input
  - $Inst_2$ is anti-dependent on $Inst_1$
    - if $Inst_1$ reads data from a register *Reg* (or memory location) which is subsequently overwritten by $Inst_2$

Dependency between instructions/data

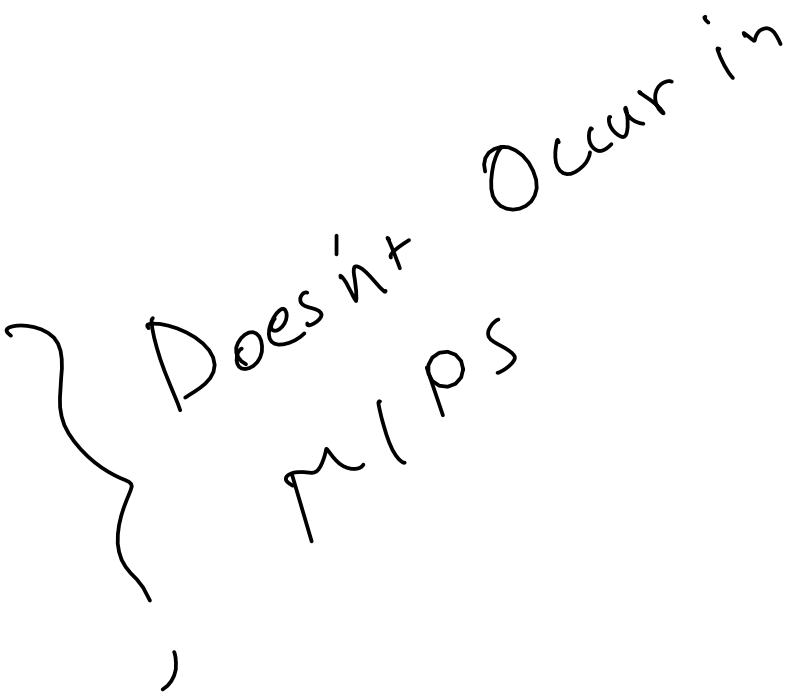
- Assume: $Inst_1$ fetched prior to $Inst_2$
  - $Inst_2$ is output dependent on $Inst_1$
    - if both write in the same register *Reg* (or memory location)
    - $Inst_2$ writes its output after $Inst_1$
  - $Inst_2$ is control dependent on $Inst_1$
    - if $Inst_1$ must complete before a decision can be made whether or not to execute $Inst_2$

Data Hazards

- Data dependences between instructions
  - True or real
  - False or name
- Inst1 & Inst2 are so close that their overlapping would change their access order to register, Reg.

Data Hazards

- Types of data hazards
  - Read after write (RAW)
    - caused by data dependency
  - Write after read (WAR)
    - caused by anti-dependence
  - Write after write (WAR)
    - caused by output dependence



Data Hazards

- WAW hazards occur
  - Write operation in more than one stages
  - Allow an instructions to proceed even when a previous instruction is stalled

- Will it occur in MIPS?

Data Hazards

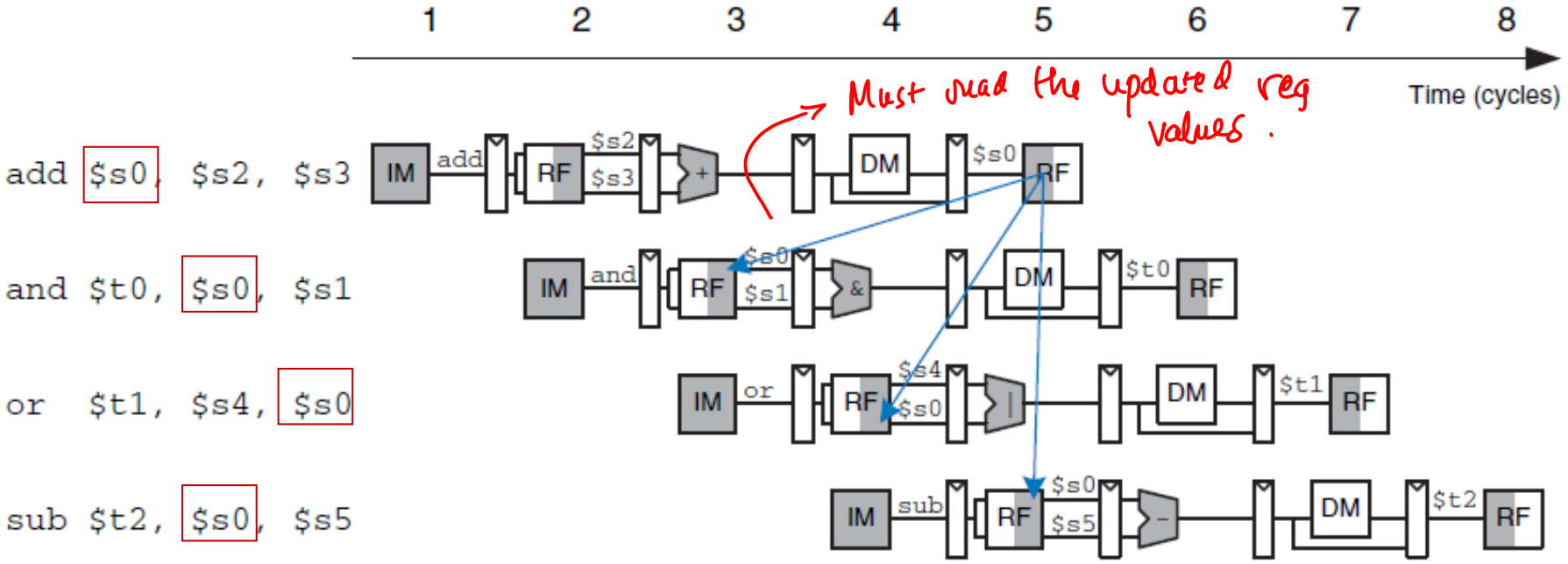
- WAR hazards occur
  - Write stage precedes a read stage

- Will it occur in MIPS?

Data Hazards

- Only RAW hazard occurs in MIPS

↳ read after write



Data Hazards

- How does one solve RAW hazards?

Data Hazards

- How does one solve RAW hazards?
  - Hardware-based solutions
  - Software-based solutions

Data Hazards: H/W-based Solution

- Interlocking -- a simple solution
  - Detect the hazard
  - Stall the pipeline → This will cause loss of parallelism.
- Degrade the speedup

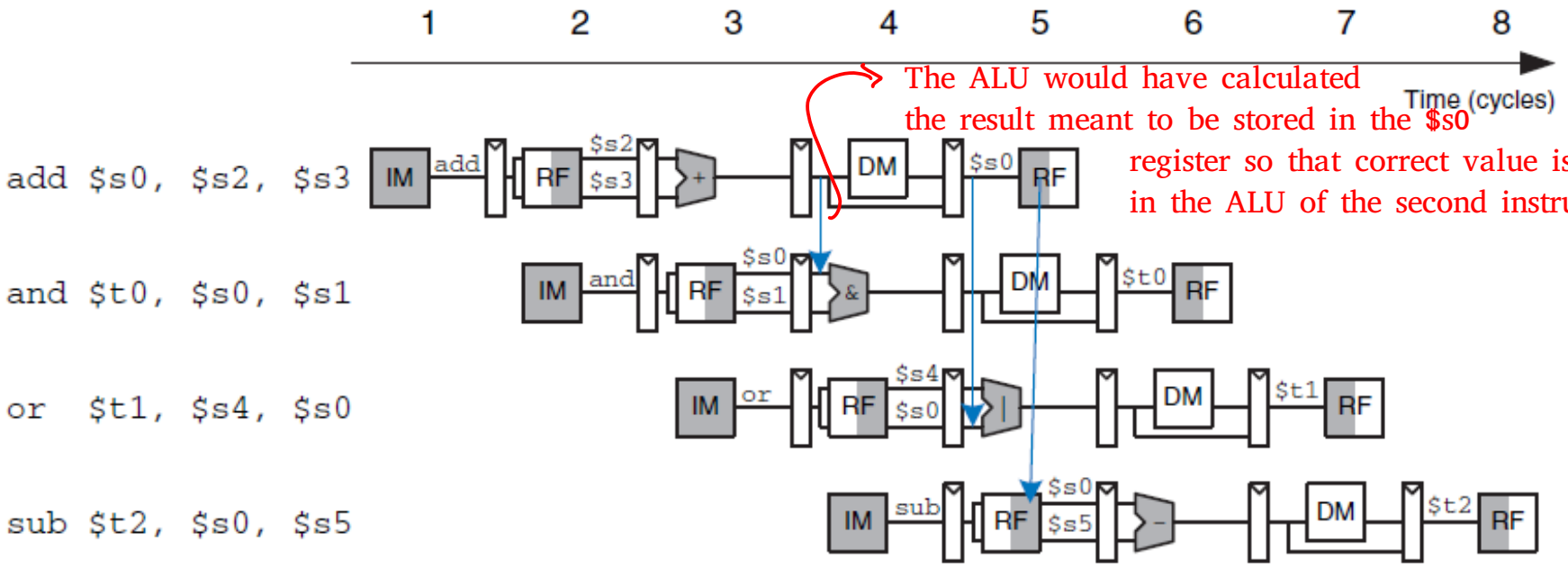
Data Hazards: H/W-based Solution

Use this

- Forwarding – a sophisticated solution
  - The result of the ALU output of $Inst_1$ in the EX stage can immediately forward back to ALU input of EX stage as an operand for $Inst_2$

Data Hazards: H/W-based Solution

- Forwarding or bypassing



The ALU would have calculated the result meant to be stored in the $s0 register so that correct value is input in the ALU of the second instruction

The register file has already been updated in the first half of the clock. So reading won't be a problem in the second half.

Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

  - Put MUXs in front of ALU select

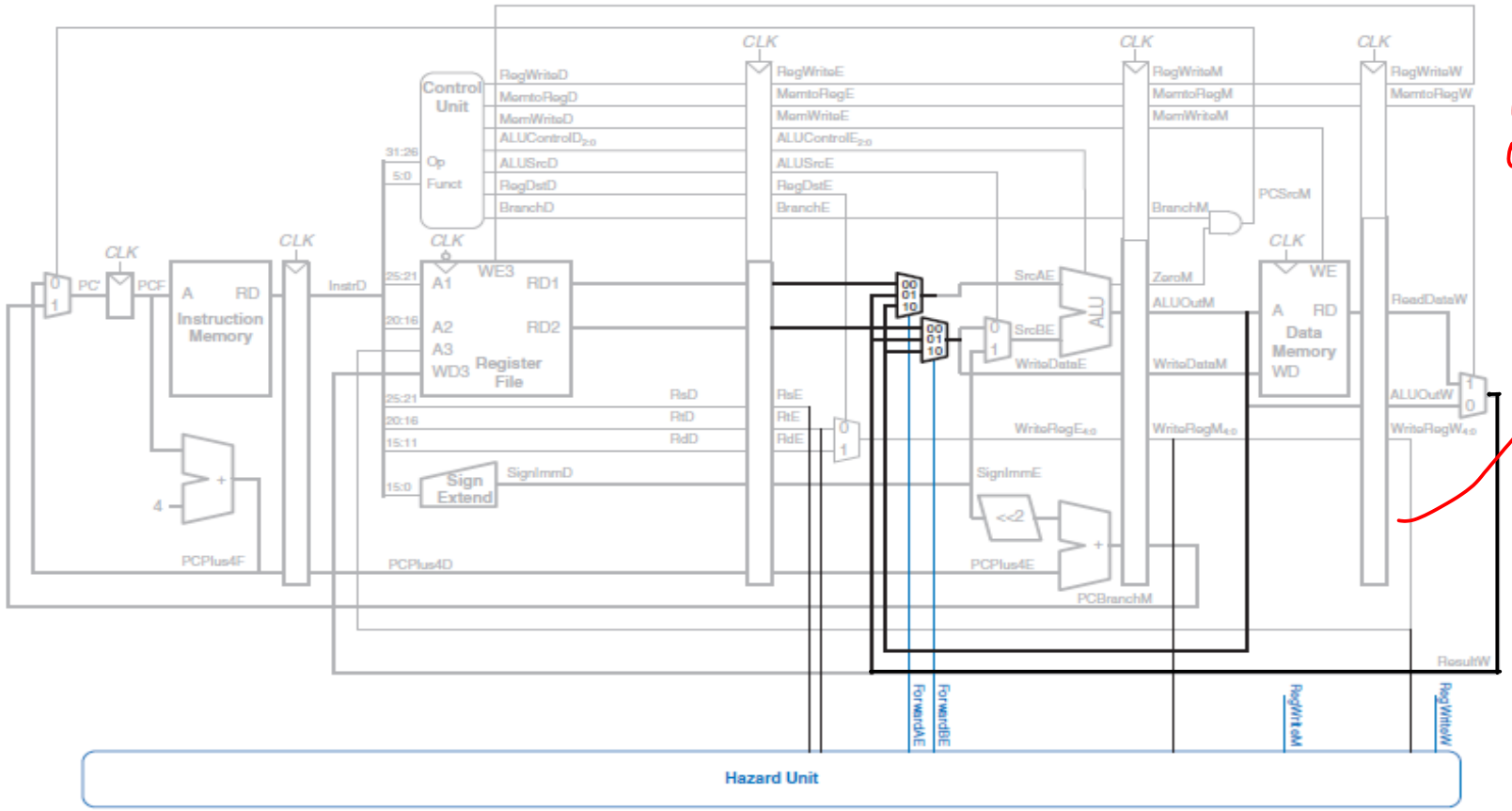
  - ALU's inputs:
    - Register file or Decode stage
    - Memory stage
    - Writeback stage

Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

  - An instruction in the Execute stage
    - A source register matching the destination register of an instruction in
      - Memory stage and/or
      - Writeback stage

Data Hazards: H/W-based Solution



00 – rs, rt
01 – ALU
10 – From this latch.

For both the source registers rs and rt, they have been muxed with the ALU output.

Data Hazards: H/W-based Solution

- Hazard unit generates control signals for
  - Mux SrcA [ForwardAE]
  - Mux SrcB [ForwardBE]
- The control logic for ForwardAE
- if ((rsE!=0) AND (rsE == WriteRegM) and RegWriteM) then
  ForwardAE = 10
- else if ((rsE!=0) AND (rsE == WriteRegW) and RegWriteW) then
  ForwardAE = 01
- else
  ForwardAE = 00

Write Reg is register to be written in register file (see in latch of MEM stage)

See in the latch storing info for MEM stage

ALUOut goes in

See in latch of write stage

Output from Data Memory goes in.

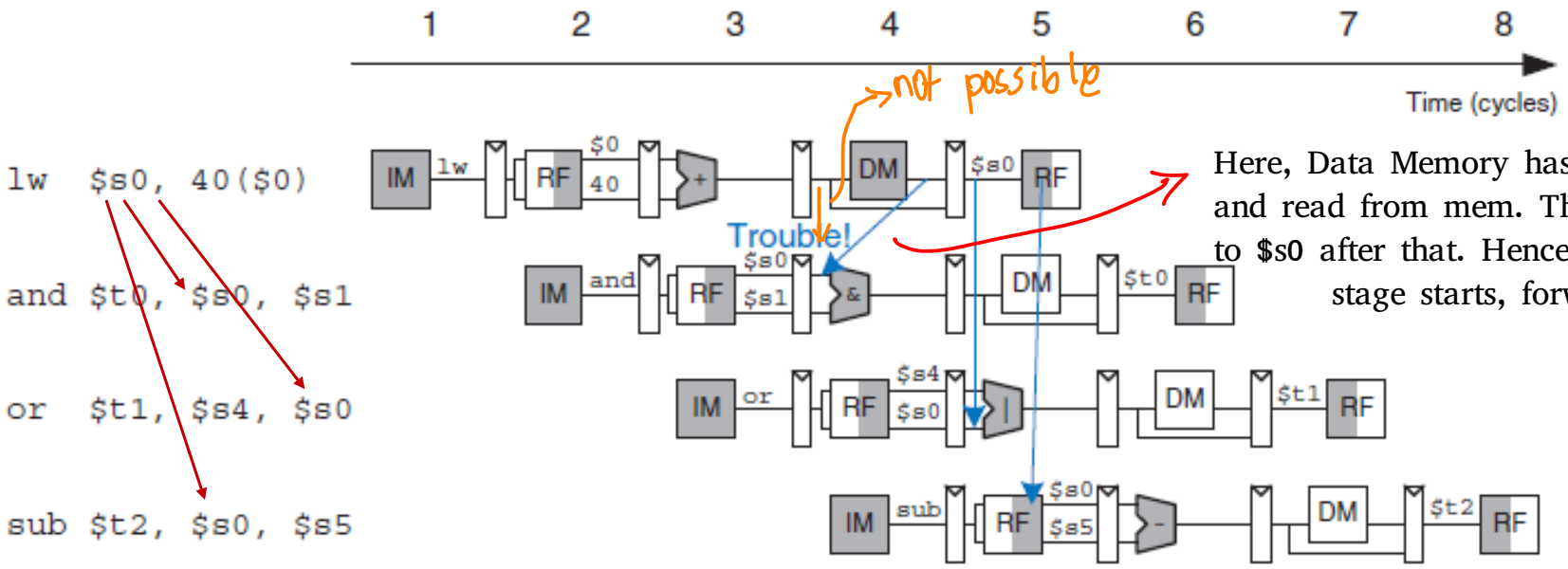
rs, rt ftw.

