

CPU Design

Debiprasanna Sahoo

Assistant Professor

Department of Computer Science and Engineering
School of Electrical and Computer Sciences
Indian Institute of Technology Bhubaneswar



Content

Book

Computer Organization and Design:
The Hardware/Software Interface-
RISC-V Edition, 5th Edition, 2017

Chapter-4

David A. Patterson and John L.
Henessey

Reference Books

Computer Organization and Design:
The Hardware/Software Interface-
MIPS Edition, 5th Edition, 2017

Chapter-4

Computer Architecture: A
Quantitative Approach

Appendix-C

David A. Patterson and John L.
Henessey

Manual

The RISC-V Instruction
Set Manual

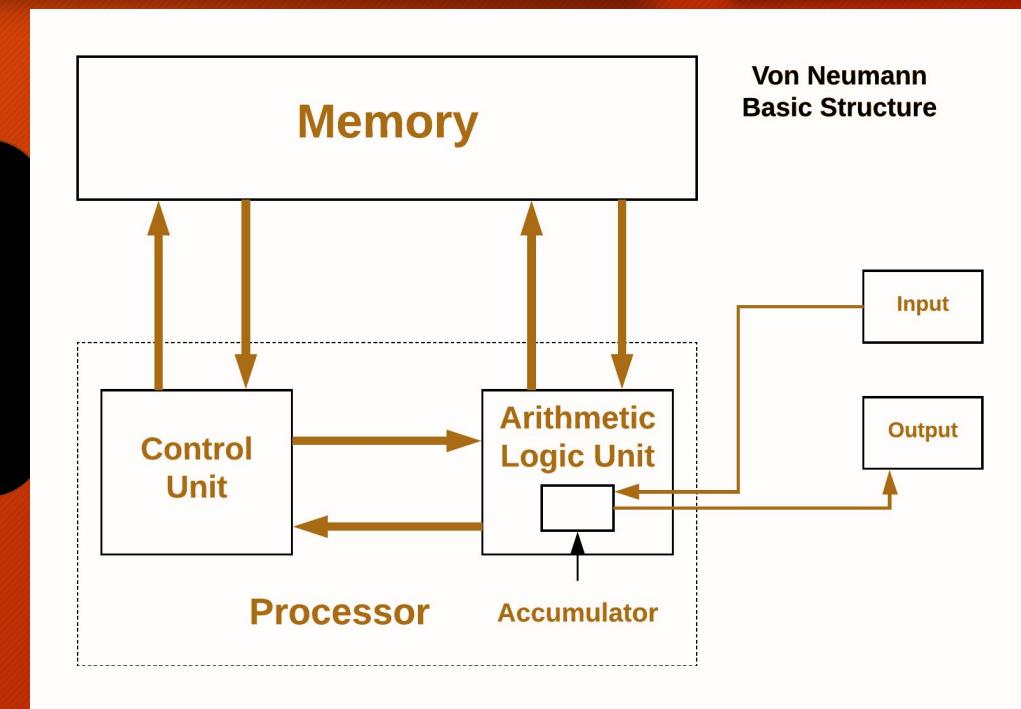
Volume I: User-Level ISA
Document Version 2.2

Andrew Waterman and
Krsti Asanovi

Von-Neumann Architecture (Recap)

Components of a Computer:

- Processor which does the computation with the help of Arithmetic and Logic Unit (ALU) and Control Unit (CU).
- Memory that stores data on which computation can happen.
- Input and Output Devices that supplies or uses data item.

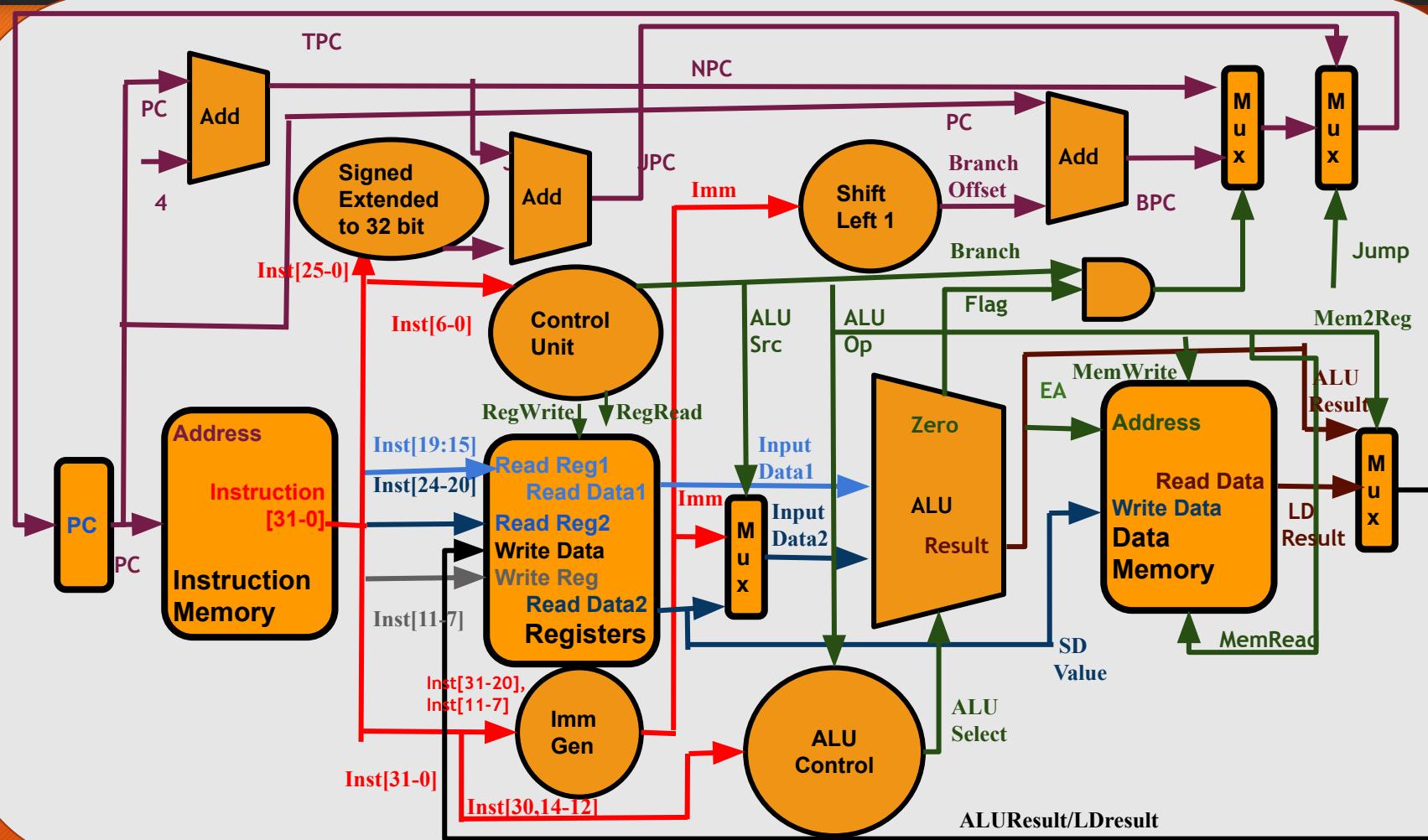


Geeks for Geeks

Learning Outcome: A Basic Design of RISC-V

- Create and draw different components of a basic RISC-V Computer with the following capabilities:
 - The memory-reference instructions load doubleword (ld) and store doubleword (sd)
 - The arithmetic-logical instructions add, sub, and, and or
 - The conditional branch instruction branch if equal (beq)
 - No shift, multiply, floating point, etc.
- Simulate a basic RISC-V Computer with above capabilities

Overview of Implementation (Control and Data Path)