Data Hazards: H/W-based Solution

- Hazard unit generates control signals for
  - Mux SrcA [ForwardAE]
  - Mux SrcB [ForwardBE]
- The control logic ForwardBE (same as in ForwardAE except rt, instead of rs)
- if ((rtE!=0) AND (rtE == WriteRegM) and RegWriteM) then
  ForwardBE = 10
- else if ((rtE!=0) AND (rtE == WriteRegW) and RegWriteW) then
  ForwardBE = 01
- else
  ForwardBE = 00

Data Hazards

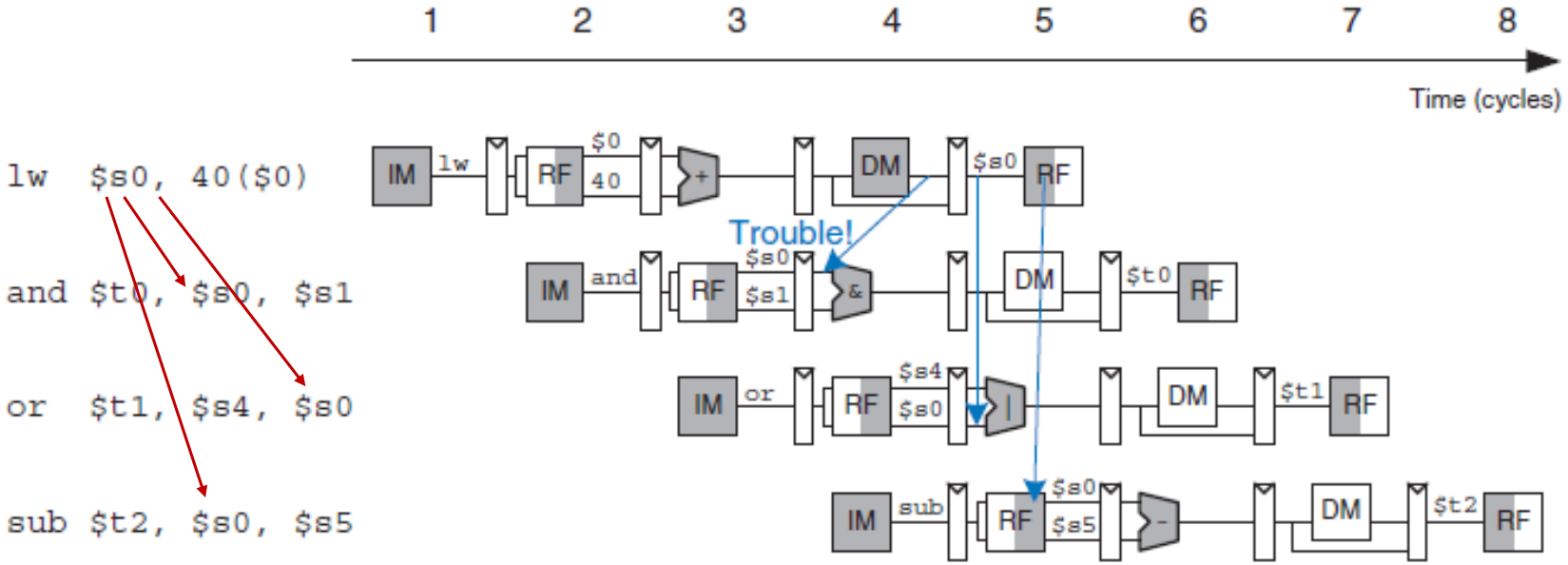
- Can we solve this one by forwarding?
- Is lw-instruction only causes this?



Here, Data Memory has to process the address and read from mem. This value is written to $s0 after that. Hence, as soon as the MEM stage starts, forwarding can't be done.

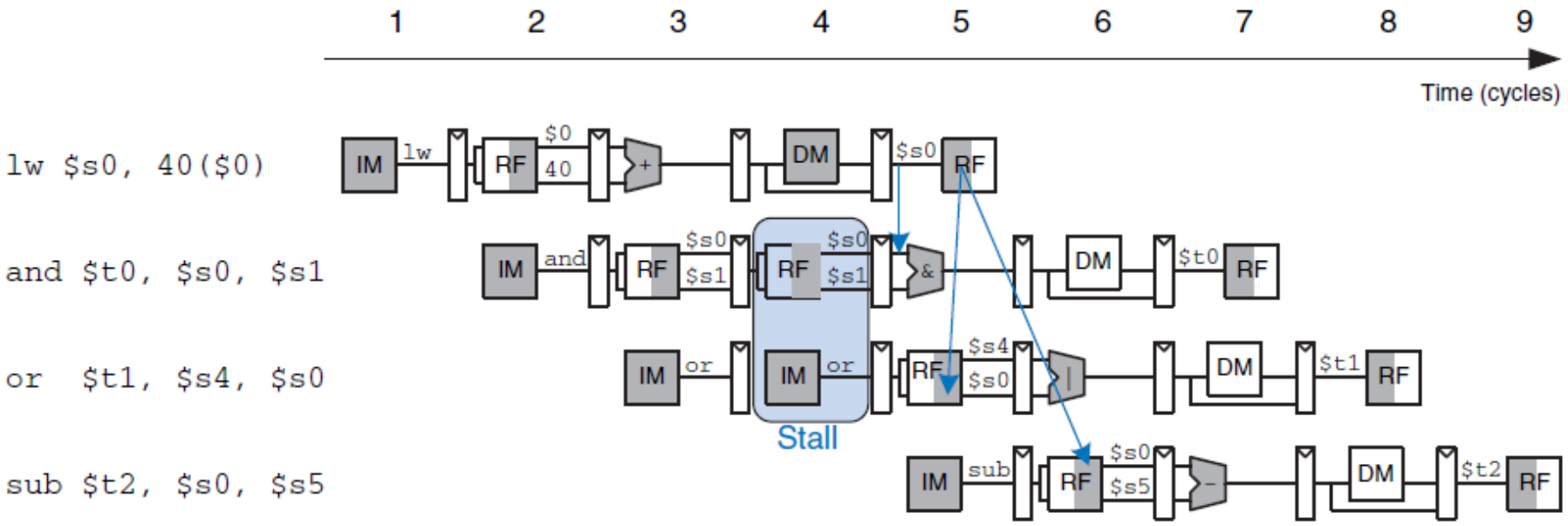
Data Hazards

- Can we solve this one by forwarding? [No]
- How does one solve it?



Data Hazards

- Forwarding with pipeline interlocking (stalling)



Have to stall all the following instructions. For this software implementation is NOP

Data Hazards

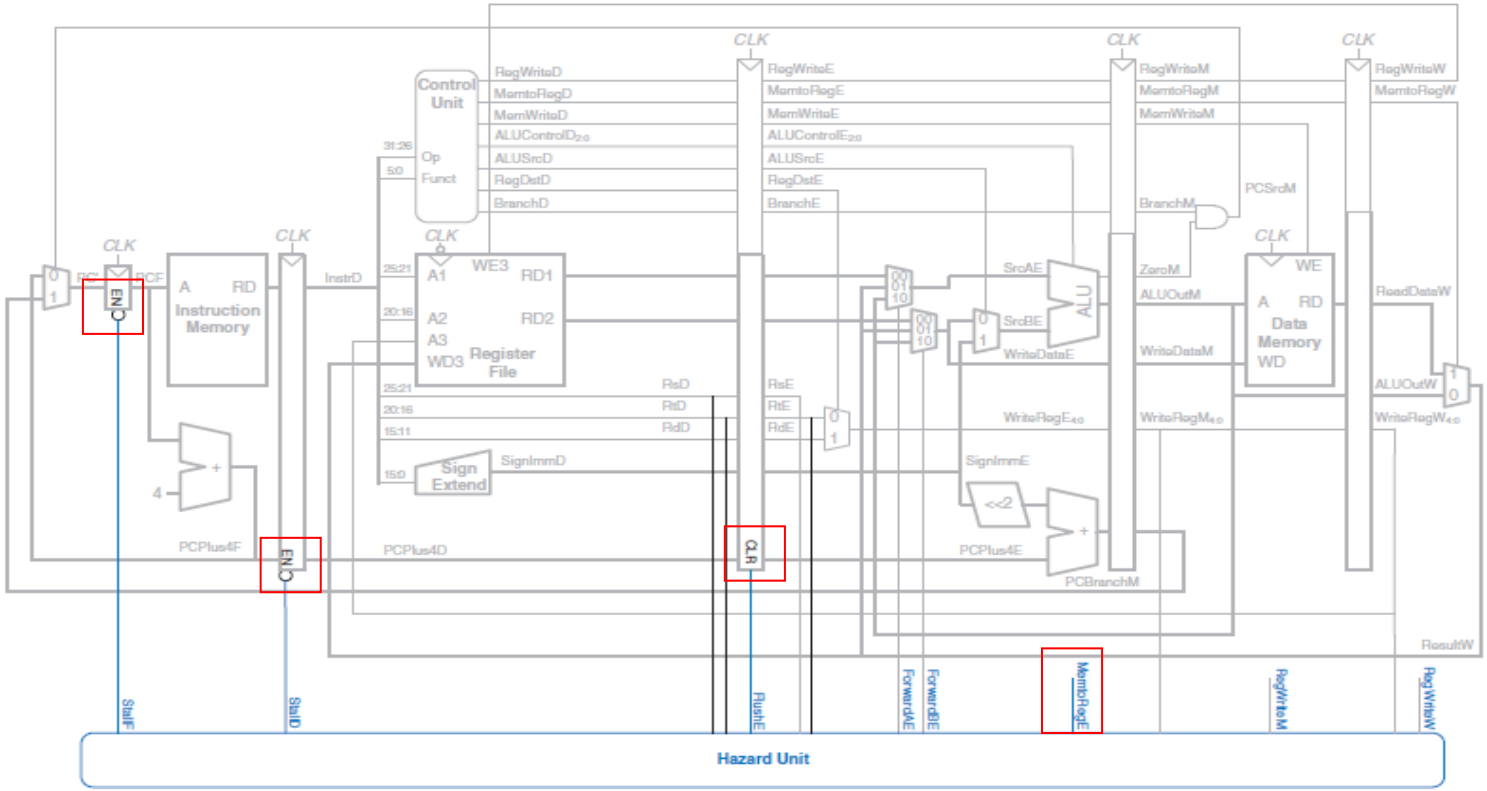
- Forwarding with pipeline interlocking (stalling)
- Hazard unit checks
  - Is it the lw-instruction?
    - Destination register (rtE) matches with
    - Source operand in the Decode stage (rsD or rtD)
    - Stall the Decode stage (how long?)
- How does one perform stall operation?

Data Hazards

- How does one perform stall operation?
  - Incorporate the Enable (EN) control signal
    - Fetch pipeline register
    - Decode pipeline register
  - Incorporate a synchronous reset/clear (CLR)
    - Execute pipeline register

Data Hazards

- Forwarding with pipeline interlocking (stalling)



Data Hazards

- Control logic for forwarding with pipeline interlocking (stalling)
  - If lw-instruction
    - Asserted MemtoRegE
  - lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
  - StallF = lwstall Switch off the program counter
  - StallD = lwstall Switch off the latch after instruction stage
  - FlushE = lwstall Clear the values in the latch after decode stage

Note that rt value provides the reg no. used by lw.

Data Hazards

- What if hazard unit unable to detect

Data Hazards

- What if hazard unit unable to detect
  - Compiler has to control
    - noop-instruction
    - Rearrange the program code (instruction scheduling or pipeline scheduling)

Data Hazards

instruction scheduling or *pipeline scheduling*.

Original code	Insert the NOOP instruction	After ordering the code
LW R1, 40 ($40)	LW R1, 40 ($0)	LW R1, 40 ($0)
ADDI R1, R1, 5	NOOP	LW R2, 44 ($0)
SW R1, 50 ($0)	NOOP	NOOP
LW R2, 44 ($0)	ADDI R1, R1, 5	ADDI R1, R1, 5
ADD R1, R2, R3	SW R1, 50 ($0)	SW R1, 50 ($0)
SW R2, 54 ($0)	LW R2, 44 ($0)	ADD R1, R2, R3
	NOOP	SW R3, 54($0)
	NOOP	
	ADD R1, R2, R3	
	SW R3, 54 ($0)	

How does one decide no. of consecutive NOOP instruction to put after an instruction? It depends on delay (clock cycle) to produce the correct operand for the dependent instruction(s).

Summary

- Types of hazards
- Minimization of structural hazards with
  - Increased resources
  - Interlocking technique
- Types of data hazards
- Minimization of data hazards with
  - Forwarding technique
  - Forwarding with Interlocking technique
  - Compiler-based technique

Computer Architecture (CS F342)

Design and Analysis of Instructions

Minimization of Structural & Data Hazards in Pipelined MIPS (RISC) Processor

Pipelined-based Processor

- Pipelining technique exploits parallelism
  - How?
- Can it faces any difficulties while doing so?