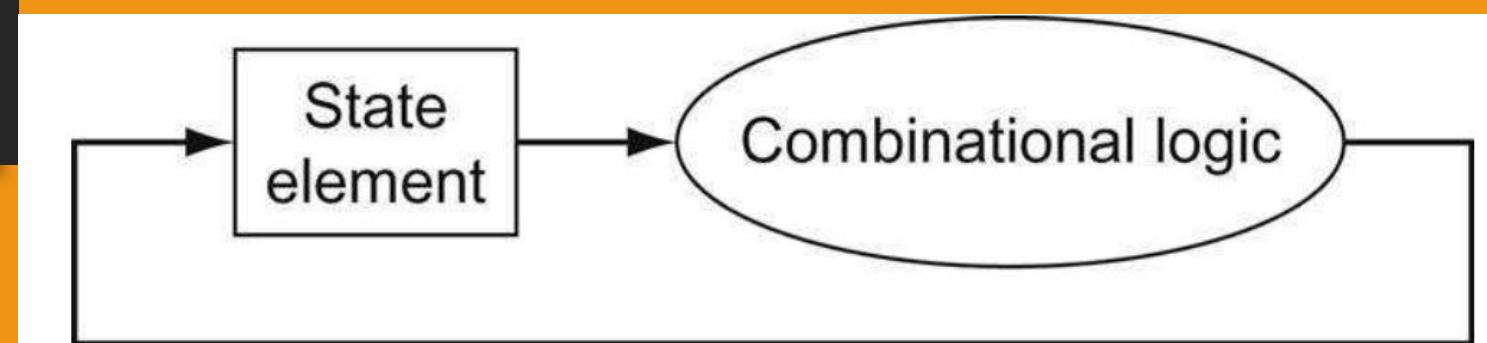
Digital Logic Fundamentals

Combinational Circuits

Sequential Circuits

State Elements or Memory

Clocks



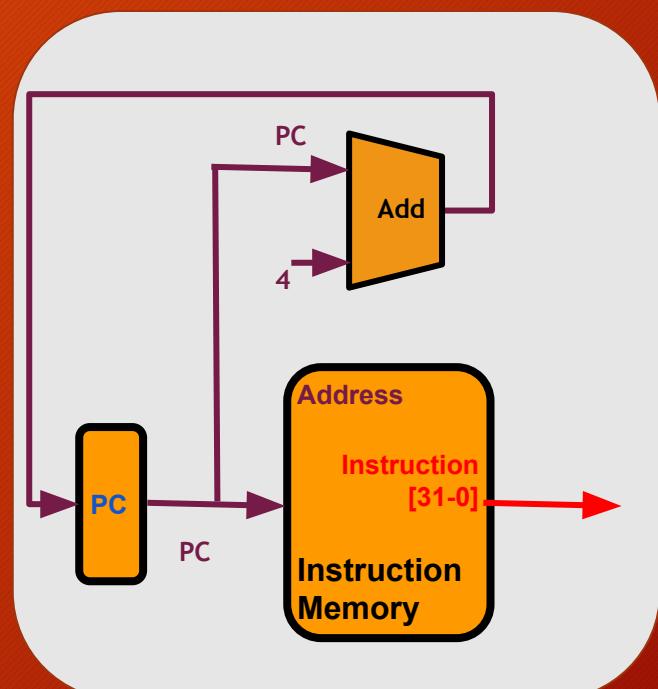
Instruction Fetch

Instruction Memory:
Stores instructions
as machine code

Program Counter:
Points to the next
address of the
instruction to be
executed

Adder: To calculate
the next address to be
executed (except
branch instructions)

- Fetched instruction and PC is incremented
 - $\text{instruction} = \text{IM}[\text{PC}]$
 - $\text{PC} = \text{PC} + 4$



Register File Design

Read Registers: Register index specified in an instruction to access an input register

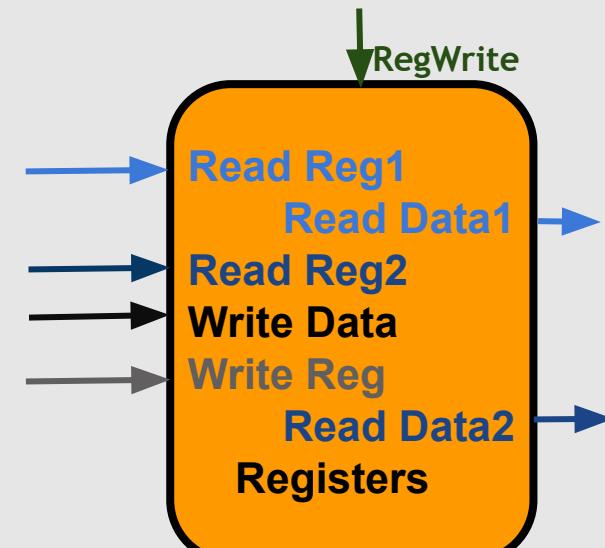
Read Data: Output data of the register accessed

Write Registers: Register index specified in an instruction to write results

Write Data: Input data for the register to be written

RegWrite: Read/Write Signal

- **Read and Write Registers**
 - $rsl1 = \text{instruction}[19-15]$
 - $rsl2 = \text{instruction}[24-20]$
 - $rd = \text{instruction}[11-7]$
 - $rs1 = \text{GPR}[rsl1]$
 - $rs2 = \text{GPR}[rsl2]$
 - $\text{GPR}[rd] = \text{ALUResult}$



Data Memory

Address: Effective memory address to be read (EA) calculated as ALUResult for Load and Store Instructions

Read Data: Output data of the memory address (LDResult)

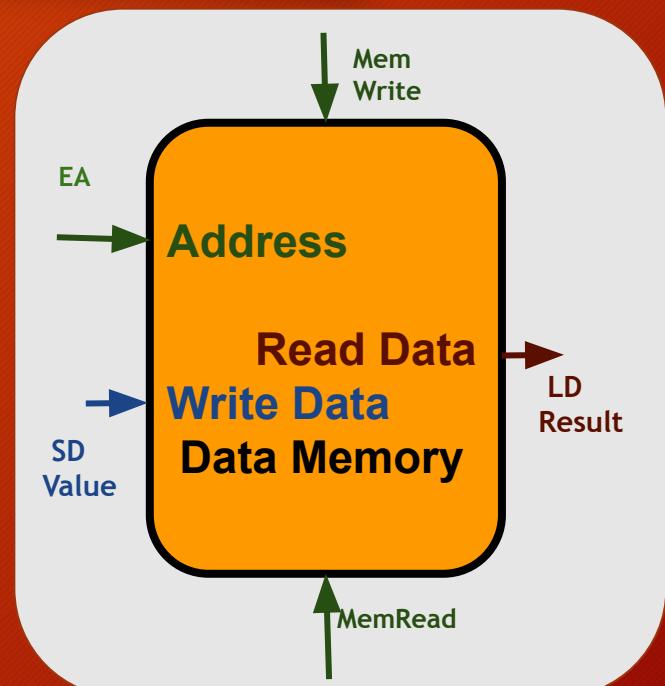
Write Data: Input data from the register to be written to memory (SD_Value)

MemRead: Read Signal

MemWrite: Write Signal

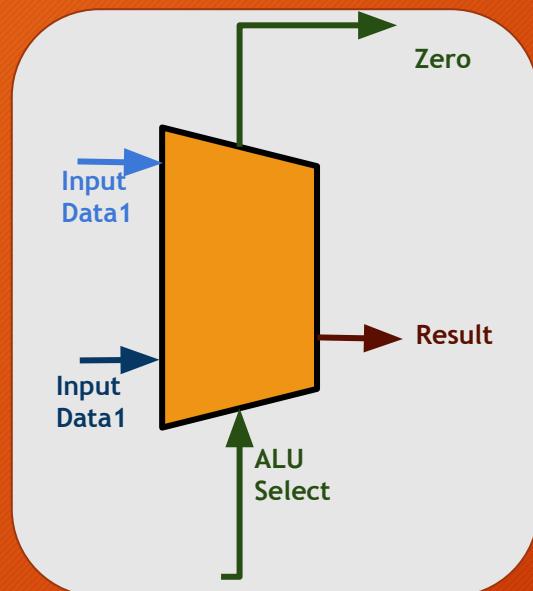
Both MemRead and MemWrite signals can't be high in the same time.

- If MemWrite
Then **DM[ALUResult] = SD_Value**
- If MemRead
Then **LDResult = DM[ALUResult]**



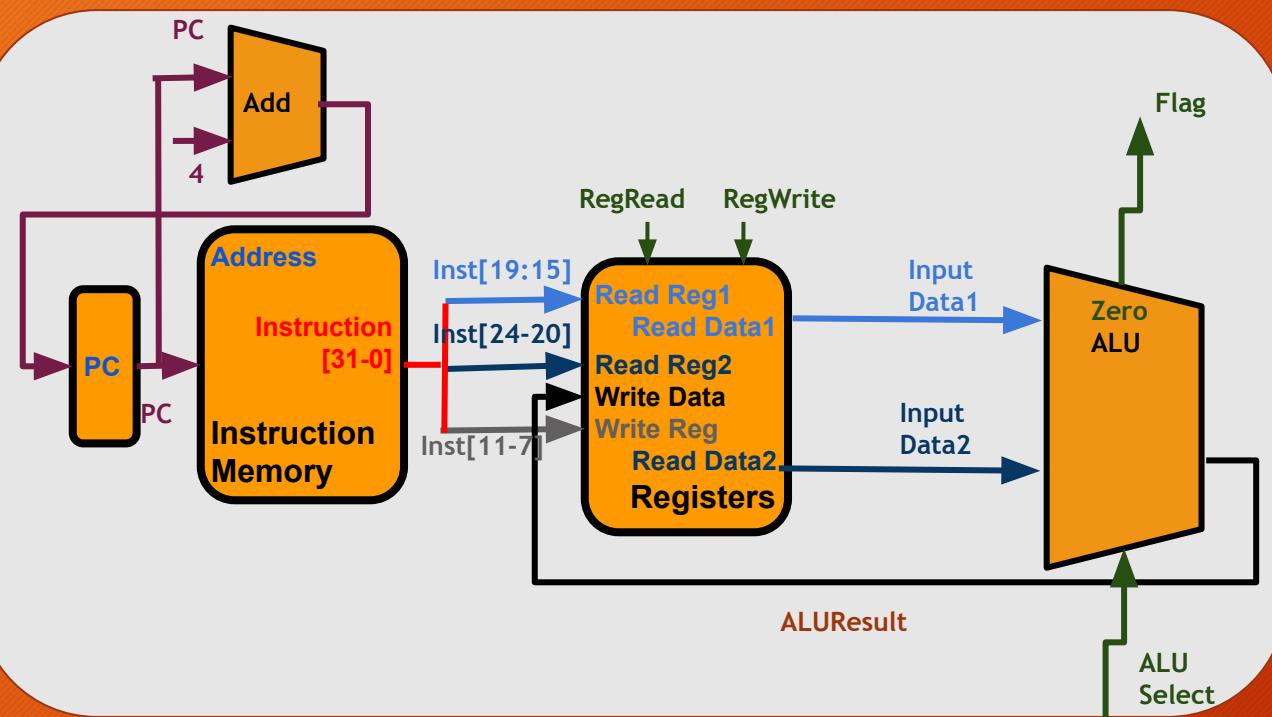
ALU Design

- If ALUSrc
- Then rs2 = imm
- Else Then rs2 = GPR[rs12]
- ALUresult = ALU(ALUSelect, rs1, rs2)
- ALUZeroFlag = (rs1 == rs2)



Signal	Type	Source/Destination	As Per Figure and Code
Register: rs1, rs2 Immediate: Imm	Input	Source: Register/Instruction	rs1: Input Data1 rs2, imm: Input Data2
Register: rd	Output	Destination: Registers for Non-LS Instruction	Result
Effective Address: EA	Output	Destination: Memory for LS Instructions	Result
Status Signal: Flags	Output	Destination: Various Operations	Zero
Control Signal	Input	Source: Control Unit	ALU Select. ALUSrc

Data-path Design for R-Type Instruction



- instruction = IM[PC]
- PC = PC + 4
- rs1 = instruction[19-15]
- rs2 = instruction[24-20]
- rd = instruction[11-7]
- If RegRead Then rs1 = GPR[rs1]
- If RegRead Then rs2 = GPR[rs2]
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- func7 = instruction[31-25]
- ALUresult = ALU(ALUSelect, rs1, rs2)
- If RegWrite Then GPR[rd] = ALUresult

- RegRead enabled to read GPR
- RegWrite enabled to write GPR

func7[31-25]

rs2 [24-20]

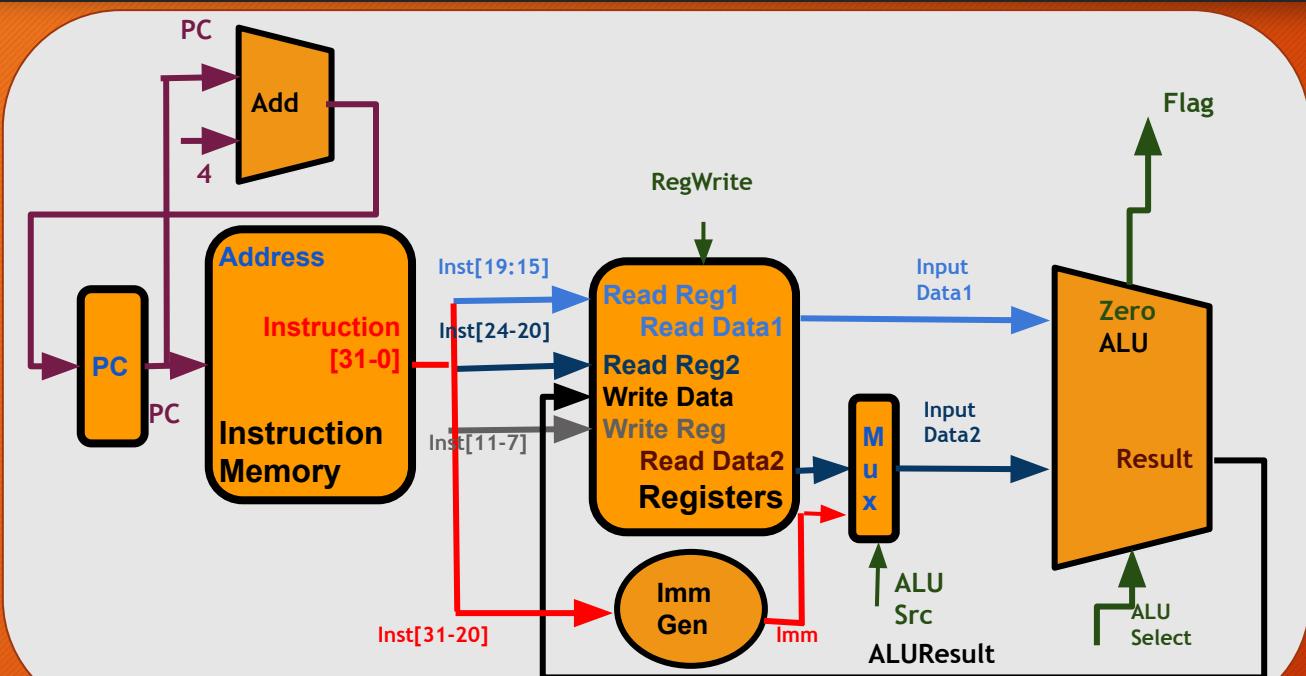
rs1 [19-15]

func3 [14-12]

rd [11-7]

Opcode [6-0]

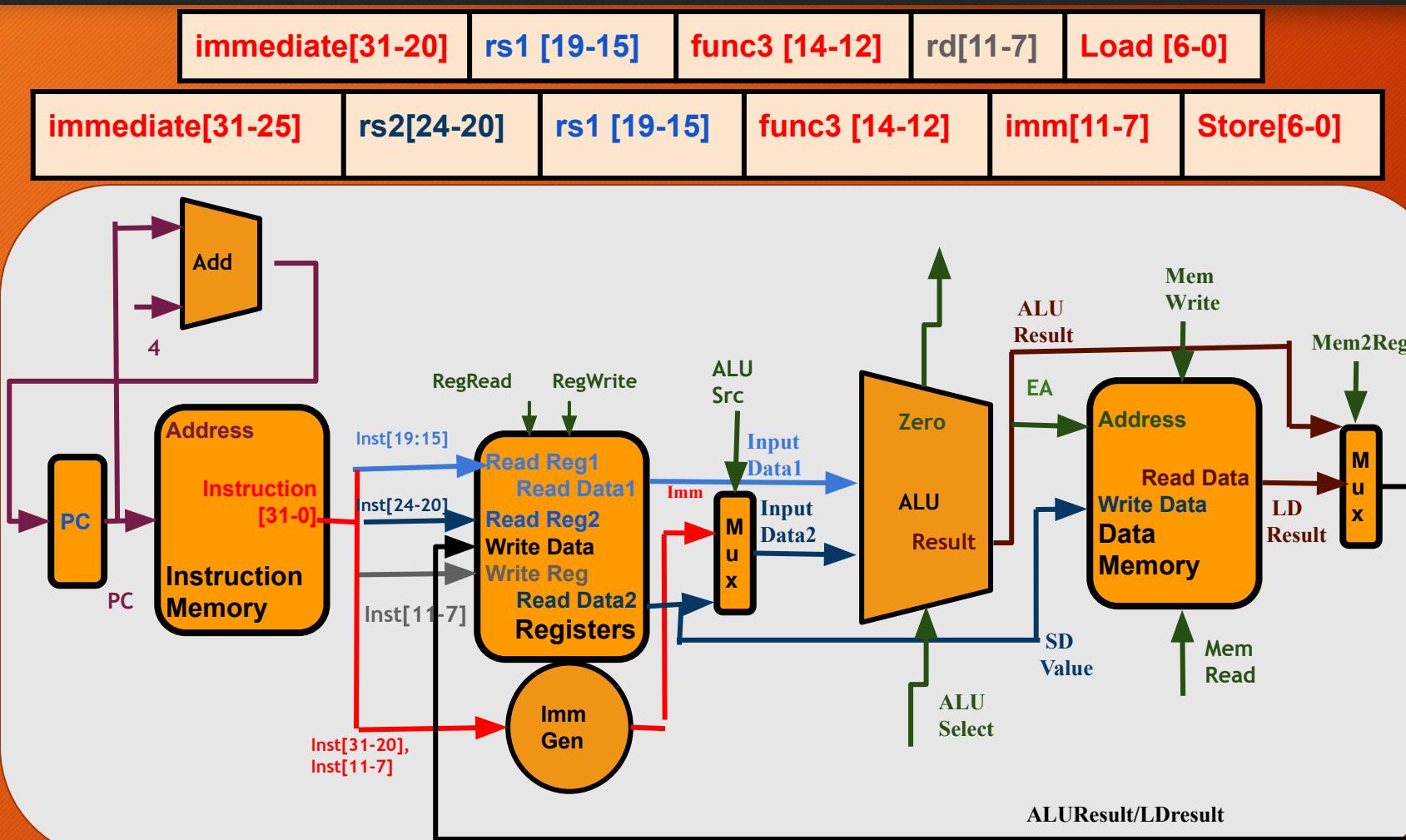
Data-path Design after Adding I-Type Instruction



- **ALUSrc:** Selects between rs2 and Imm
 - rs is used for R-Type
 - imm is used for I-Type
- **RegRead** enabled to read GPR
- **RegWrite** enabled to write GPR