Hazards in Pipelined-based Processor

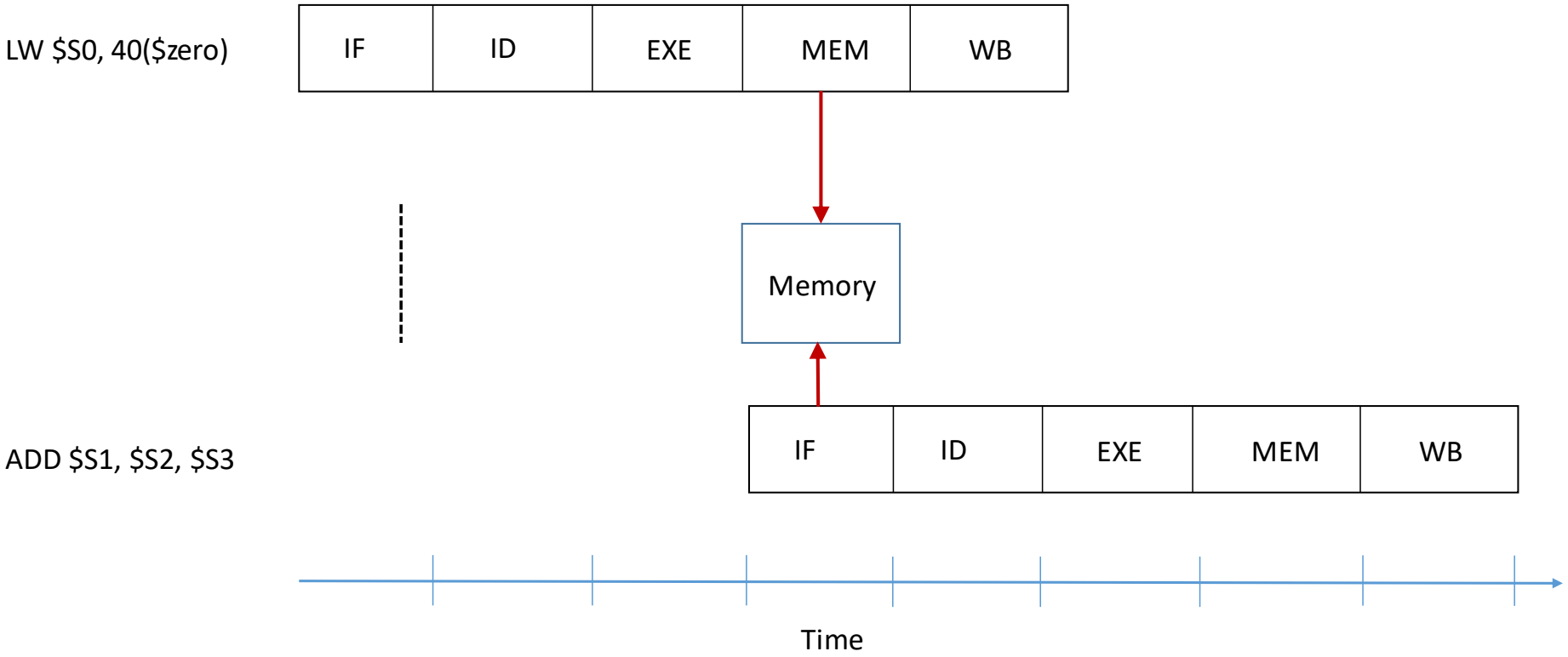
- Conditions that make pipeline to stall: Hazard
  - Structural Hazard
  - Data Hazard
  - Control Hazard

Structural Hazards

- Architectural component does not support parallelism, if we assume
  - Single memory unit
  - More than one instructions trying to write data in register file at the same time

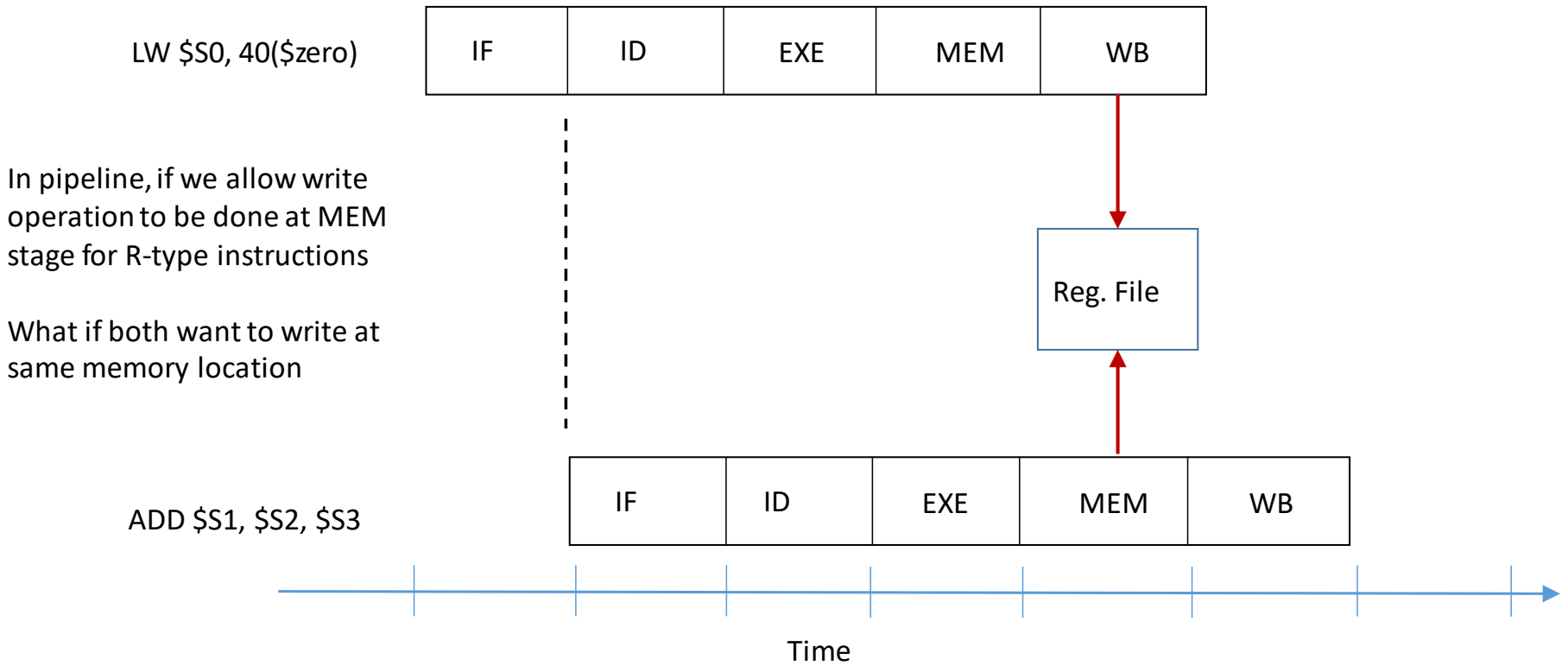
Structural Hazards

- Single memory unit



Structural Hazards

- Two instructions trying to write data in register file at the same time



Solution for Structural Hazard

- Incorporate more resources
- Stall the operation
  - arbitration with interlocking

Dependency between instructions/data

- What are the possible dependencies between two instructions: $Inst_1$ & $Inst_2$?

Dependency between instructions/data

- Assume: $Inst_1$ fetched prior to $Inst_2$
  - $Inst_2$ is data dependent on $Inst_1$
    - if $Inst_1$ writes its output in a register, Reg (or memory location)
    - $Inst_2$ reads as that as its input
  - $Inst_2$ is anti-dependent on $Inst_1$
    - if $Inst_1$ reads data from a register *Reg* (or memory location) which is subsequently overwritten by $Inst_2$

Dependency between instructions/data

- Assume: $Inst_1$ fetched prior to $Inst_2$
  - $Inst_2$ is output dependent on $Inst_1$
    - if both write in the same register *Reg* (or memory location)
    - $Inst_2$ writes its output after $Inst_1$
  - $Inst_2$ is control dependent on $Inst_1$
    - if $Inst_1$ must complete before a decision can be made whether or not to execute $Inst_2$

Data Hazards

- Data dependences between instructions
  - True or real
  - False or name
- Inst1 & Inst2 are so close that their overlapping would change their access order to register, Reg.

Data Hazards

- Types of data hazards
  - Read after write (RAW)
    - caused by data dependency
  - Write after read (WAR)
    - caused by anti-dependence
  - Write after write (WAR)
    - caused by output dependence

Data Hazards

- WAW hazards occur
  - Write operation in more than one stages
  - Allow an instructions to proceed even when a previous instruction is stalled

- Will it occur in MIPS?

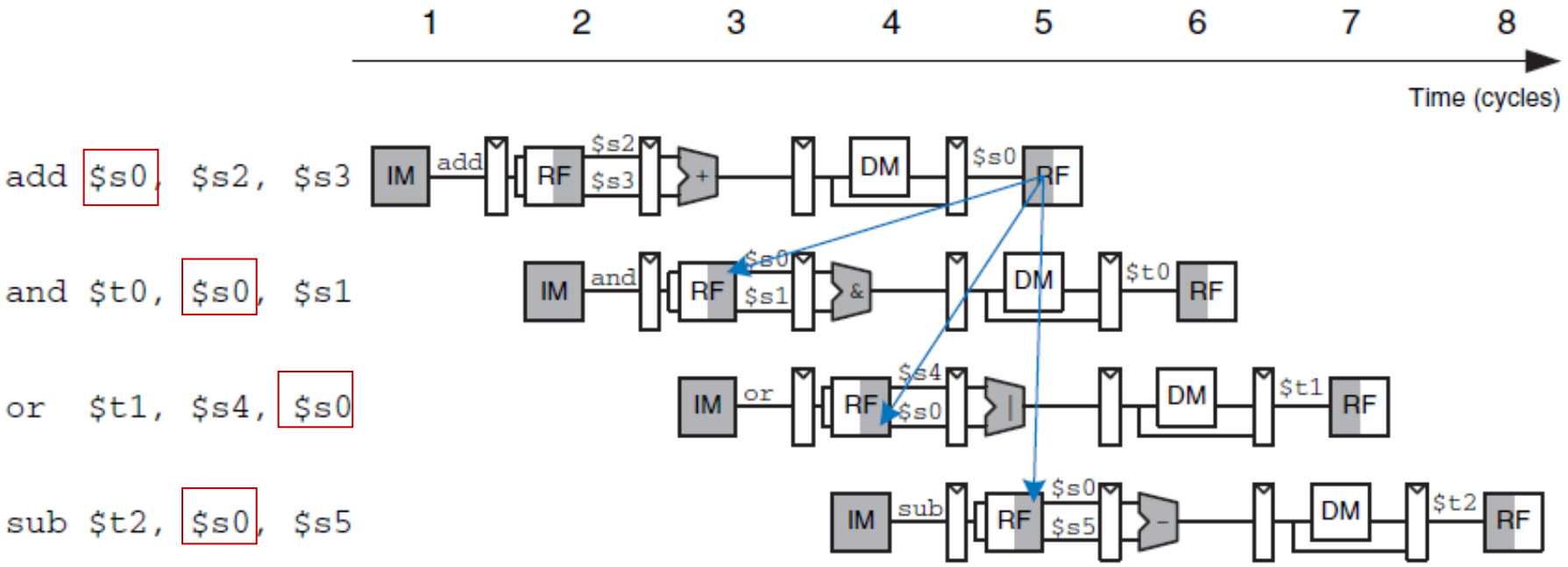
Data Hazards

- WAR hazards occur
  - Write stage precedes a read stage

- Will it occur in MIPS?

Data Hazards

- Only RAW hazard occurs in MIPS



Data Hazards

- How does one solve RAW hazards?

Data Hazards

- How does one solve RAW hazards?
  - Hardware-based solutions
  - Software-based solutions

Data Hazards: H/W-based Solution

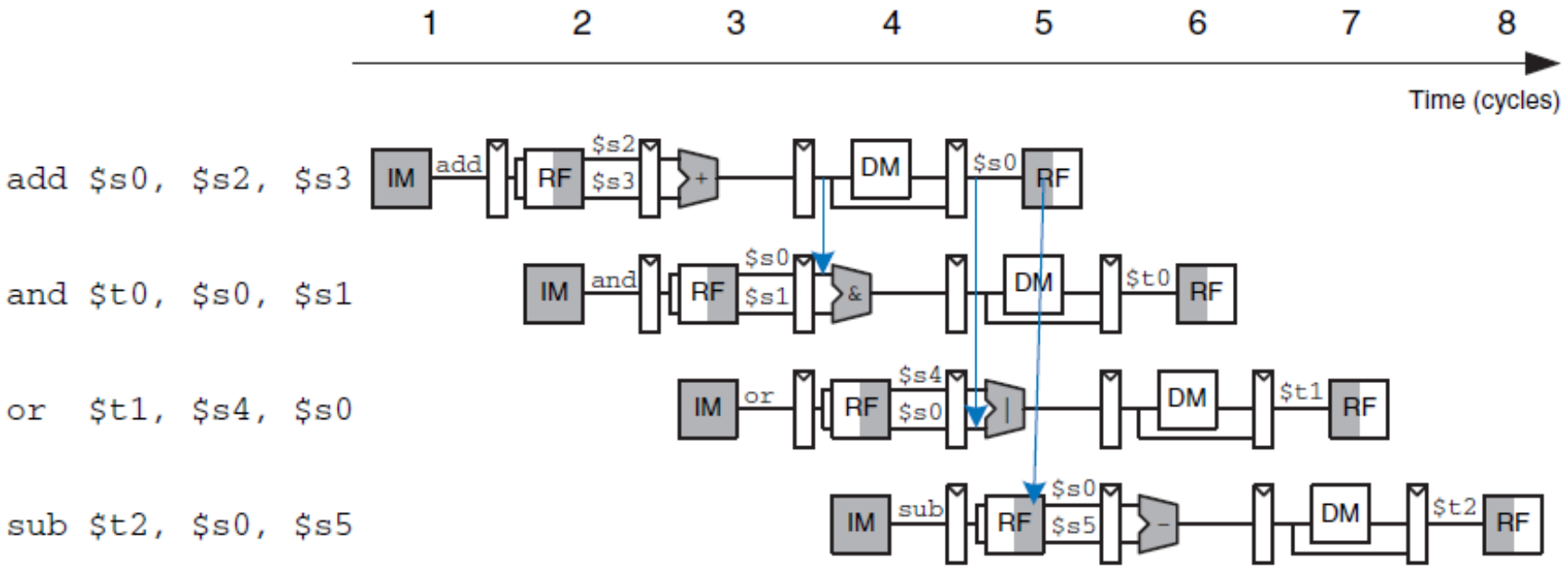
- Interlocking -- a simple solution
  - Detect the hazard
  - Stall the pipeline
- Degrade the speedup

Data Hazards: H/W-based Solution

- Forwarding – a sophisticated solution
  - The result of the ALU output of $Inst_1$ in the EX stage can immediately forward back to ALU input of EX stage as an operand for $Inst_2$

Data Hazards: H/W-based Solution

- Forwarding or bypassing



Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

  - Put MUXs in front of ALU select

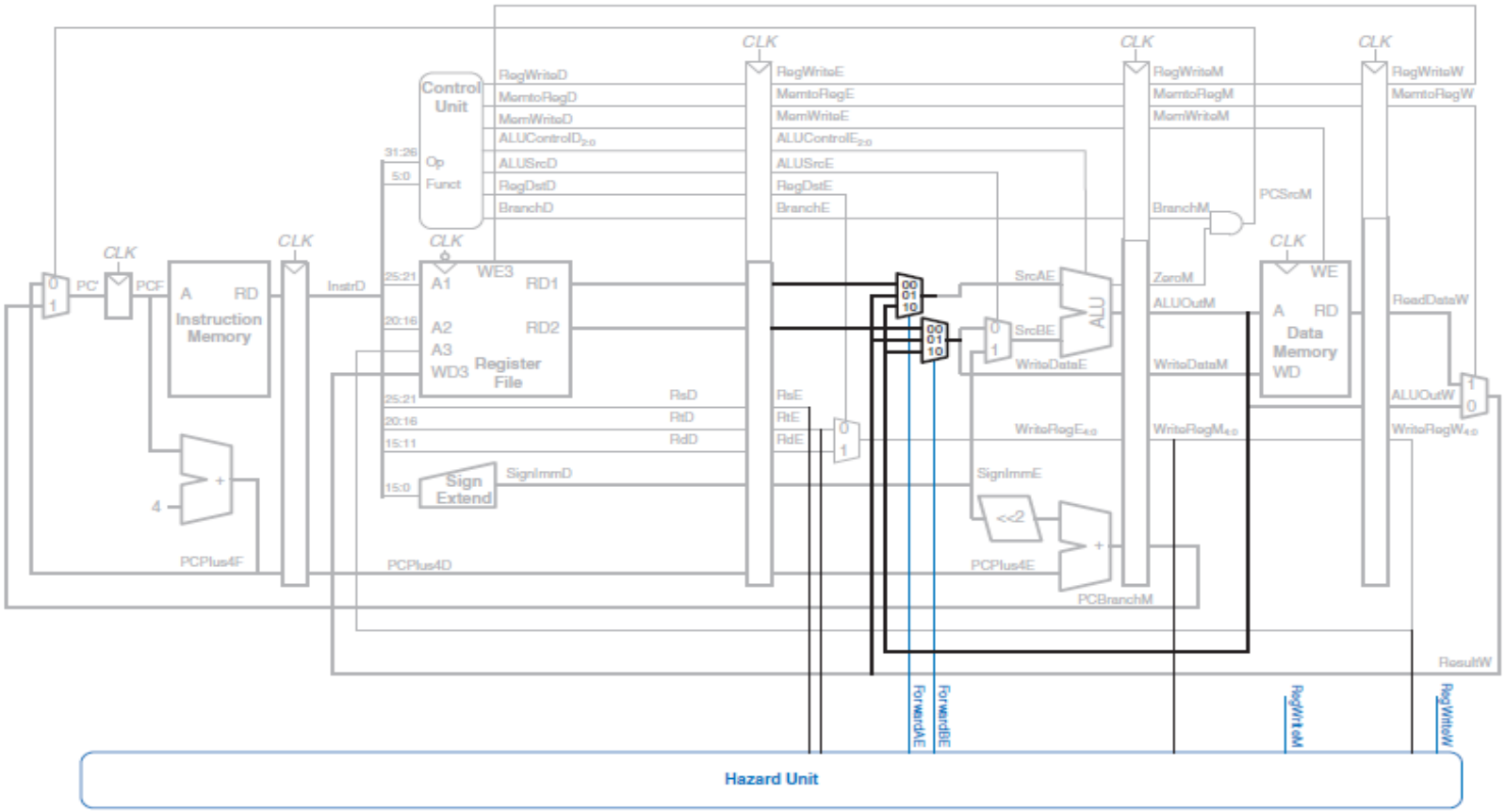
  - ALU's inputs:
    - Register file or Decode stage
    - Memory stage
    - Writeback stage

Data Hazards: H/W-based Solution

- How does one perform forwarding or bypassing?