- **instruction = IM[PC]**
- **PC = PC + 4**
- **imm = instruction[31-20]**
- **opcode = instruction[6-0]**
- **func3 = instruction[14-12]**
- **rsl1 = instruction[19-15]**
- **rsl2 = instruction[24-20]**
- **rd = instruction[11-7]**
- **rs1 = If RegRead Then GPR[rsl1]**
- **If ALUSrc**
- **Then rs2 = imm**
- **Else If RegRead Then rs2 = GPR[rsl2]**
- **ALUResult = ALU(ALUSelect, rs1, rs2)**
- **If RegWrite Then**
- **GPR[rd] = ALUresult**

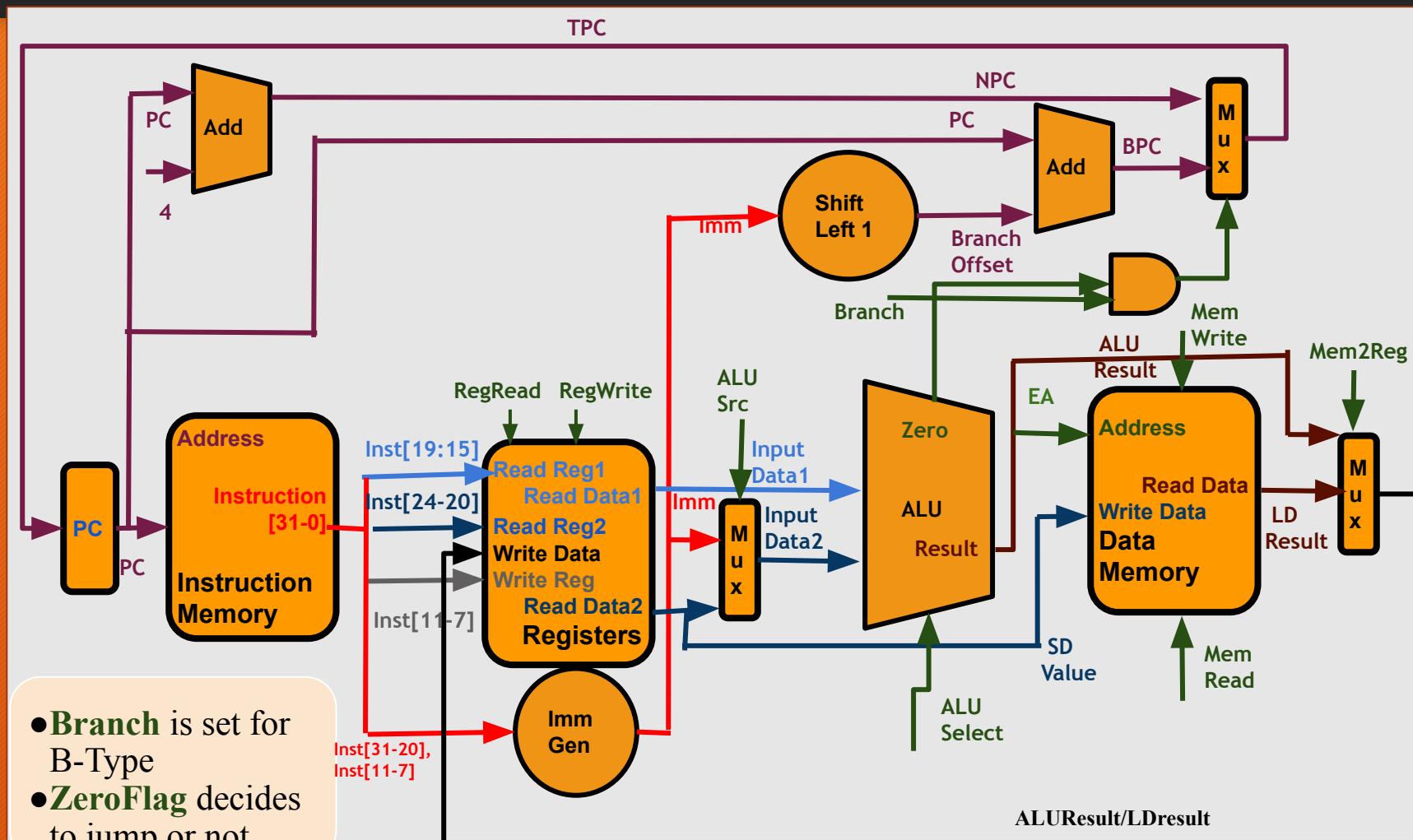
Data-path Design after Adding Memory Instructions



- **instruction = IM[PC]**
- **PC = PC + 4**
- **imm=ImmGen(instruction[31-20],instruction[11-7])**
- **opcode = instruction[6-0]**
- **func3 = instruction[14-12]**
- **rsl1 = instruction[19-15]**
- **rsl2 = instruction[24-20]**
- **rd = instruction[11-7]**
- **If RegRead Then rs1 = GPR[rsl1]**
- **If ALUSrc**
- **Then rs2 = imm**
- **Else If RegRead Then rs2 = GPR[rsl2]**
- **ALUresult = ALU(ALUSelect, rs1, rs2)**
- **If MemWrite**
Then DM[ALUresult] = rs2
- **If MemRead**
Then LDResult = DM[ALUresult]
- **If RegWrite Then**
- **GPR[rd] = Mem2Reg(ALUresult, LDresult)**

- **RegRead** enabled to read GPR
- **ALUSrc:** Selects between rs2 and Imm
 - rs is used for R-Type and SD
 - imm is used for I-Type, LD, SD
- **MemWrite** is used for SD
- **MemRead** is used for LD
- **RegWrite** enabled to write GPR
- **Mem2Reg** selects between ALUResults and LDResults

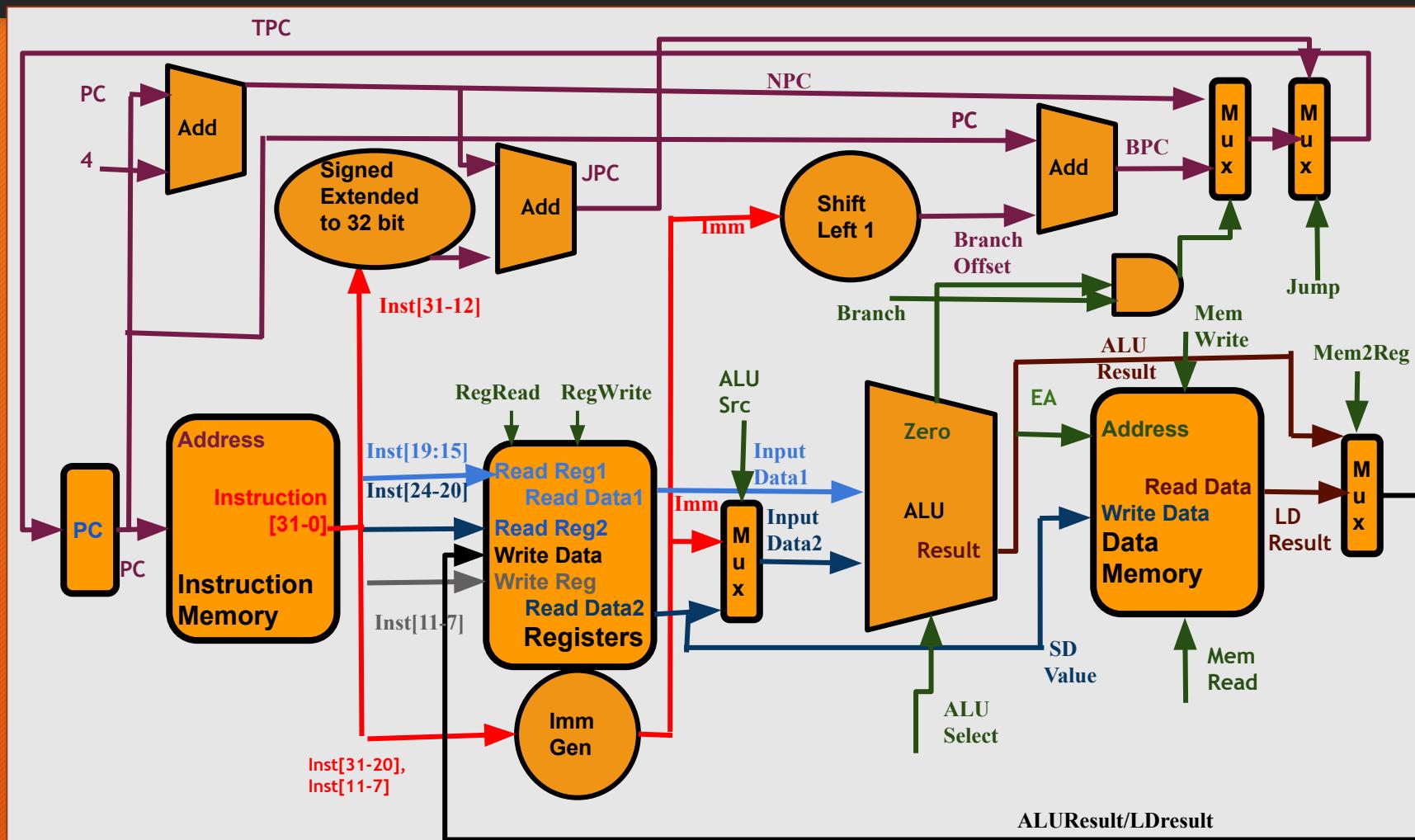
Data-path Design after Adding B-Type Instructions