  - An instruction in the Execute stage
    - A source register matching the destination register of an instruction in
      - Memory stage and/or
      - Writeback stage

Data Hazards: H/W-based Solution



Data Hazards: H/W-based Solution

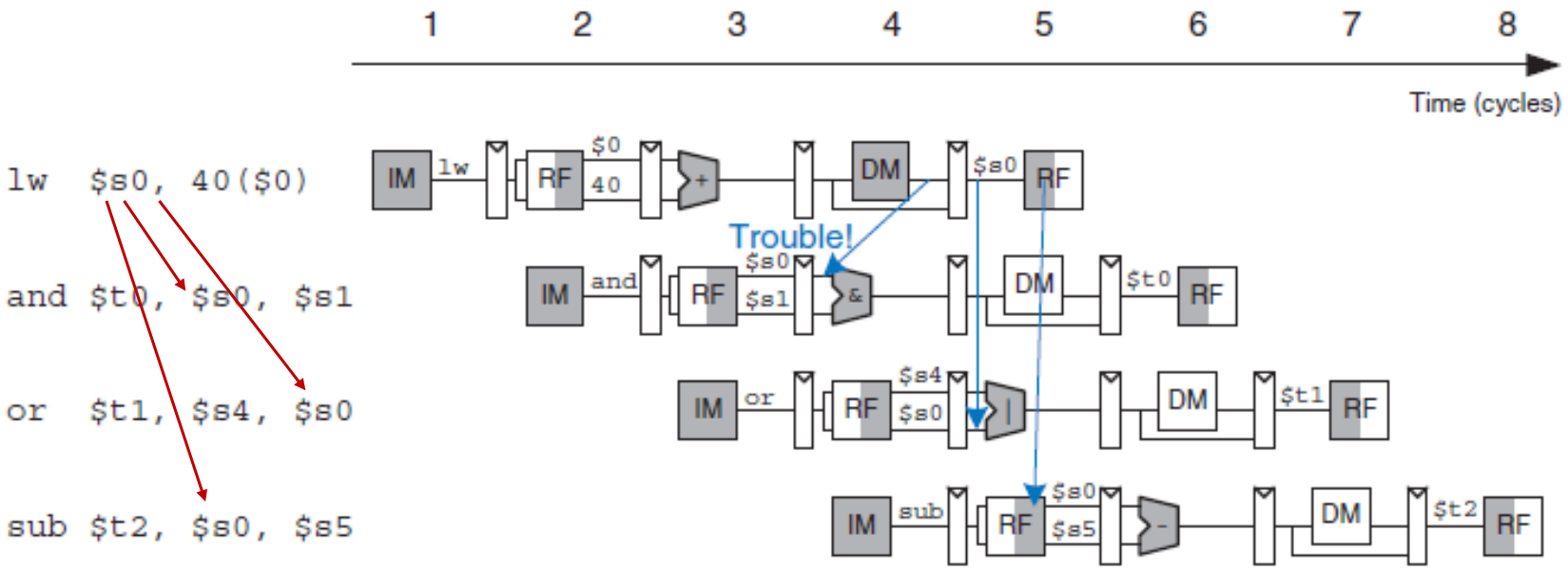
- Hazard unit generates control signals for
  - Mux SrcA [ForwardAE]
  - Mux SrcB [ForwardBE]
- The control logic for ForwardAE
- if ((rsE!=0) AND (rsE == WriteRegM) and RegWriteM) then
  ForwardAE = 10
- else if ((rsE!=0) AND (rsE == WriteRegW) and RegWriteW) then
  ForwardAE = 01
- else
  ForwardAE = 00

Data Hazards: H/W-based Solution

- Hazard unit generates control signals for
  - Mux SrcA [ForwardAE]
  - Mux SrcB [ForwardBE]
- The control logic ForwardBE (same as in ForwardAE except rt, instead of rs)
- if ((rtE!=0) AND (rtE == WriteRegM) and RegWriteM) then
  ForwardBE = 10
- else if ((rtE!=0) AND (rtE == WriteRegW) and RegWriteW) then
  ForwardBE = 01
- else
  ForwardBE = 00

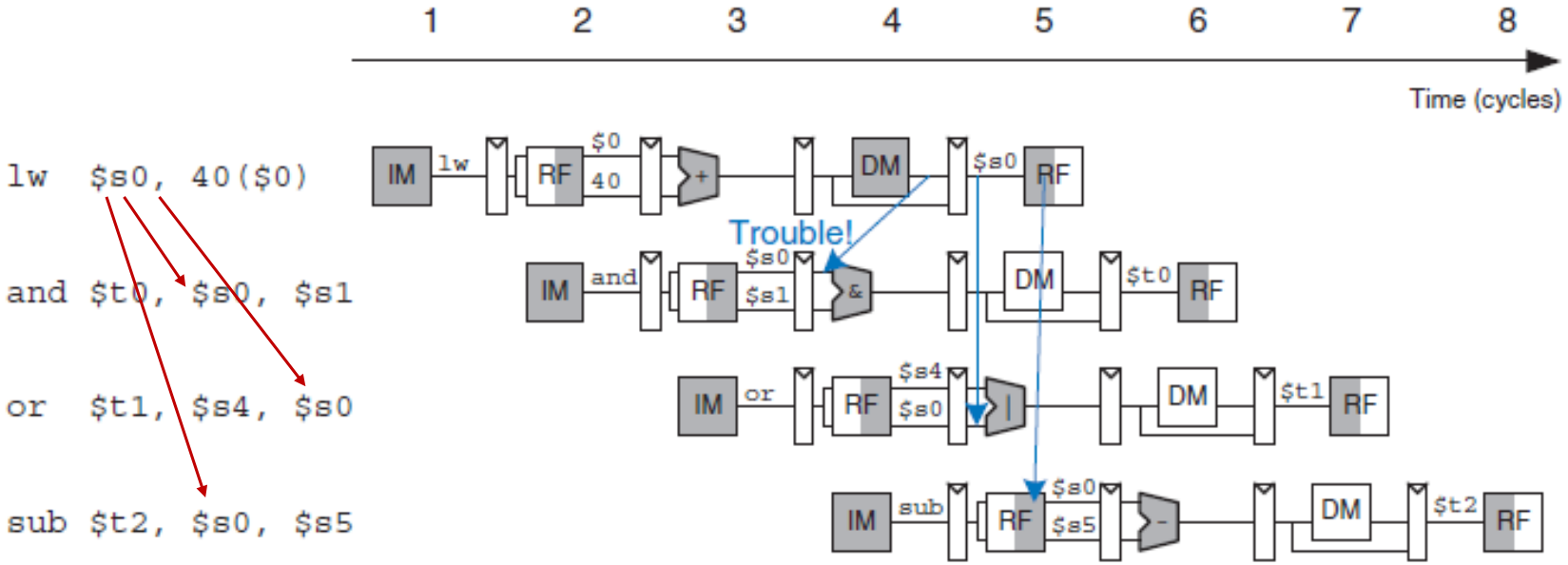
Data Hazards

- Can we solve this one by forwarding?
- Is lw-instruction only causes this?



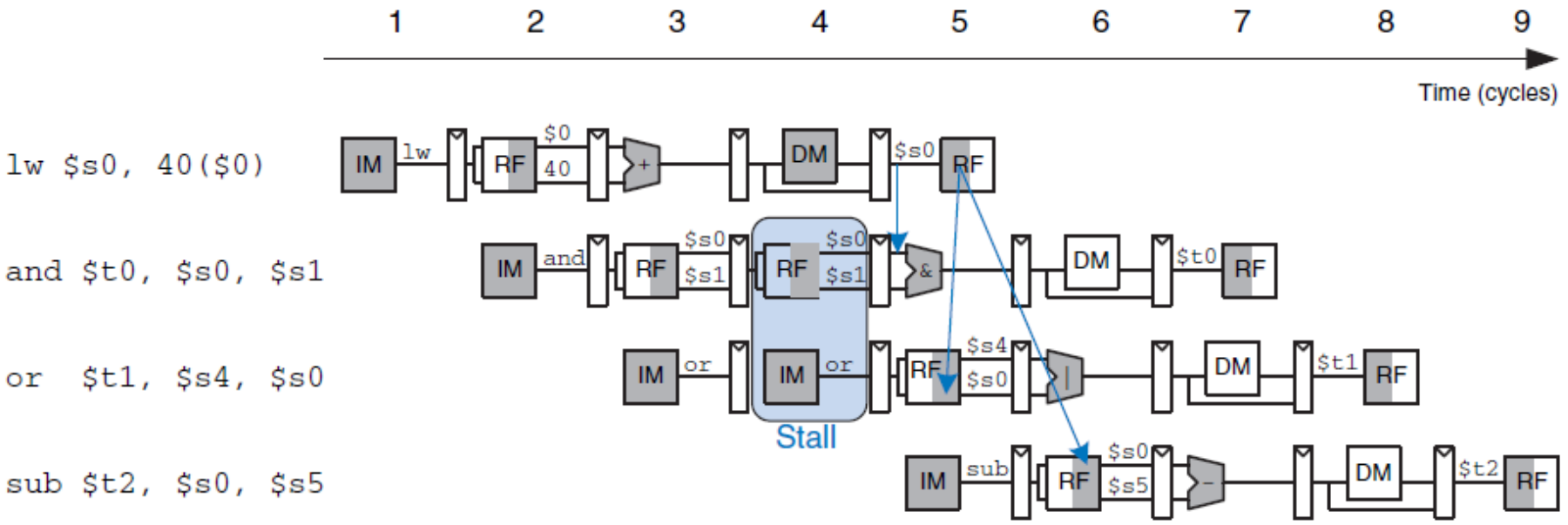
Data Hazards

- Can we solve this one by forwarding? [No]
- How does one solve it?



Data Hazards

- Forwarding with pipeline interlocking (stalling)



Data Hazards

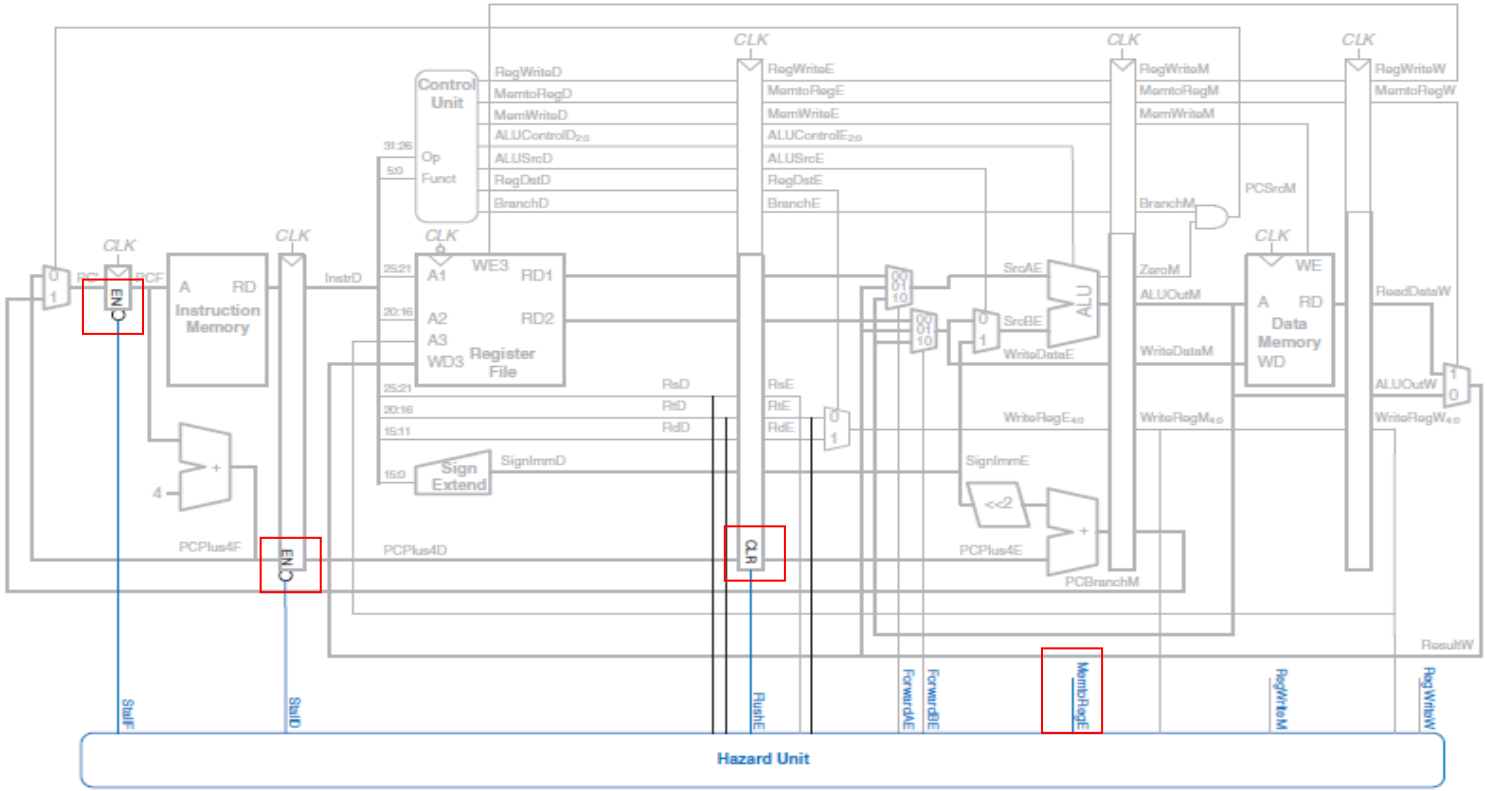
- Forwarding with pipeline interlocking (stalling)
- Hazard unit checks
  - Is it the lw-instruction?
    - Destination register (rtE) matches with
    - Source operand in the Decode stage (rsD or rtD)
    - Stall the Decode stage (how long?)
- How does one perform stall operation?

Data Hazards

- How does one perform stall operation?
  - Incorporate the Enable (EN) control signal
    - Fetch pipeline register
    - Decode pipeline register
  - Incorporate a synchronous reset/clear (CLR)
    - Execute pipeline register

Data Hazards

- Forwarding with pipeline interlocking (stalling)



Data Hazards

- Control logic for forwarding with pipeline interlocking (stalling)
  - If lw-instruction
    - Asserted MemtoRegE
  - lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
  - StallF = lwstall
  - StallD = lwstall
  - FlushE = lwstall

Performance of pipelines with stalls

- A stall causes the pipeline performance to degrade from the ideal performance (1 instr. per cycle time)

- Speedup $= \frac{Avg.\ instr.\ time\ unpipelined}{Avg.\ \ instr.\ time\ pipelined}$

$$= \frac{CPI\ unpipelined \times Clock\ cycle\ unpipelined}{CPI\ pipelined \times Clock\ cycle\ pipelined}$$

- CPI pipelined $= Ideal\ CPI + Pipeline\ stall\ clock\ cycle\ per\ instr.$

$$= 1 + pipeline\ stall\ clock\ cycle\ per\ instruction$$

- Speedup $= \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycle\ per\ instruction}$

$$= \frac{Pipeline\ depth}{1 + pipeline\ stall\ cycle\ per\ instruction}$$

Data Hazards

- What if hazard unit unable to detect

Data Hazards

- What if hazard unit unable to detect
  - Compiler has to control
    - noop-instruction
    - Rearrange the program code (instruction scheduling or pipeline scheduling)

Data Hazards

- Instruction scheduling or pipeline scheduling

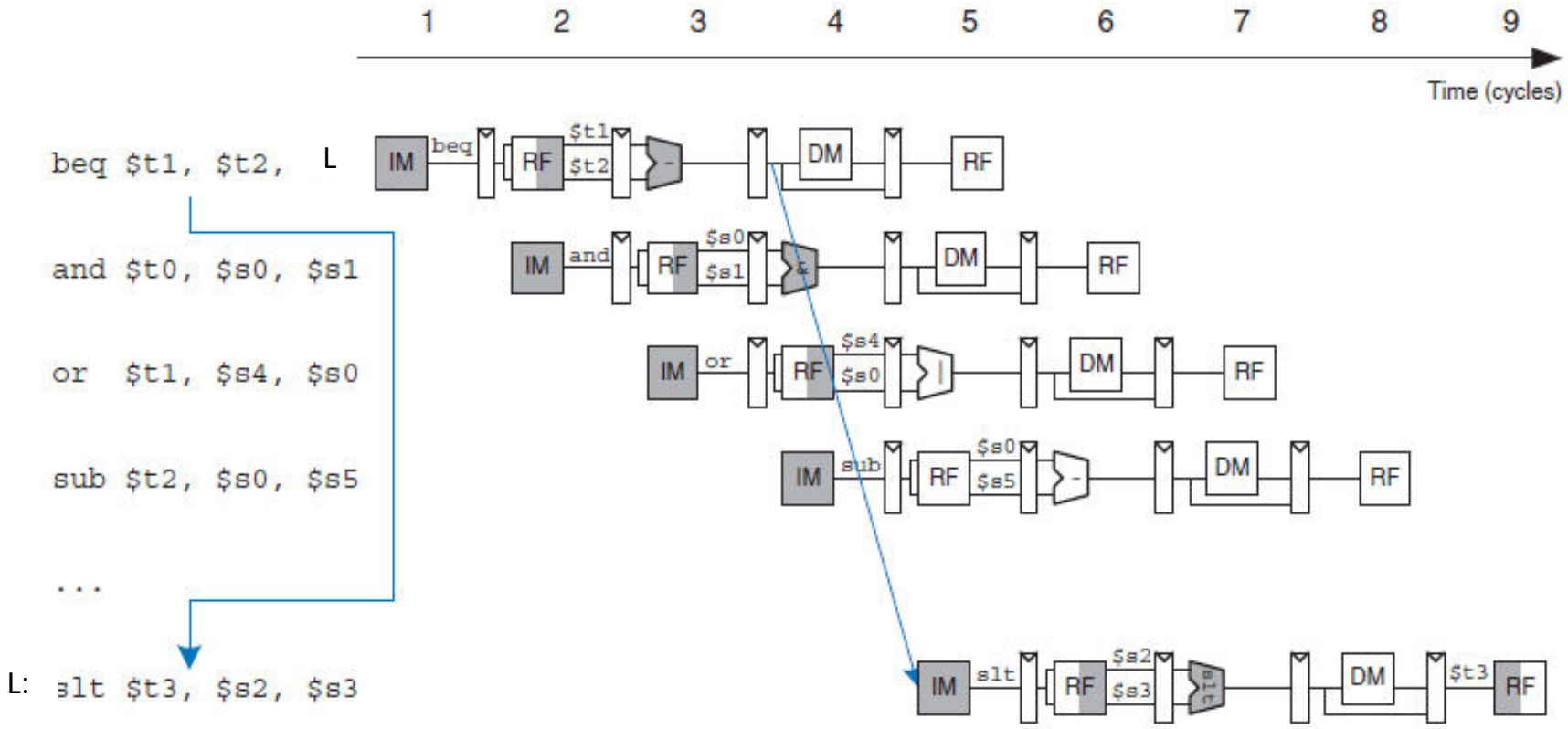
Original code	Insert the NOOP instruction	After ordering the code
LW R1, 40 ($40)	LW R1, 40 ($0)	LW R1, 40 ($0)
ADDI R1, R1, 5	NOOP	LW R2, 44 ($0)
SW R1, 50 ($0)	NOOP	NOOP
LW R2, 44 ($0)	ADDI R1, R1, 5	ADDI R1, R1, 5
ADD R1, R2, R3	SW R1, 50 ($0)	SW R1, 50 ($0)
SW R2, 54 ($0)	LW R2, 44 ($0)	ADD R1, R2, R3
	NOOP	SW R3, 54($0)
	NOOP	
	ADD R1, R2, R3	
	SW R3, 54 ($0)	

How does one decide no. of consecutive NOOP instruction to put after an instruction? It depends on delay (clock cycle) to produce the correct operand for the dependent instruction(s).

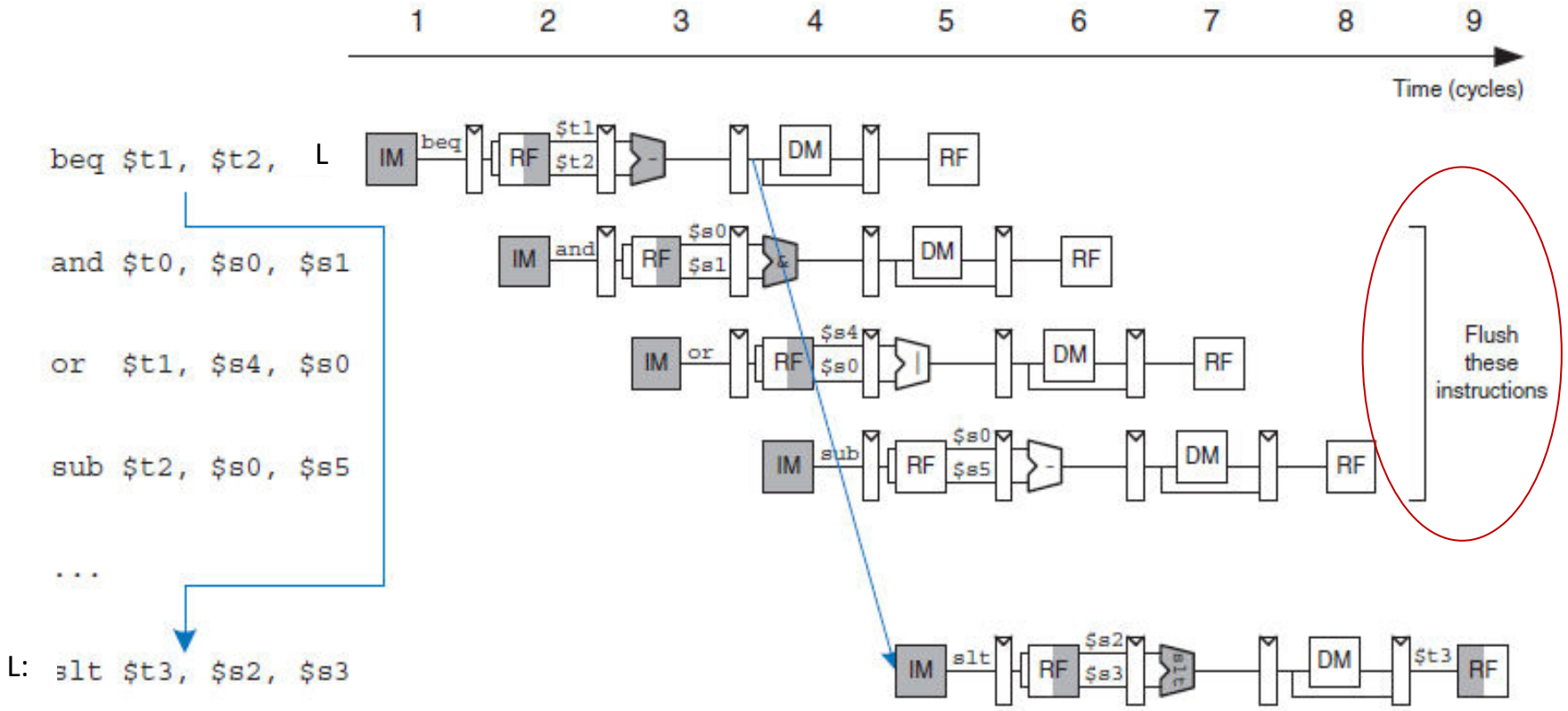
Summary

- Types of hazards
- Minimization of structural hazards with
  - Increased resources
  - Interlocking technique
- Types of data hazards
- Minimization of data hazards with
  - Forwarding technique
  - Forwarding with interlocking technique
  - Performance analysis
  - Compiler-based technique

What is this?


1
2
3
4
5
6
7
8
9
Time (cycles)
beq $t1, $t2, L
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
...
L: slt $t3, $s2, $s3
IM
beq
RF
$t1
$t2
DM
RF
IM
and
RF
$s0
$s1
DM
RF
IM
or
RF
$s4
$s0
DM
RF
IM
sub
RF
$s0
$s5
DM
RF
IM
slt
RF
$s2
$s3
DM
$t3
RF

An example of Control Hazards



Control Hazards

- Branch target address is computed only at the end of IE stage
- Cause higher performance penalty as compared to data hazards
- How does one reduce such penalty?

Control Hazards

- What if branch target address is computed at the end of ID stage

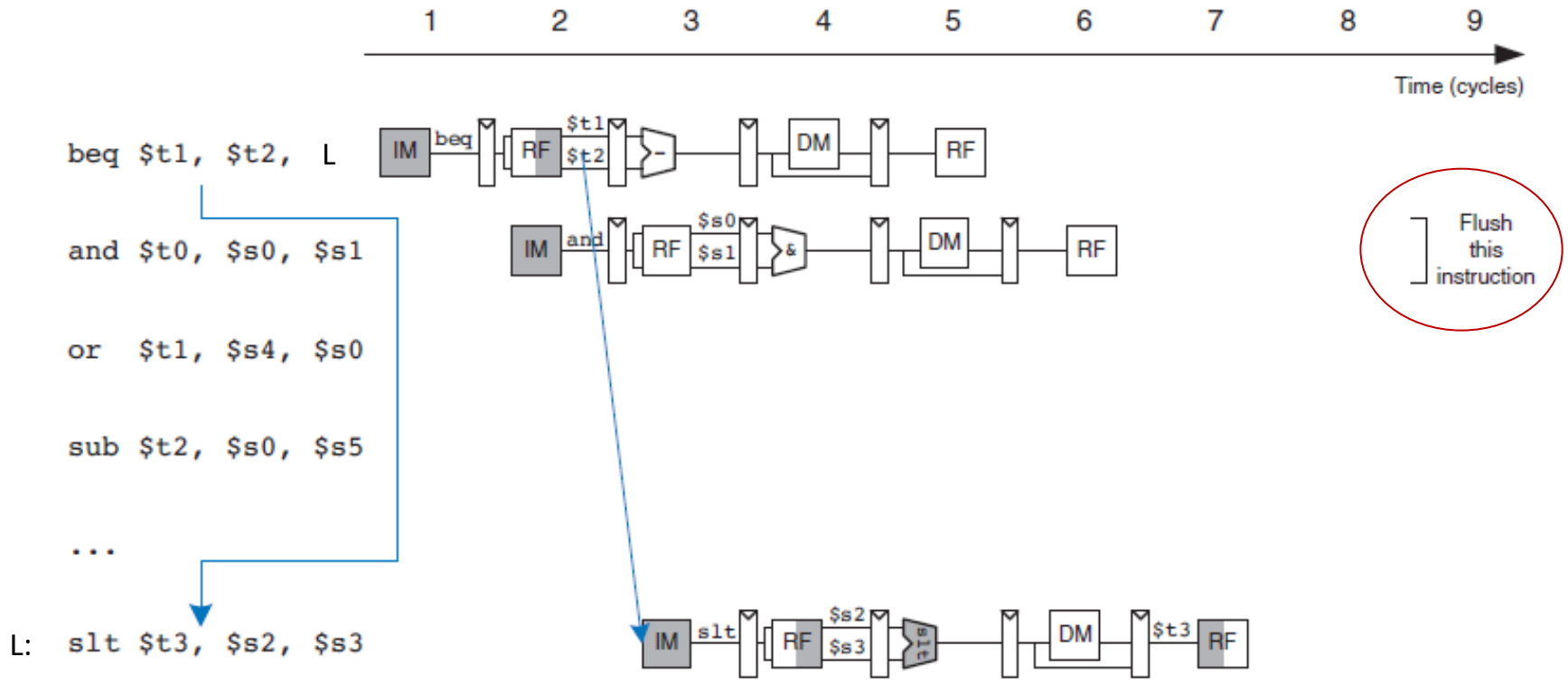
Branch instr.	IF	ID	EXE	MEM	WB		
Branch succ.		**IF**	IF	ID	EXE	MEM	WB
Branch succ + 1				IF	ID	EXE	MEM
Branch succ + 2					IF	ID	EX

$$\text{Speedup} = \frac{CPI\ unpipelined}{1+pipeline\ stall\ cycle\ from\ branchs}$$