- Branch is set for B-Type
- ZeroFlag decides to jump or not

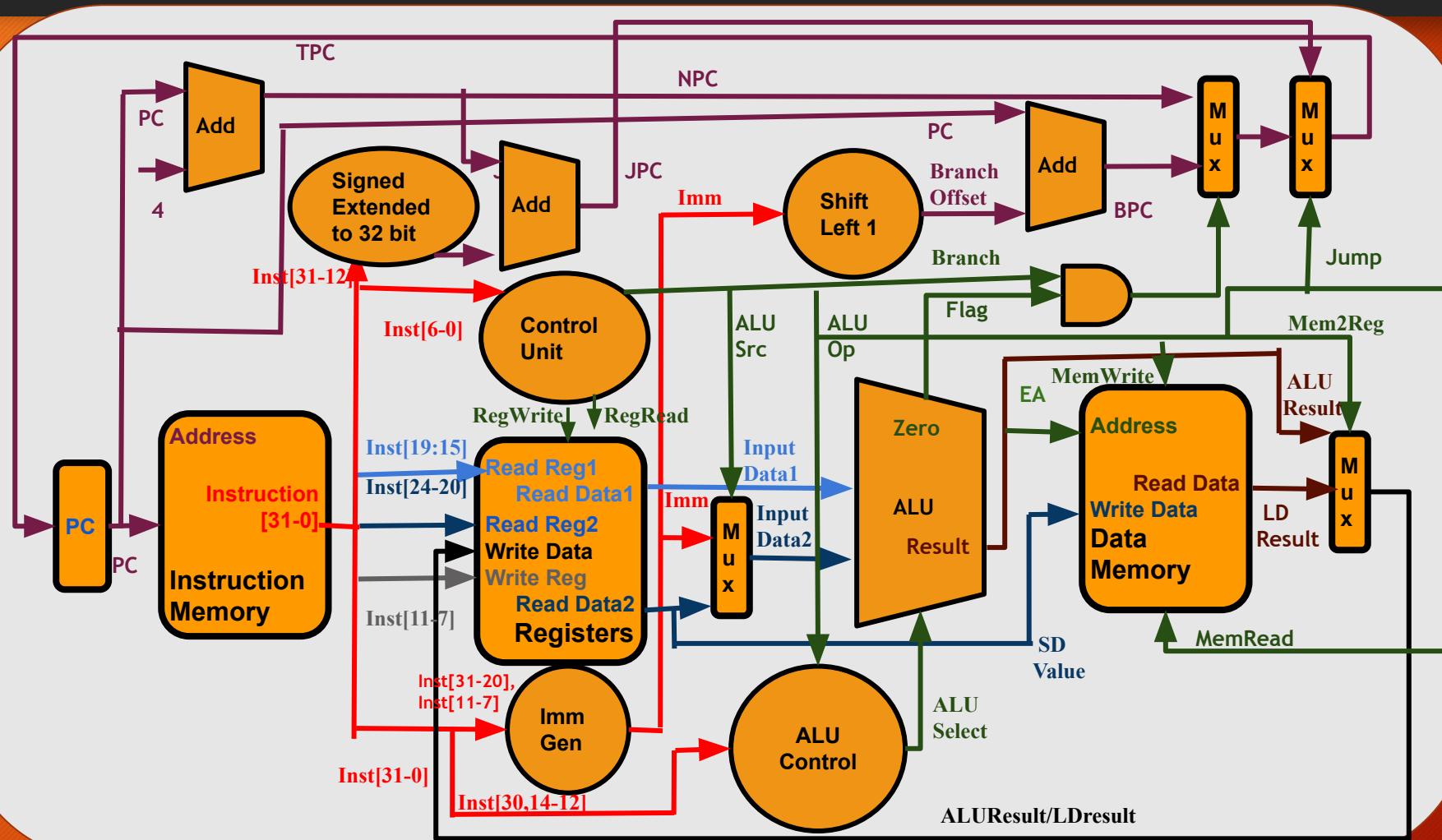
- instruction = IM[PC]
- NPC = PC + 4
- imm=ImmGen(instruction[31-20],instruction[11-7])
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rsl1 = instruction[19-15]
- rsl2 = instruction[24-20]
- rd = instruction[11-7]
- rs1 = GPR[rsl1]
- If ALUSrc
- Then If RegRead Then rs2 = imm
- Else If RegRead Then rs2 = GPR[rsl2]
- ALUresult/EA = ALU(ALUSelect, rs1, rs2)
- ALUZeroFlag = (rs1 == rs2)
- If MemWrite Then DM[EA] = rs2
- If MemRead Then LDResult = DM[EA]
- BPC = (imm << 1) + PC
- If Branch and ALUZeroFlag
- Then TPC = BPC Else TPC = NPC
- If RegWrite Then
- GPR[rd] = Mem2Reg(ALUresult, LDresult)
- PC = TPC

Data-path Design after Adding J-Type Instructions



- **instruction = IM[PC]**
- **NPC = PC + 4**
- **JPC = NPC + signedExtend(instruction[31-12])**
- **imm = ImmGen(instruction[31-20], instruction[11-7])**
- **opcode = instruction[6-0]**
- **func3 = instruction[14-12]**
- **rsl1 = instruction[19-15]**
- **rsl2 = instruction[24-20]**
- **rd = instruction[11-7]**
- **rs1 = GPR[rsl1]**
- **If ALUSrc**
 - Then If RegRead Then rs2 = imm
 - Else If RegRead Then rs2 = GPR[rsl2]
- **ALUresult = ALU(ALUSelect, rs1, rs2)**
- **ALUZeroFlag = (rs1 == rs2)**
- **If MemWrite Then DM[ALUresult] = rs2**
- **If MemRead Then LDResult = DM[EA]**
- **BPC = (imm << 1) + PC**
- **If Branch and ALUZeroFlag**
 - Then TPC = BPC Else TPC = NPC
- **If Jump Then TPC = JPC**
- **If RegWrite Then**
 - **GPR[rd] = Mem2Reg(ALUresult, LDresult)**
- **PC = TPC**

Adding Control Path and Controller



Control unit generates signals to all the other units of the system

Control Word: The bit signal matrix that is generated by the control unit to be used as signal. The control word constitute of RegRead, RegWrite, ALUSrc, ALUOp, Branch, Jump, MemWrite, MemRead, Mem2Reg

Adding Control Path and Controller

- **instruction = IM[PC]**
- **NPC = PC + 4**
- **JPC=NPC + SignedExtend(instruction[31-12])**
- **imm=ImmGen(instruction[31-20],instruction[11-7])**
- **opcode = instruction[6-0]**
- **func3 = instruction[14-12]**
- **rsl1 = instruction[19-15]**
- **rsl2 = instruction[24-20]**
- **rd = instruction[11-7]**
- **ControlWord = controller(opcode, func)**
- If RegRead Then **rs1 = GPR[rsl1]**
- If ALUSrc
 - Then If RegRead Then **rs2 = imm**
 - Else If RegRead Then **rs2 = GPR[rsl2]**

- **ALUSelect = ALUControl(ALUOp, imm)**
- **ALUresult = ALU(ALUSelect, rs1, rs2)**
- **ALUZeroFlag = (rs1 == rs2)**
- If MemWrite Then
 - **DM[ALUResult] = rs2**
- If MemRead Then
 - **LDResult = DM[ALUResult]**
- **BPC = (imm << 1) + PC**
- If Branch and ALUZeroFlag
 - Then **TPC = BPC**
 - Else **TPC = NPC**
- If Jump Then **TPC = JPC**
- If RegWrite Then
 - If Mem2Reg
 - Then **GPR[rd] = LDresult**
 - Else **GPR[rd] = ALUresult**
- **PC = TPC**

ControlWord

- **RegRead** enabled to read GPR
- **ALUSrc**: Selects between rs2 and Imm
- **ALUOp**: Operation to be executed on the ALU
- **MemWrite** is used for Store
- **MemRead** is used for loads
- **RegWrite** enabled to write GPR
- **Mem2Reg** selects between ALUResults and LDResults
- **Branch and Jump** selects between Branch PC, Jump PC, and Next PC

Generating Control Signals: ALU Design

ALU Action	ALU Control
AND	0000
OR	0001
Add	0010
Sub	0110

ALU action is the action intended on the ALU

Instruction Opcode	ALUOp	Operation	Func7	Func3	ALU Action	ALU Control
ld	00	Load	xxxxxx	xxx	add	0010
sd	00	Store	xxxxxx	xxx	add	0010
beq	01	Branch	xxxxxx	xxx	sub	0110
R-Type	10	Add	0000000	000	add	0010
R-Type	10	Sub	0100000	000	sub	0110
R-Type	10	And	0000000	111	AND	0000
R-Type	10	Or	0000000	110	Or	0001

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

ALUOp comes from Opcode

Func3 and Func7 selects the R-Type Inst

Operation specifies the actual intension

Generating Control Signals: ALU Design

Instruction Opcode	ALUOp	Operation	Func7	Func3	ALU Action	ALU Control
ld	00	Load	xxxxxxx	xxx	add	0010
sd	00	Store	xxxxxxx	xxx	add	0010
beq	01	Branch	xxxxxxx	xxx	sub	0110
R-Type	10	Add	0000000	000	add	0010
R-Type	10	Sub	0100000	000	sub	0110
R-Type	10	And	0000000	111	AND	0000
R-Type	10	Or	0000000	110	Or	0001

ALU Op

Solve K-map?

Func7



ALU Control Signal

Func3

Generation Control Signal: Opcode Handling

Instruction	ALU Src	Mem-Reg	Req-Write	Mem-Read	Mem-Write	Branch	ALU Op1	ALU Op0
R-Format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

“

Congratulations! We have designed a single-cycle processor. Have you implemented it?

”

How about improving the design?

INSTRUCTION PIPELINE

Learning Outcome: A Pipelined Design of RISC-V

- Analyze and calculate the impact of doing pipelining
- Create and draw different components of a pipelined RISC-V Computer with the following capabilities:
 - The memory-reference instructions load doubleword (ld) and store doubleword (sd)
 - The arithmetic-logical instructions add, sub, and, and or
 - The conditional branch instruction branch if equal (beq)
 - No shift, multiply, floating point, etc.
- Simulate a pipelined RISC-V Computer with above capabilities
- Visualize the dependency between instructions and solve the dependency challenges

Performance of a CPU

CPU Execution Time: The total time required for the computer to complete instruction execution.

Throughput (Instruction/Cycles): The number of instructions executed per cycle (IPC). It determines the performance of the system. In pipelined system, we measure number of cycles taken by an instruction (CPI).

Processor Clock: The time for one clock which runs at a constant rate.

Total number of instructions is fixed for an application. Billions of instructions are present in each application.

Frequency: $1/\text{Length of Clock Cycle}$

Clock Cycle Time Analysis

Critical Path of a Circuit: Longest path between two registers if the combinational circuit between the registers is represented as graph of elements.

Clock Cycle Time: Sum of the gate delays of the critical path.

Critical Path Non-pipelined System: The gate delays of the critical path between two PC accesses.

Instruction Class	Fetch	Decode	Execute	Memory	Writeback	Total
Id	200	100	200	200	100	800
sd	200	100	200	200	-	700
beq	200	100	200	-		500
R-Type	200	100	200	-	100	600



Critical Path

Instruction Time for Each Instruction

Pipelining: Concept

Assembly line: Overlapping execution of multiple instruction in parallel

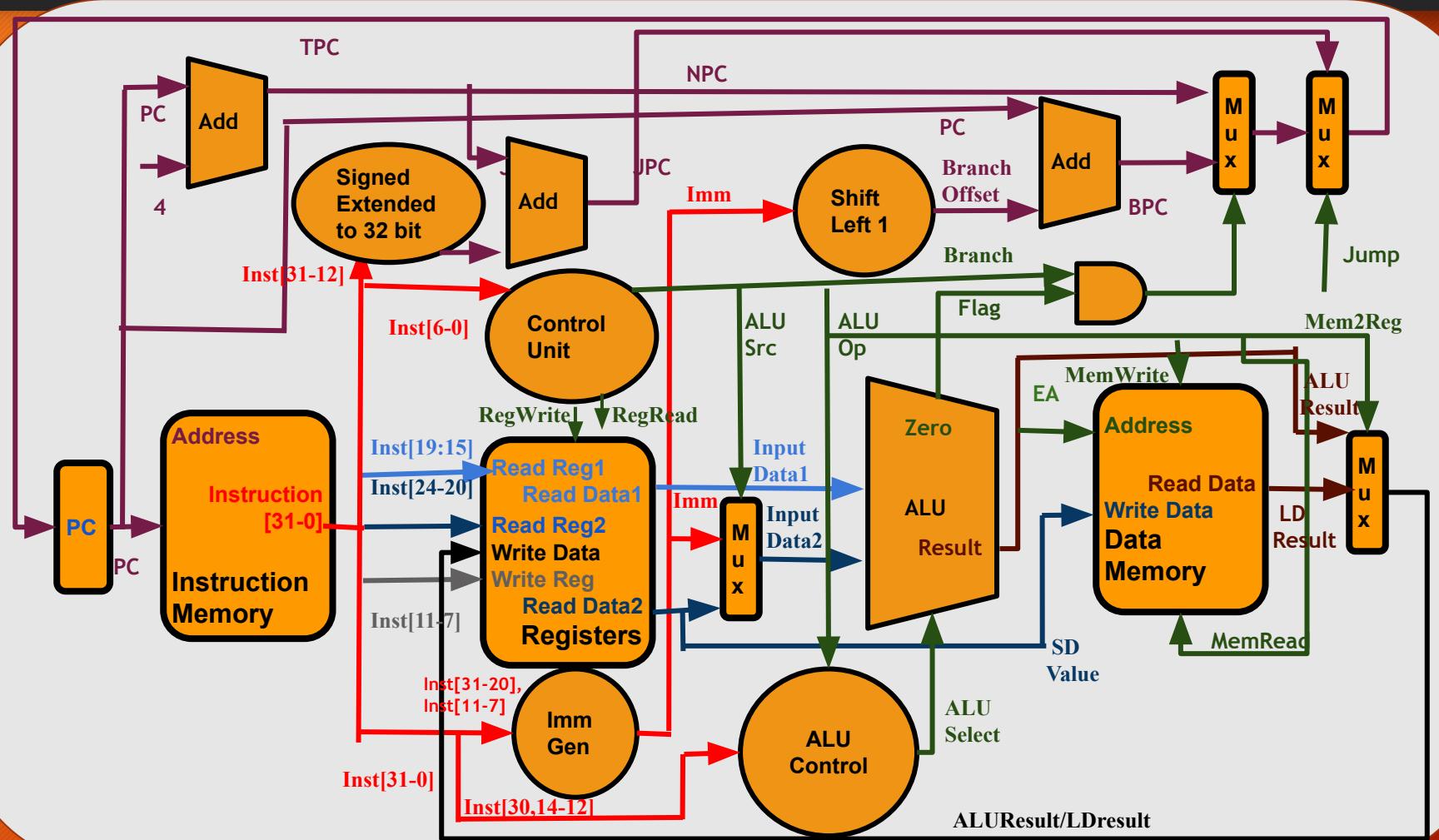
Pipeline stage/segment: Different components of the hardware that work in parallel on different instruction

Goal of pipeline: Overlap instructions and maximize throughput and reduce the delay between the two registers.



Critical Path Pipelined System: The gate delays of the critical path between two intermediate register accesses.