$$= \frac{Pipeline\ depth}{1+Branch\ frequency \times Branch\ penalty}$$

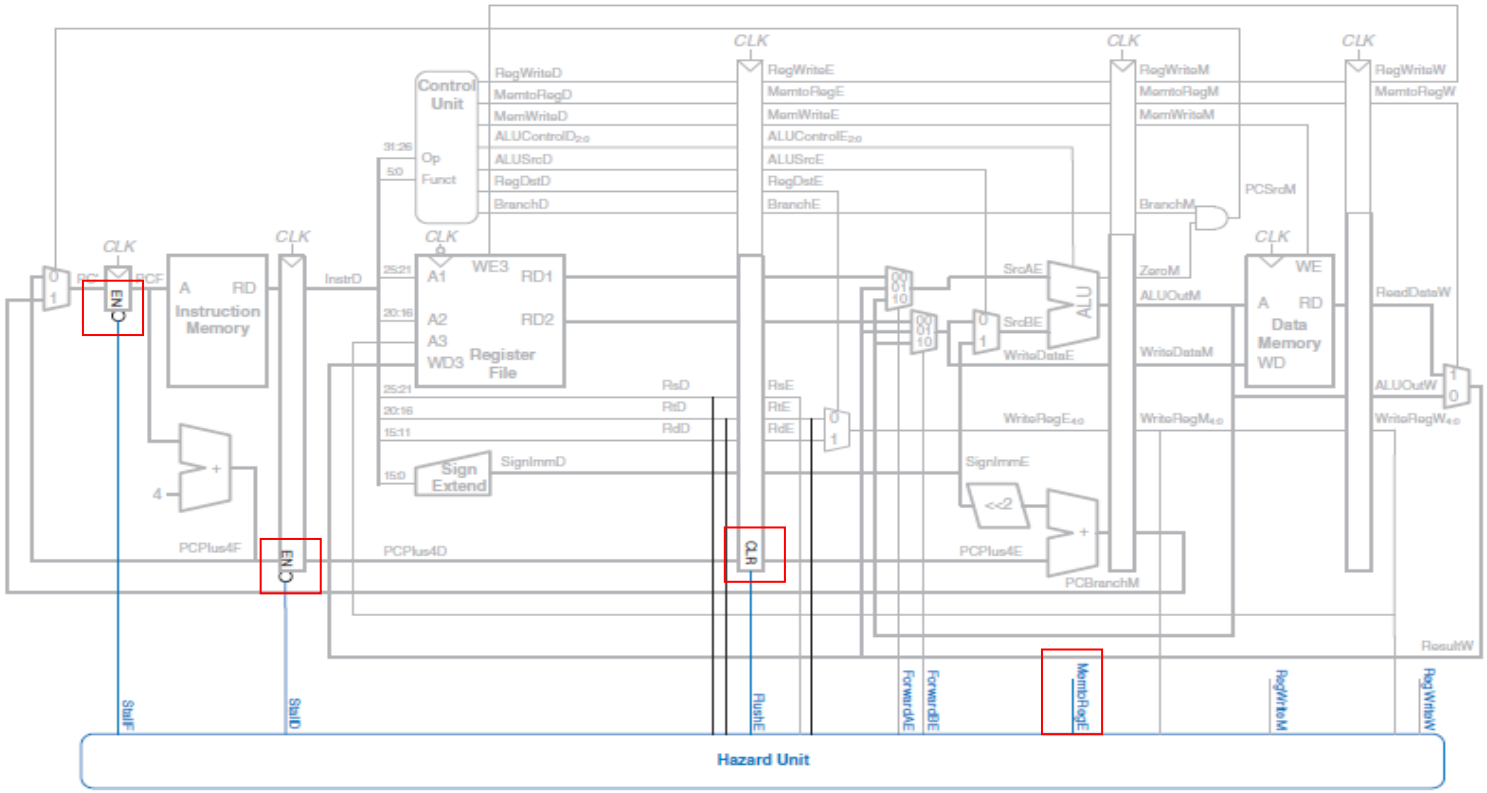
Control Hazards

- Branch target address is computed at the end of ID stage

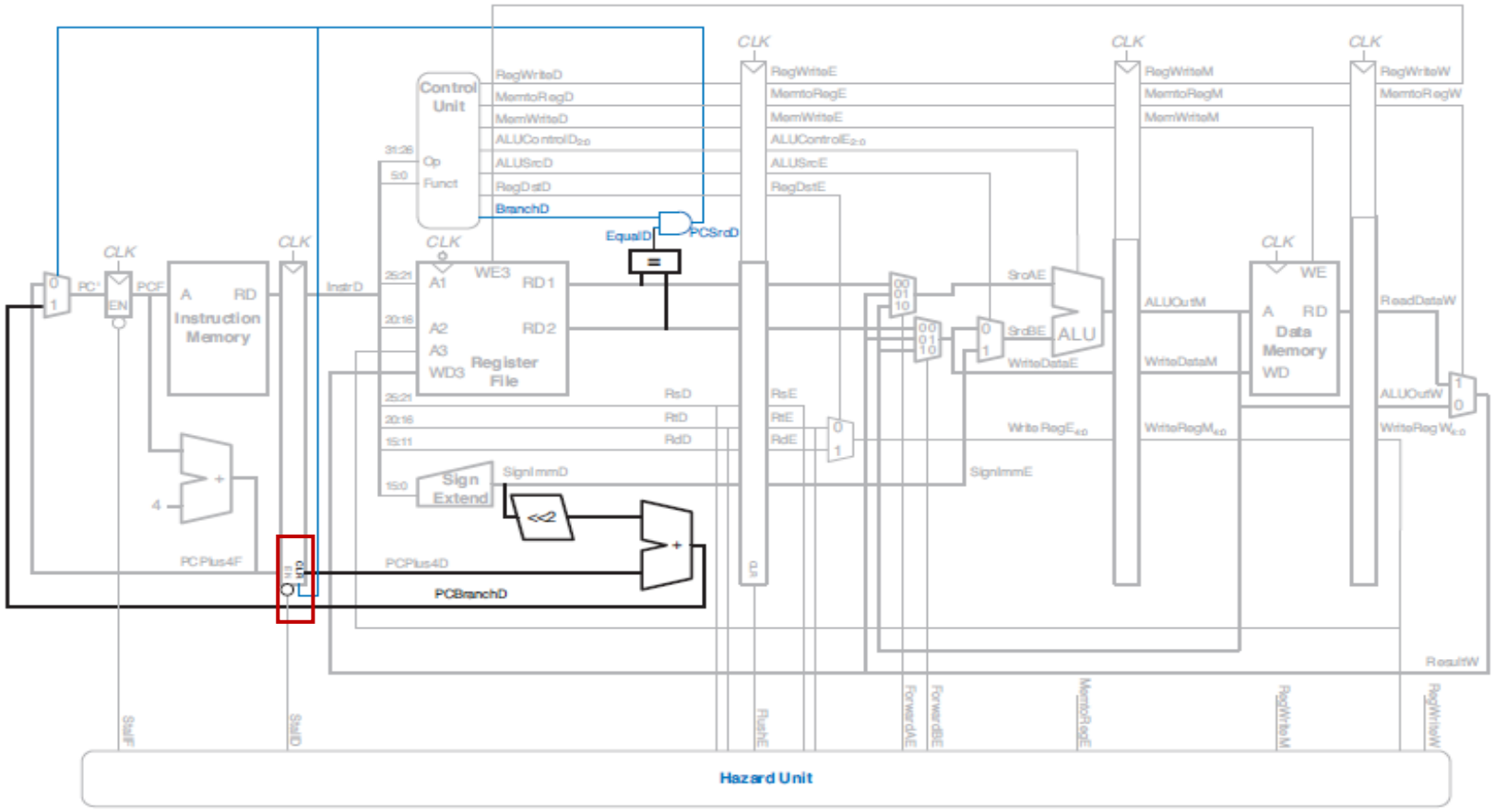


Control Hazards

- Branch target address is computed at the end of ID stage
- What modifications are needed in the pipelined datapath & controlpath?



Control Hazards: Modified Pipeline

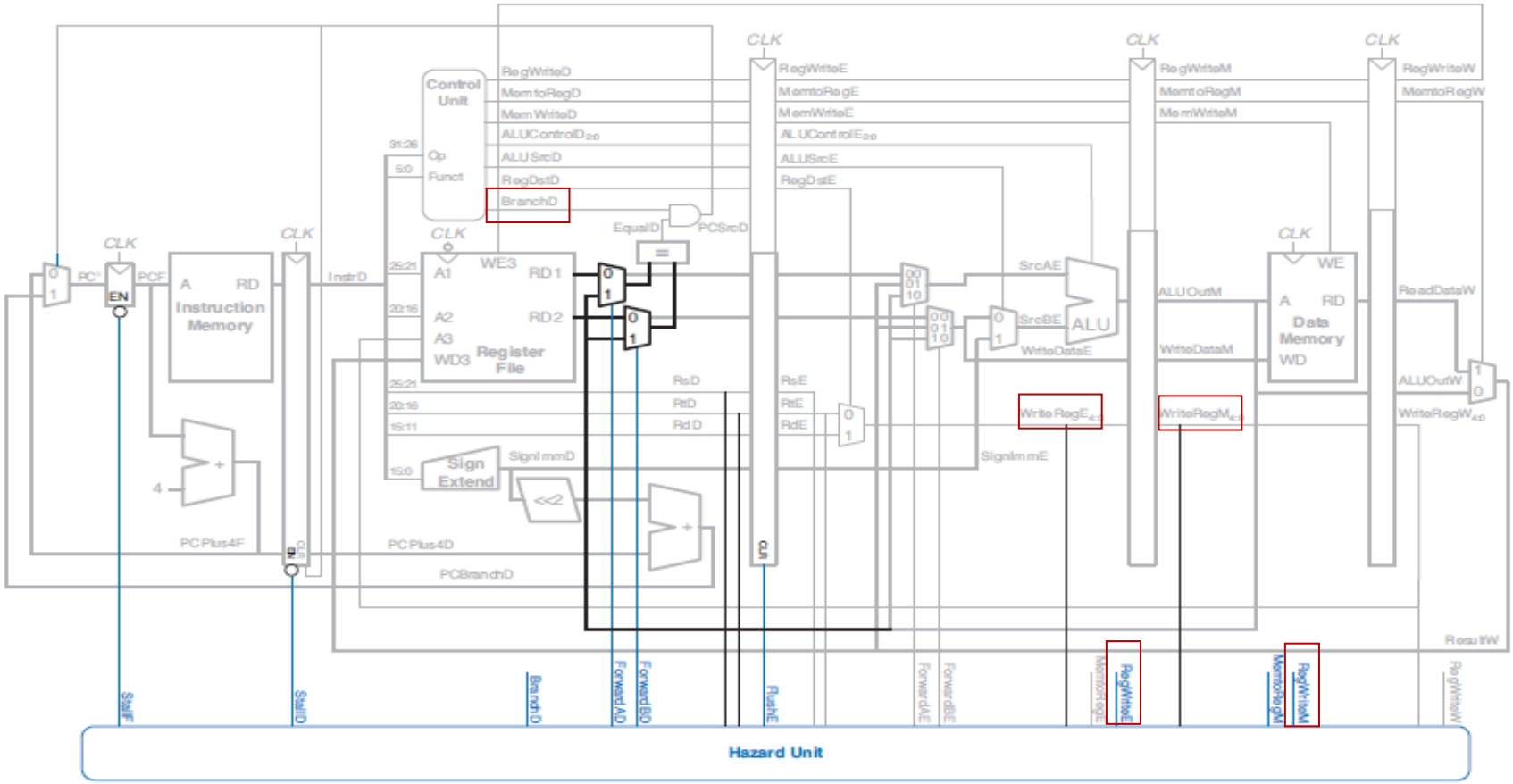


Control Hazards: Modified Pipeline

- What if one source operand of branch instruction was computed by a previous instruction and has not yet been updated into register file?

- One can use forwarding techniques
- Stalling technique can be used (lw-type)

Control Hazards & H/W-based solutions



Control Hazards & H/W-based solutions

- The function of the decode stage & forwarding logic

- FowardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
- FowardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM

Control Hazards & H/W-based solutions

- The function of the stall detection logic for a branch instruction
  - What if one of the source operand is in Execute stage (R-type instr.)
  - What if one of the source operand is in Memory stage (lw-type instr.)
- Branchstall=

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)

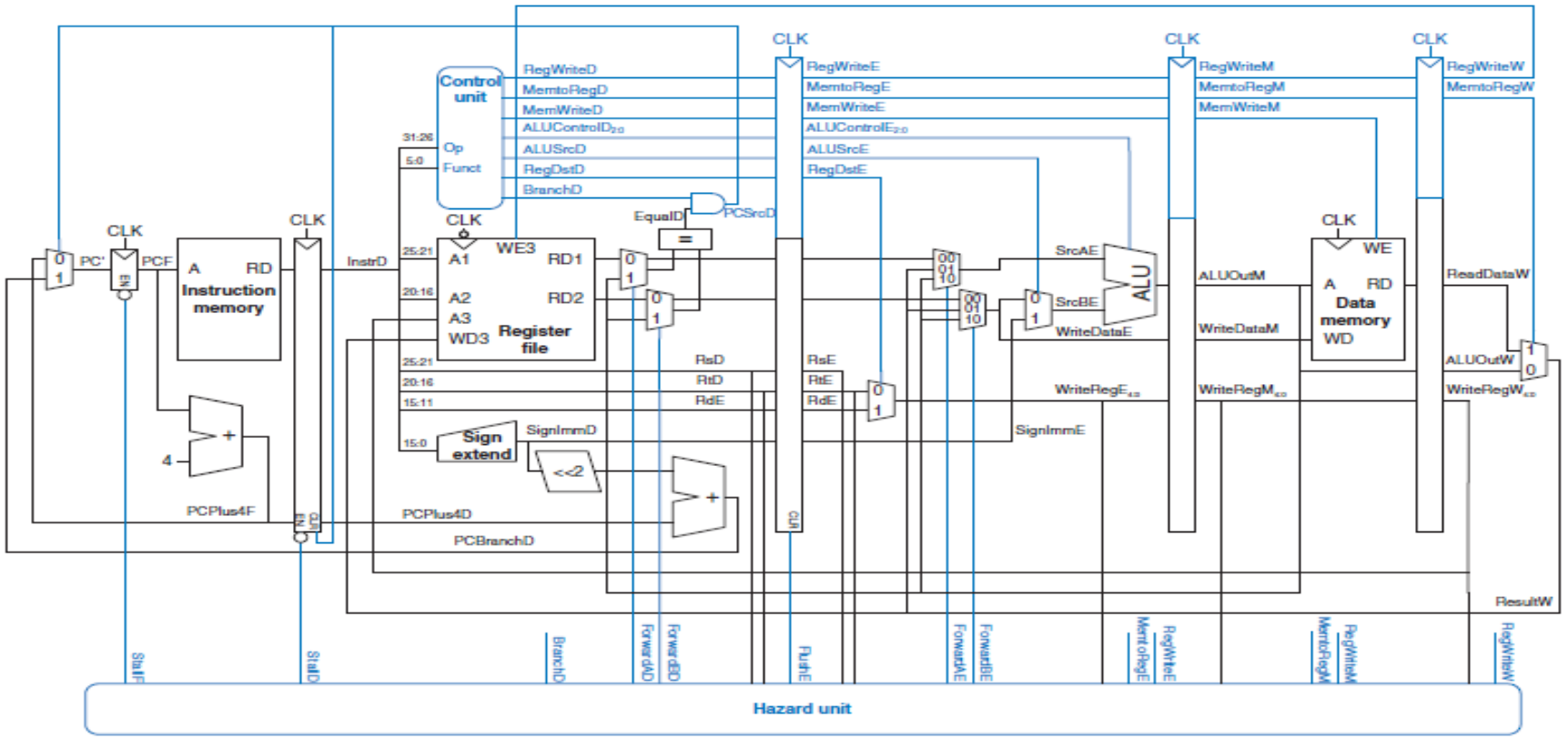
OR

BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

Control Hazards & H/W-based solutions

- Now the pipelined processor stall due to either a load or a branch hazard
- StallF = StallD = FlushE = lwstall OR branchstall

Pipelined processor with full hazard handling unit



Pipelined processor with full hazard handling unit

- Determine the cycle time
  - consider the critical path
  - five pipeline stages
- Register file is written in the first half of the Writeback cycle and read in the second half of the Decode cycle
- The cycle time of the Decode and Writeback stages is twice the time necessary to do the half-cycle of work
- Cycle period, $T_c = max \begin{pmatrix} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + T_{mux} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFWrite}) \end{pmatrix}$

Pipelined processor with full hazard handling unit

- XYZ needs to compare the pipelined processor performance to that of the single-cycle and multicycle processors considered in earlier. Most of the logic delays is given in Table. The other element delays are 40 ps for an equality comparator, 15 ps for an AND gate, 100 ps for a register file write, and 220 ps for a memory write. Help XYZ compare the execution time of 100 billion instructions of the program for each processor.

Parameter	Delay (ps)
t_{pcq}	30
t_{mem}	250
t_{RFread}	20
t_{ALU}	200
t_{mux}	25
$t_{RFwrite}$	20
t_{Setup}	20

Pipelined processor with full hazard handling unit

- According to Equation on slide 15, the cycle time of the pipelined processor is

  T_{c3} = max[30 + 250 + 20, 2(150 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 +20, 30 + 220 + 20, 2(30 + 25 + 100)] = 550 ps.

- According to Equation of CPI, the total execution time is $T_3 = (100 \times 10^9$ instructions)(1.15 cycles/instruction) $(550 \times 10^{-12}$ s / cycle) = 63.3 seconds.
- For the single-cycle processor it is 92.5 seconds and 133.9 seconds for the multicycle processor.
- What can be observed?

Control Hazards & Software-based solutions

- Assumption: predict-not-taken or predict-untaken
- Where does this assumption come from?
- Compiler rearranges the code
- Control flow will change only when prediction is wrong
- Example:

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Control Hazards & Software-based solutions

- Assumption: predict-not-taken or predict-untaken
- Where does this assumption come from?
- Compiler rearranges the code
- Control flow will change only when prediction is wrong
- Example:

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target +1				IF	ID	EX	MEM	WB	
Branch target +2					IF	ID	EX	MEM	WB

Control Hazards & Software-based solutions

- Assumption: predict-taken
- Compiler rearranges the code, for both the assumptions, so that the most frequent path matches the hardware's choice
- Predict-taken has less advantages than predict-not-taken

Control Hazards & Software-based solutions

- Which assumption is suitable for pipelined MIPS-based architecture?
- Predict-untaken

Control Hazards & Software-based solutions

- How many types of branch are possible?
  - Forward branch
  - Backward branch
- For backward-type branch, predict-taken is suitable
- For forward-type branch, predict-untaken is suitable

Control Hazards & Software-based solutions

- Is there another way to specify predict-taken or predict-untaken in the branch instruction?
- A bit in the branch opcode
  - Bit set means predict-taken
  - Bit not set means predict-untaken
- Who will set or reset such bit?
  - Compiler

Control Hazards & Software-based solutions

- Can we eliminate one cycle delay associated in branch prediction-taken?
- Delayed branch technique

Control Hazards & Delayed branch (S/W-based approach)

- Delayed branch technique
- Examples
- Find useful & independent instruction
- How many delay slots?