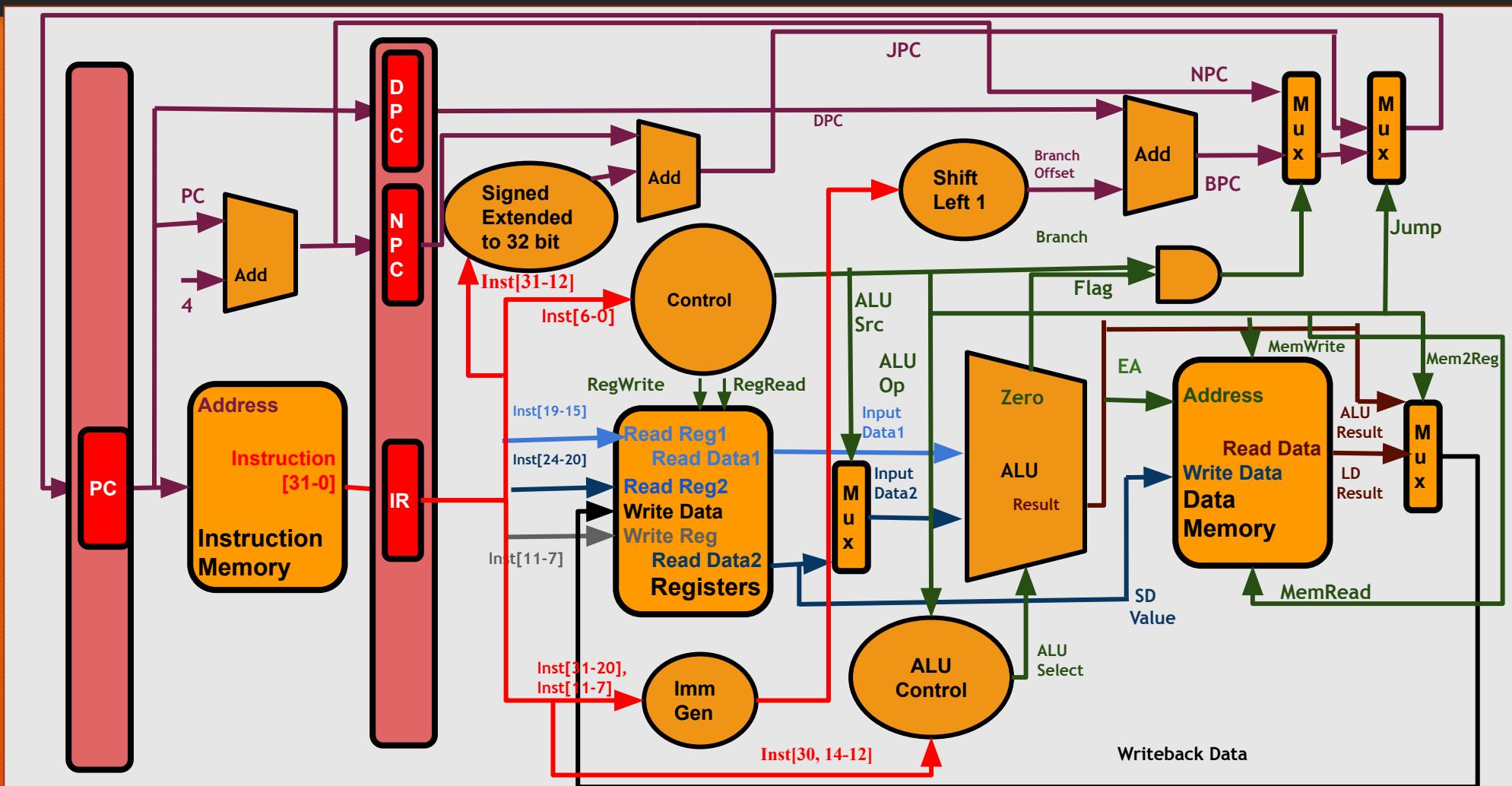
Single Cycle Computer Review



Control Path: The signals that drive the computation

Data Path: The signals that helps flow of the data.

2-Stage Instruction Pipeline: Design



2-Stage Instruction Pipeline: Implementation

Instruction Fetch

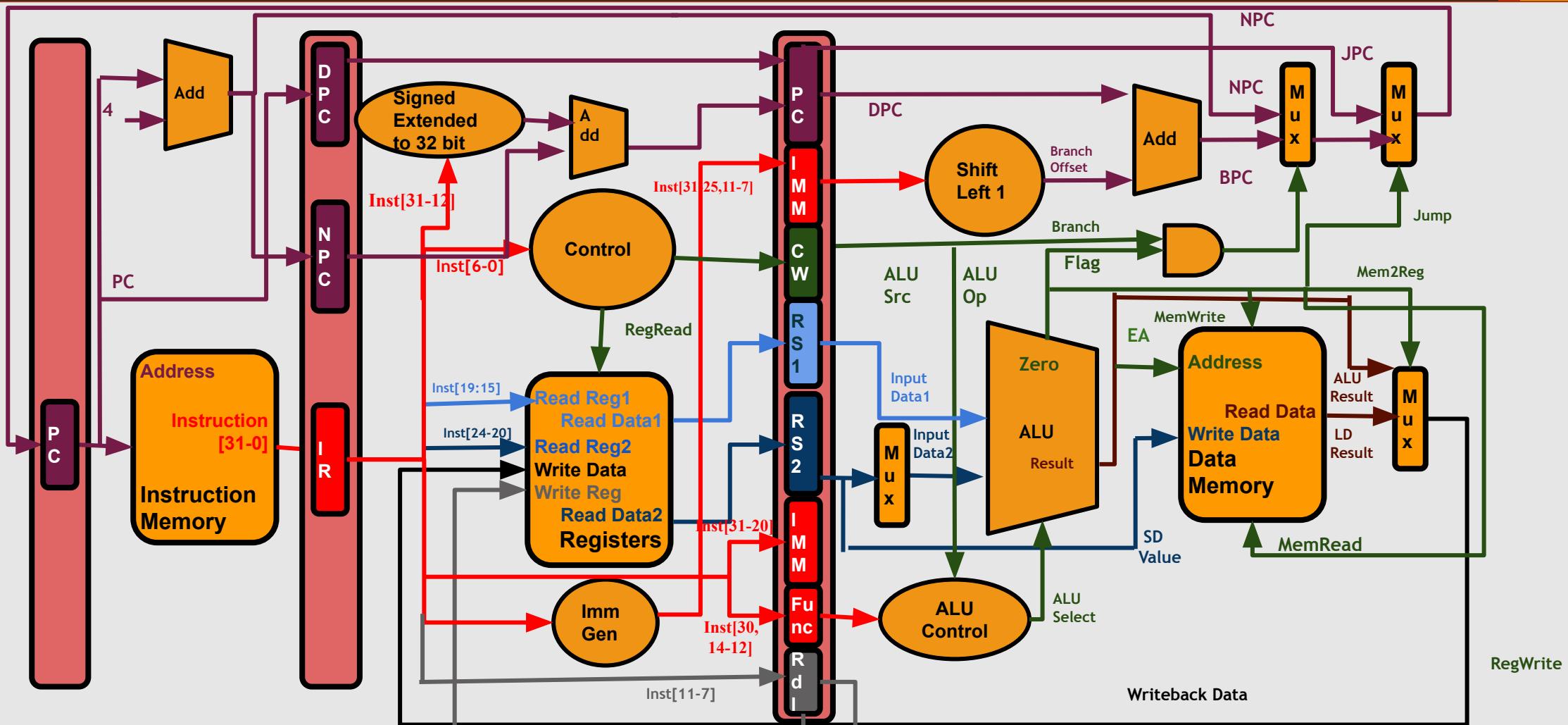
- **IFID.IR = IM[PC]**
- **IFID.DPC = PC**
- **IFID.NPC = PC + 4**

Instruction Execution

- **JPC=IFID.NPC +
SignedExtend(IFID.IR[31-12])**
- **imm = IFID.IR[31-20]**
- **opcode = IFID.IR[6-0]**
- **func3 = IFID.IR[14-12]**
- **rsl1 = IFID.IR[19-15]**
- **rsl2 = IFID.IR[24-20]**
- **rd = IFID.IR[11-7]**
- **CW = controller(opcode)**
- If CW.RegRead Then **rs1 = GPR[rsl1]**
- If CW.ALUSrc
 - Then If CW.RegRead Then **rs2 = imm**
 - Else If CW.RegRead Then **rs2 = GPR[rsl2]**

- **ALUSelect = ALUControl(CW.ALUOp, imm)**
- **ALUresult = ALU(CW.ALUSelect, rs1, rs2)**
- **ALUZeroFlag = (rs1 == rs2)**
- If CW.MemWrite Then
 - **DM[ALUResult] = rs2**
- If CW.MemRead Then
 - **LDResult = DM[ALUResult]**
- **BPC = (imm << 1) + IFID.DPC**
- If CW.Branch and ALUZeroFlag
 - Then **PC = BPC**
 - Else **PC = PC + 4**
- If CW.Jump Then **PC = JPC**
- If CW.RegWrite Then
 - If CW.Mem2Reg
 - Then **GPR[rd] = LDresult**
 - Else **GPR[rd] = ALUresult**