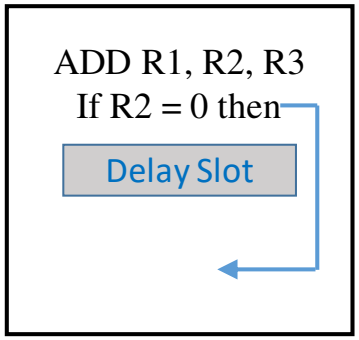
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch Delay Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Branch Delay Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

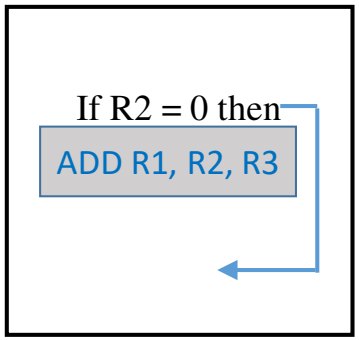
Control Hazards & Delayed branch (S/W-based approach)

- How does compiler find useful & independent instruction?
- Can compiler always find such instruction?

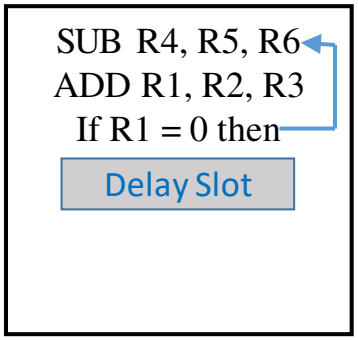
Control Hazards & Delayed branch (S/W-based approach)

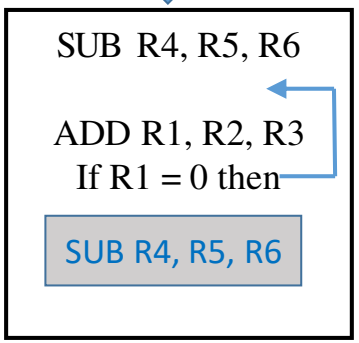
- How does compiler find useful & independent instruction?
- Can we put a branch instruction in the delay slot?



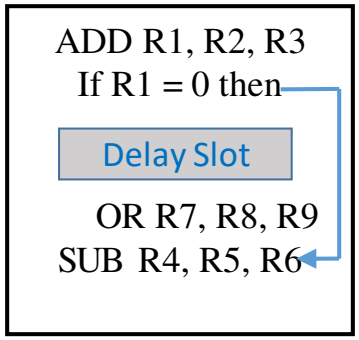


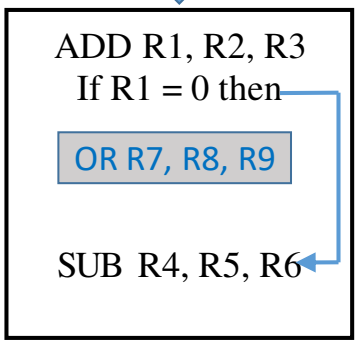
From before





From target





From fall-through

Control Hazards, static analysis & software-based solution

- Previous software-based approaches are based on static analysis
  - Assumption on Processor Design
    - Predict-not-taken
    - Predict-taken
  - Assumption on Control flow
    - Forward
    - Backward
  - Delayed branch
- Can we take decision during execution of the program?

Control Hazards and Dynamic Branch Prediction

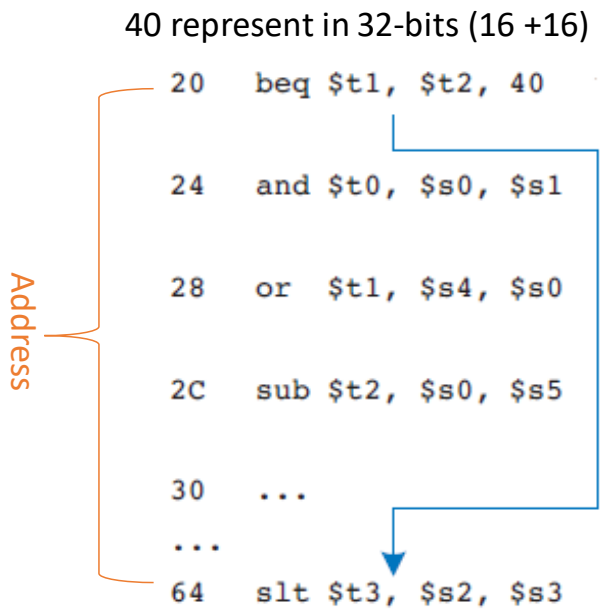
- Previous software-based approaches are based on static analysis
- We give equal priority to each branch instruction, however reality may differ
- Branch prediction change if inputs of the program change
- We need to predict the branch behavior during runtime

Control Hazards and Dynamic Branch Prediction

- How does one predict the behavior during execution?
- Exploit the previous execution history of the instruction
- What components needed to do such thing?
- Component to make prediction & store target address
- Which stage of the pipeline is suitable for that?
- IF-stage

Control Hazards and Dynamic Branch Prediction

- The *branch-target buffer* (BTB) or *branch-target (address) cache* (BTAC) is a branch-prediction cache that stores the predicted address for the next instruction after a branch



Branch address	Target address	Prediction bits
20	64	
...	...	...

Control Hazards and Dynamic Branch Prediction

- The BTB is accessed during the IF stage
- It consists of a table with branch addresses, the corresponding target addresses, and prediction information
- The PC for the next instruction to fetch is compared with the entries in the BTB. If a matching entry is found in the BTB, fetching can start immediately at the target address

Control Hazards and Dynamic Branch Prediction

- What is this prediction bit in the BTB?
- How many bits are needed?

Control Hazards and Dynamic Branch Prediction

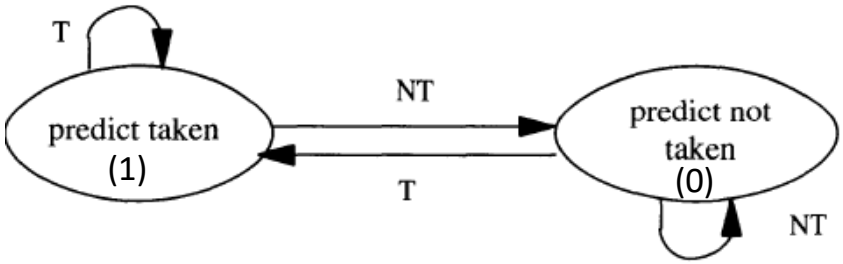
- Example: branch outcome sequence (T – Taken & N – Not Taken)
  - T N T N T N T N T N
- How does one design such a predictor?

Control Hazards and Dynamic Branch Prediction

- One bit predictor
  - If the bit is set, the branch is predicted taken
  - If the bit is not set, the branch is predicted not taken
  - In the case of a misprediction, the bit state is reversed and stored back and so is the prediction direction
- How does one design such predictor?
  - Encode the states
  - Relation among the states
  - Finite State Machine (FSM)

Control Hazards and Dynamic Branch Prediction

- One bit predictor
  - If the bit is set, the branch is predicted taken
  - If the bit is not set, the branch is predicted not taken
  - In the case of a misprediction, the bit state is reversed and stored back and so is the prediction direction



T for Taken
NT for Not Taken

Is there any shortcoming of this approach?

Example: see in the BranchPrediction.xlsx

Control Hazards and Dynamic Branch Prediction

- Shortcoming of one bit predictor
- Predict taken always
- Predict incorrectly twice (why?), rather than once, when it is not taken
- What if we have the nested loops
  - Two misprediction for inner loop
    - Entry in the loop
    - Exit from the loop
  - How does one avoid such double misprediction?

Control Hazards and Dynamic Branch Prediction

Two-bit predictor:

- Two-bits are in each entry in the BHT
- The two bits stand for the prediction states:
  - "predict strongly taken"
  - "predict weakly taken"
  - "predict strongly not taken"
  - "predict weakly not taken"

- For a misprediction in the "strongly" state cases, the prediction direction is not changed, rather the prediction goes into the respective "weakly" state

- A prediction must miss twice, before changing the state

Control Hazards and Dynamic Branch Prediction

- How does one design such 2-bits predictor?

Control Hazards and Dynamic Branch Prediction

- How does one design such 2-bits predictor?
  - Encode the different states
  - Relation among the states
  - Finite State Machine (FSM)

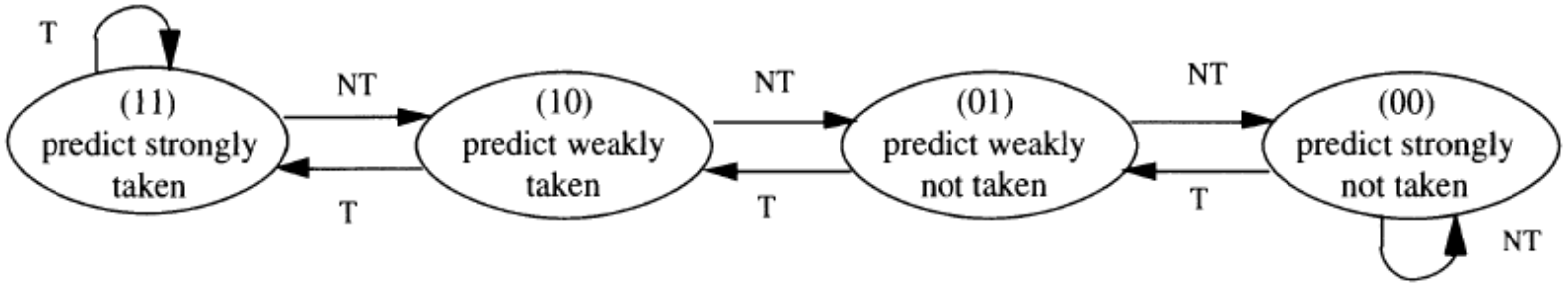
Control Hazards and Dynamic Branch Prediction

- Two kinds of 2-bits prediction methodology
  - The saturation up-down counter
  - Others

Control Hazards and Dynamic Branch Prediction

- The saturation up-down counter
  - Taken branch
    - Increment
  - Not taken
    - Decrement
  - Saturation
  - MSB of a state determine the prediction

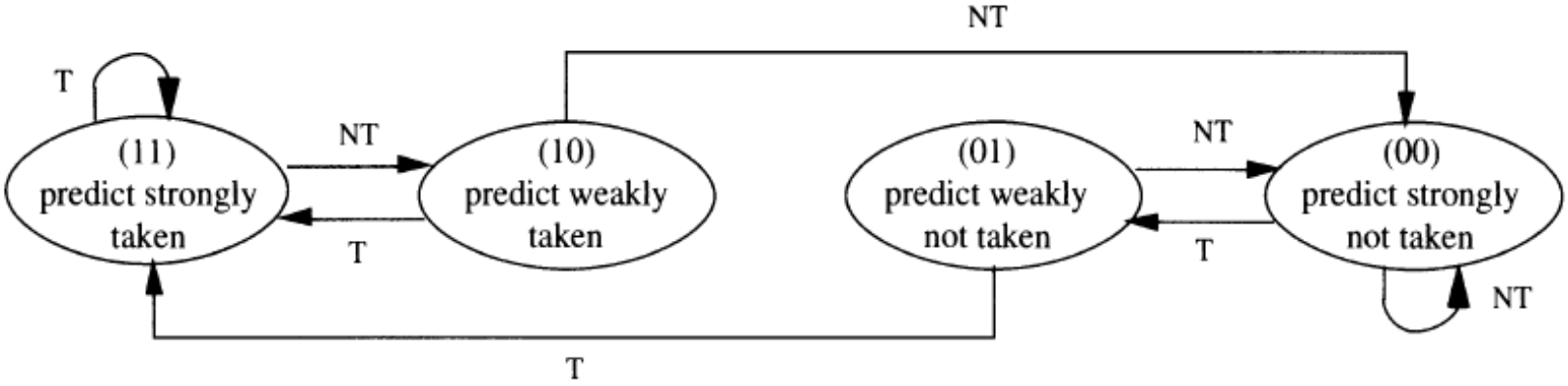
T for Taken
NT for Not Taken



Control Hazards and Dynamic Branch Prediction

- Other methodology
  - It differs from the saturation up-down counter method by changing directly from the "weakly" to the "strongly" states, in case of misprediction
  - Minimize the switching (or power consumption)

T for Taken
NT for Not Taken



Example: see in the BranchPrediction.xlsx

Control Hazards and Dynamic Branch Prediction

- n-bits predictor
  - n-bit counter (0 to 2^n-1)
  - Taken when counter value is one-half of the max. value (2^n-1)
  - Otherwise, Not taken
  - Studies of n-bits predictor have shown that 2-bits predictor do almost well, thus most systems rely on 2-bits predictor

Summary

- Control hazard
- Performance analysis
- Flush the pipeline
- Hardware-based solution technique
  - Take decision at ID stage
- Software-based solution technique
  - Predict-taken
  - Predict-untaken
  - Delayed branch
- Dynamic branch prediction techniques

Computer Architecture (CS F342)

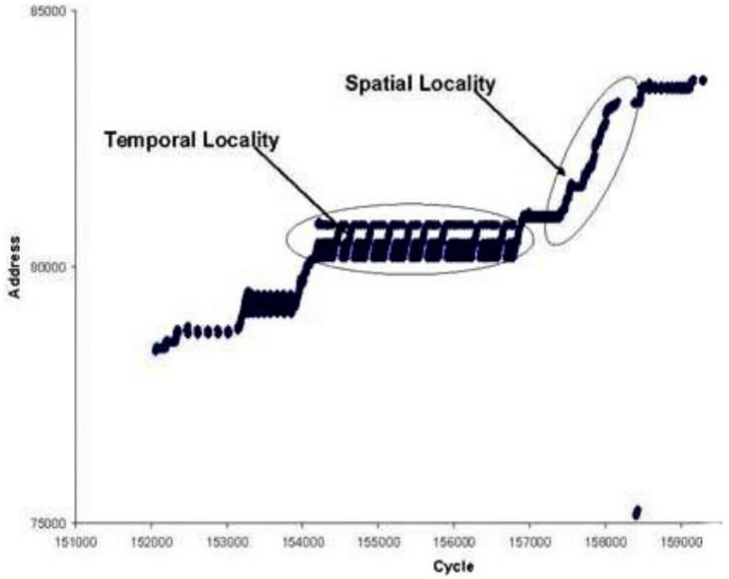
Memory/Storage Hierarchy

&

Fast Storage Unit: Cache Memory Architecture & Organization

Why do we need to study memory/storage hierarchy?

- CPU is a component in the computer systems
- Others components: Memory and I/O systems
- Programs exhibit principle of locality
  - Temporal
  - Spatial
  - A rule of thumb (the 90/10 rule): 90% of the execution of programs spends in only 10% of the code



Behavior of a program

Matrix multiplication

- Data stored in row-major order
- Data of A, B & C can be used in near future
- Data neighboring to previously accessed data
- Instructions are also to be used in near future
- Principle of locality

```
for (i=0; i<l; i++)
    for (j=0; j<m; j++)
        for (k=0; k<n; k++)
            A[i][j] += B[i][k] x C[k][j];
```

Why do we need to study memory/storage hierarchy?

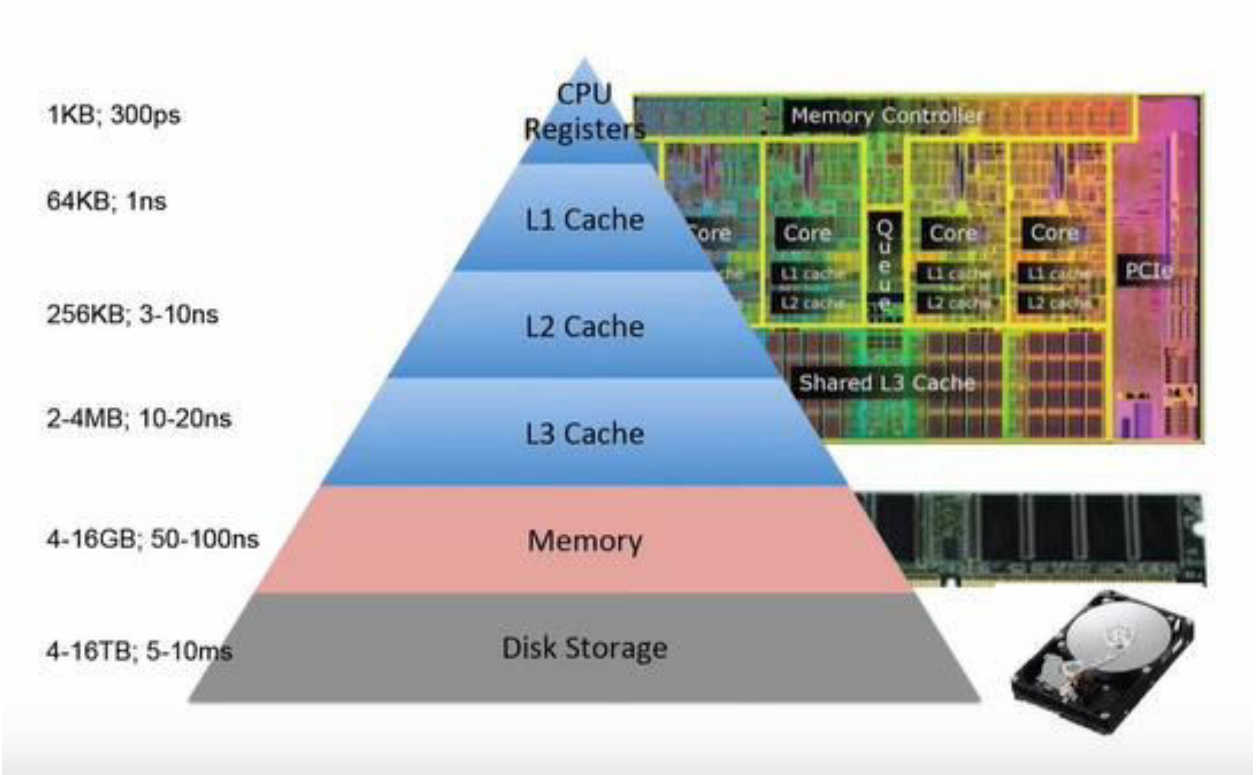
- Principle of locality can be found in most of the programs

- To exploit such principle we need memory hierarchy
  - Keep the repeatedly accessed data & instruction near to CPU
  - Not the entire code

- Memory hierarchy
  - Faster but smaller memory closer to CPU
  - Slower but larger memory faraway from the CPU

Memory hierarchy

Access **time** and **space** increase



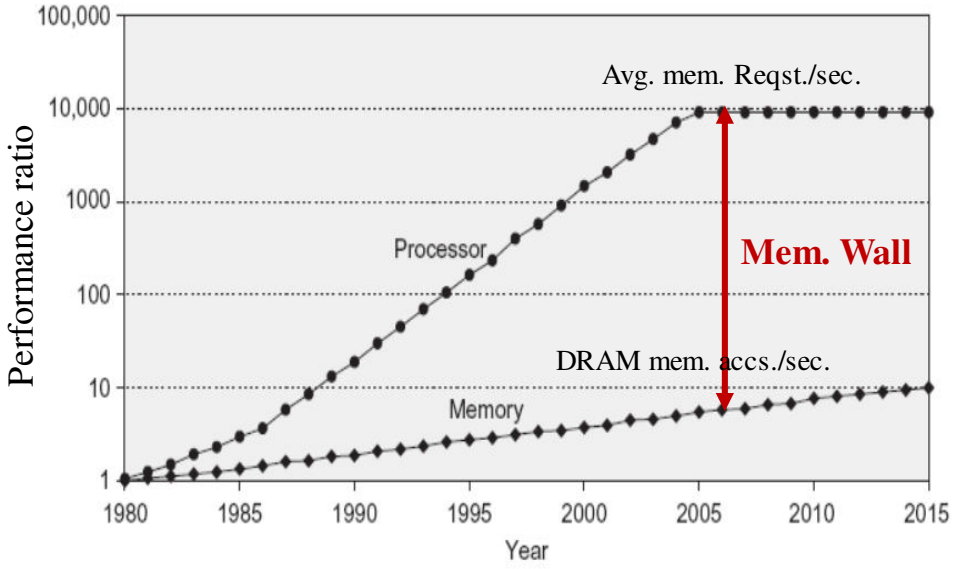
Cost increase

What makes improvement in the storage access time?

- Static Random Access Memory (SRAM) Technology
  - Registers, L1, L2 & L3 cache
- Dynamic Random Access Memory (DRAM) Technology
  - Main memory
- Magnetic Technology
  - Hard disk takes longer access time because of mechanical components

Do we really need memory hierarchy?

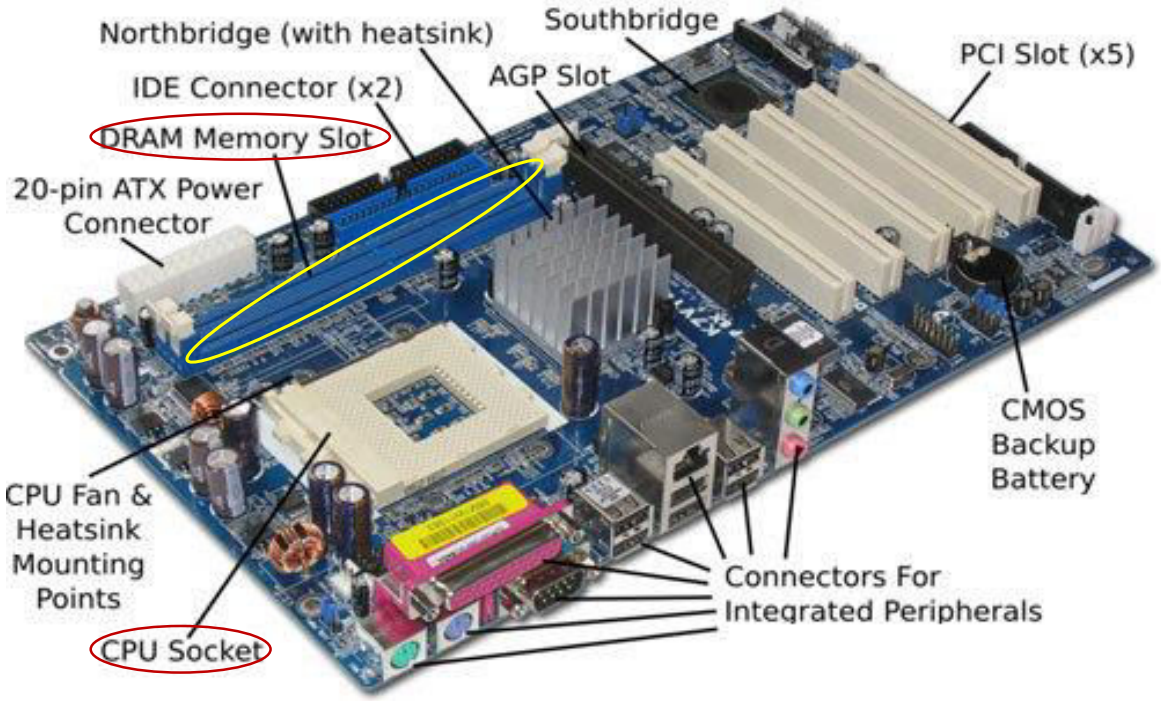
- Memory Wall Problem:
  - Significant increase in processor performance over the years
  - Not significant increase in main-memory performance over the years



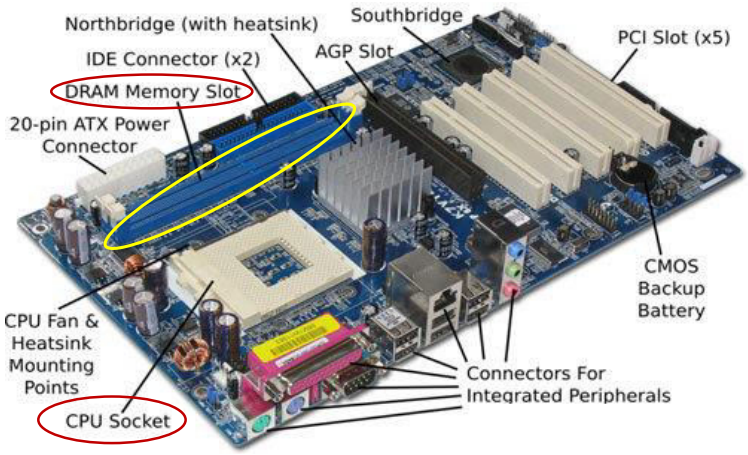
Necessity of memory hierarchy

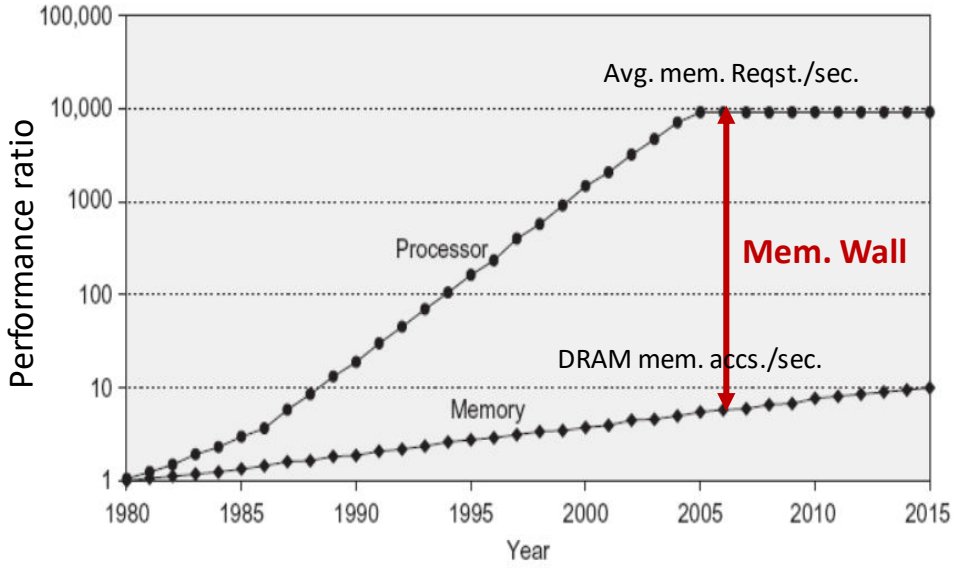
- From the previous graph
  - The gap is increasing
- The previous graph did not include the multiprocessors
  - The aggregated peak bandwidth requirement increases with no. cores/processors
- How does one deal with this increasing gap?
  - Need memory hierarchy: multi-levels of cache hierarchy

Why do we need cache memory?



Memory Wall Problem and Necessitv of Cache Memorv

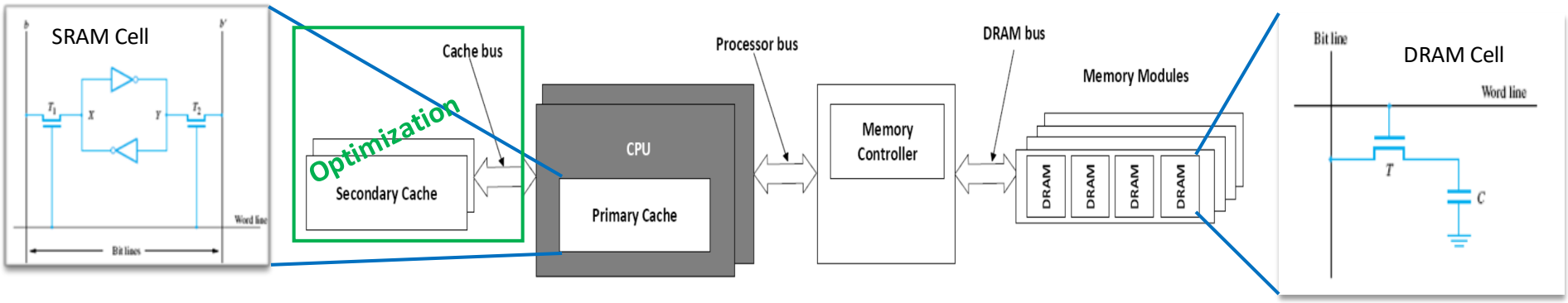




- Performance difference between processor (CPU) and memory by technology and memory is placed far away (*nm* scale) from CPU (off-chip)
- There is a gap in CPU's request (rate) for the memory accesses and the service (rate) for those request by memory
-
- Cache memory technique can speed up the performance of the memory accesses time

Cache Memory Architecture

- Cache memory is a high-speed storage unit
- **What makes it faster as compare to the memory in question?**
  - 1) Position 2) SRAM-based memory technology
- **What would be the size and characteristic of the Cache memory?**
  - Smaller in size and bit access time must be faster



Program's Behavior and Characteristics of Cache Memory

Which memory request should be keep in the cache?

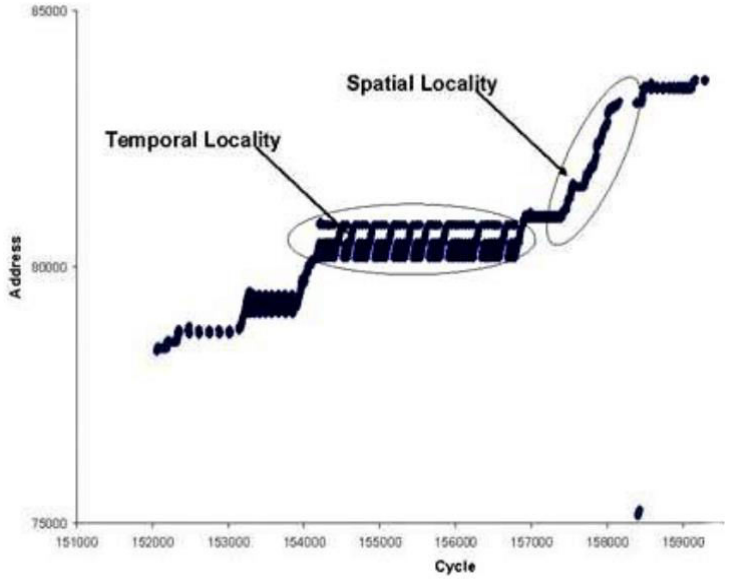
- Program's behavior: programs tend to reuse data and instructions they have used recently.
- Spatial locality and Temporal locality

How many such instructions and/or data one can keep in the cache?

- More than one or a block of instructions and/or data

How does one decide the block size?

- Processor's cache is managed by its own set of heuristics or rules
- For example, in Intel i7, block size of the primary cache is 64 bytes



Behavior of a program

Characteristics of Cache Memory

The following items are embedded in the cache:

Organization

The logical arrangement of storage unit/data

Content-management heuristics

Decide the best possible items for caching and evict out the candidate to make room for more important data not yet cached

Consistency-management heuristics

Ensure that the instructions and data that the program expects to receive are the ones the program does, indeed, receive

Consistency with 1) self, 2) main memory 3) other caches

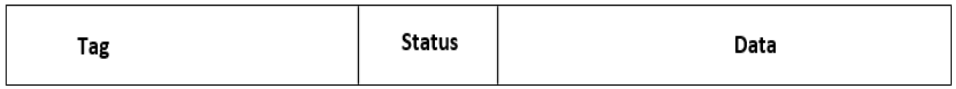
Cache Organization: Blocks, Tags and Set

A cache stores chunks of data (called cache blocks or cache lines) that come from the memory.

A cache is typically much smaller than the memory:

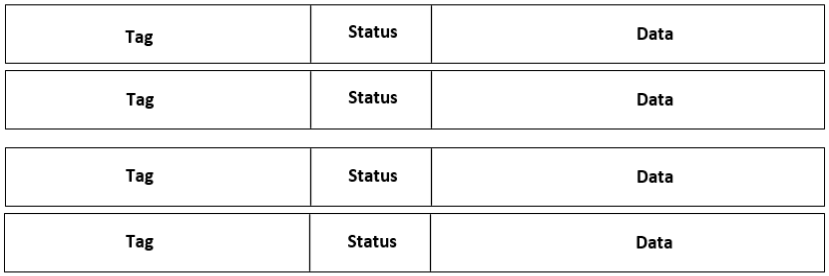
How does CPU know whether any particular datum is present in the cache or not?

Cache tags fulfil this necessity.



What if the cache contains more than one block?

Cache can have the set of choices for the incoming blocks



Optimization Techniques to Improve the Memory Access Time (or Miss Rate)

How does following items affect the cache miss rate?:

- Cache size (higher or smaller)
- Cache block size (higher or smaller)
- Set or associativity size (higher or smaller)
- Cache hierarchies
  - Issues in hierarchy management

Summary

- Necessity of Memory Hierarchy
- Necessity of Cache memory
- Characteristic of Cache memory
- Program's behavior
- Elements of Cache Organization
- Elements of Cache Optimization