3 Stage Instruction Pipeline: Design



3-Stage Instruction Pipeline: Implementation

Instruction Fetch

- **IFID.IR = IM[PC]**
- **IFID.DPC = PC**
- **IFID.NPC = PC + 4**

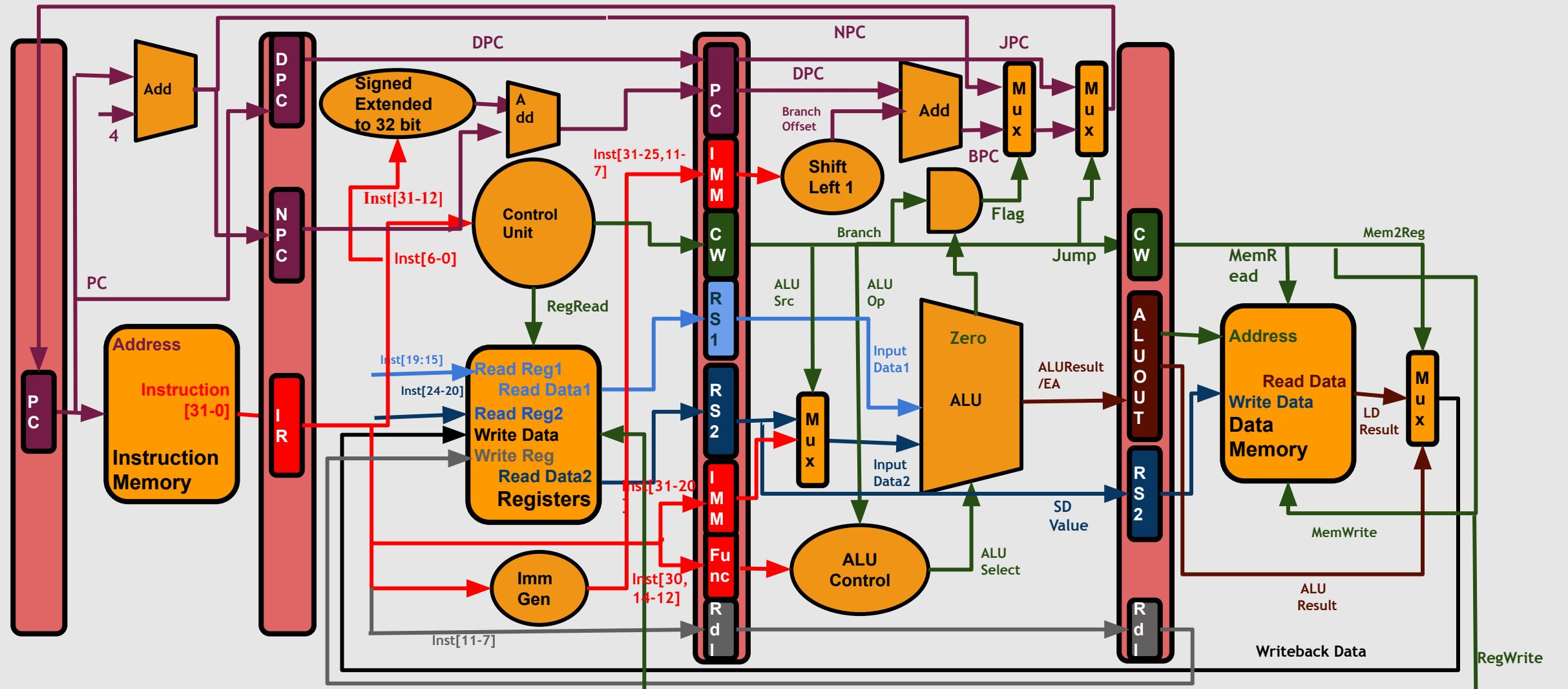
Instruction Decode

- **IDEK.JPC=IFID.NPC + SignedExtend(IFID.IR[31-12])**
- **IDEK.DPC = IFID.DPC**
- **IDEK.imm = IFID.IR[31-20]**
- **IDEK.func = IFID.IR[14-12]**
- **IDEK.rdl = IFID.IR[11-7]**
- **IDEK.CW = controller(IFID.IR[6-0])**
- If IDEK.CW.RegRead Then
 - IDEK.rs1 = GPR[IFID.IR[19-15]]**
- If IDEK.CW.ALUSrc
 - Then If IDEK.CW.RegRead Then
IDEK.rs2 = IFID.IR[31-20]
 - Else If IDEK.CW.RegRead Then
IDEK.rs2 = GPR[IFID.IR[24-20]]

Instruction Execution

- **ALUSelect = ALUControl(IDEK.CW.ALUOp, IDEK.func)**
- **ALUOUT = ALU(ALUSelect, IDEK.rs1, IDEK.rs2)**
- **ALUZeroFlag = (IDEK.rs1 == IDEK.rs2)**
- If IDEK.CW.MemWrite Then
 - DM[ALUOUT] = IDEK.rs2**
- If IDEK.CW.MemRead Then
 - LDOUT = DM[ALUOUT]**
- **BPC = (IDEK.imm << 1) + IDEK.DPC**
- If IDEK.CW.Branch and ALUZeroFlag
 - Then PC = BPC
 - Else PC = PC + 4
- If IDEK.CW.Jump Then PC = IDEK.JPC
- If IDEK.CW.RegWrite Then
 - If IDEK.CW.Mem2Reg
 - Then **GPR[IDEK.rdl] = LDOUT**
 - Else **GPR[IDEK.rdl] = ALUOUT**

4 Stage Instruction Pipeline: Design



4-Stage Instruction Pipeline: Implementation

Instruction Fetch

- $IFID.IR = IM[PC]$
- $IFID.DPC = PC$
- $IFID.NPC = PC + 4$

Instruction Decode

- $IDEX.JPC = IFID.NPC + \text{SignedExtend}(IFID.IR[31-12])$
- $IDEX.DPC = IFID.DPC$
- $IDEX.imm = IFID.IR[31-20]$
- $IDEX.func = IFID.IR[14-12]$
- $IDEX.Rdl = IFID.IR[11-7]$
- $IDEX.CW = \text{controller}(IFID.IR[6-0])$
- If $IDEX.CW.RegRead$ Then
 - $IDEX.rs1 = GPR[IFID.IR[19-15]]$
- If $IDEX.CW.ALUSrc$
 - Then If $IDEX.CW.RegRead$ Then
 $IDEX.rs2 = IFID.IR[31-20]$
 - Else If $IDEX.CW.RegRead$ Then
 $IDEX.rs2 = GPR[IFID.IR[24-20]]$

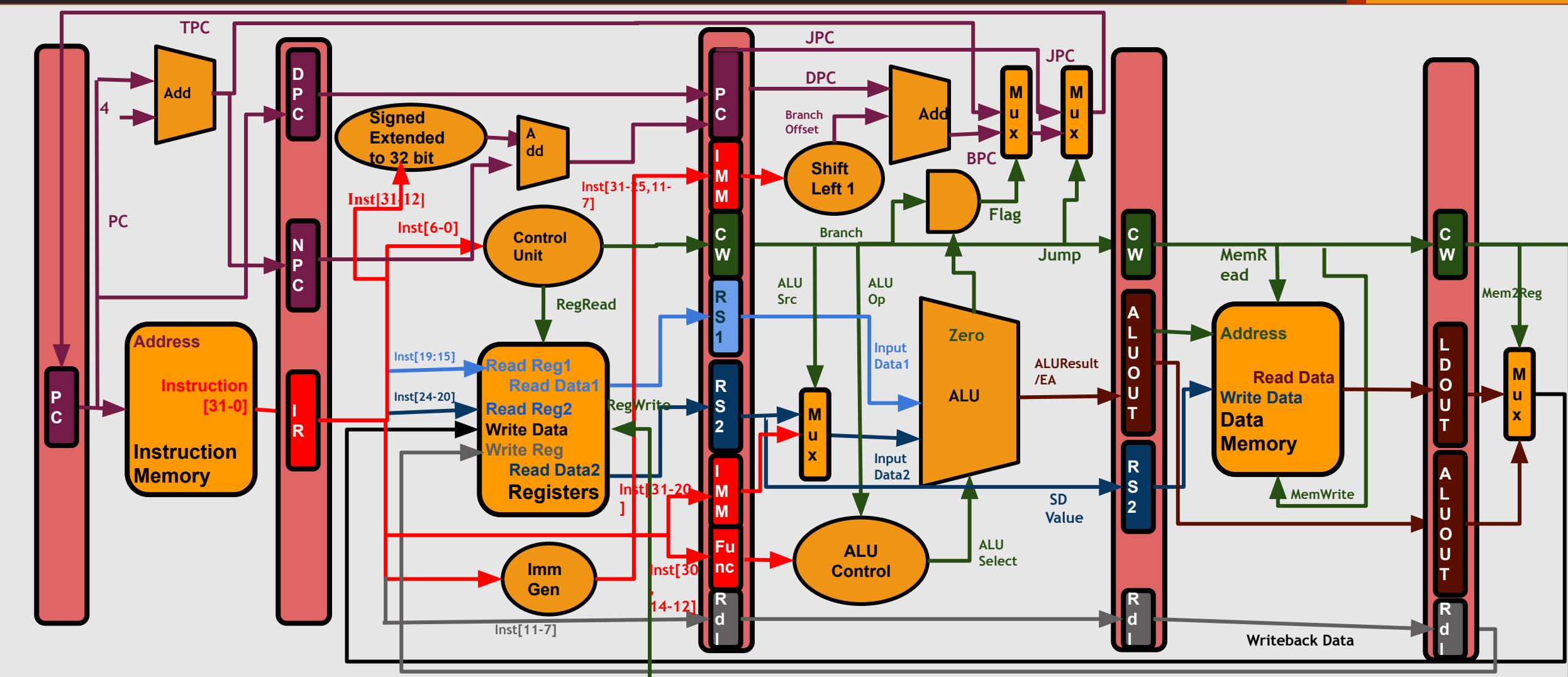
Instruction Execution

- $ALUSelect = ALUControl(IDEX.CW.ALUOp, IDEX.func)$
- $EXMO.ALUOUT = ALU(ALUSelect, IDEX.rs1, IDEX.rs2)$
- $ALUZeroFlag = (IDEX.rs1 == IDEX.rs2)$
- $EXMO.CW = IDEX.CW$
- $EXMO.rs2 = IDEX.rs2$
- If $IDEX.CW.Branch$ and $ALUZeroFlag$
 - Then $PC = (IDEX.imm << 1) + IDEX.DPC$
 - Else $PC = PC + 4$
- If $IDEX.CW.Jump$ Then $PC = IDEX.JPC$

Memory Access and Register Write

- If $IDEX.CW.MemWrite$ Then
 - $DM[EXMO.ALUOUT] = EXMO.rs2$
- If $IDEX.CW.MemRead$ Then
 - $LDOUT = DM[EXMO.ALUOUT]$
- If $IDEX.CW.RegWrite$ Then
 - If $IDEX.CW.Mem2Reg$
 - Then $GPR[EXMO.Rdl] = LDOUT$
 - Else $GPR[EXMO.Rdl] = EXMO.ALUOUT$

5 Stage Instruction Pipeline: Design



5-Stage Instruction Pipeline: Implementation

Instruction Fetch

- $IFID.IR = IM[PC]$
- $IFID.DPC = PC$
- $IFID.NPC = PC + 4$

Instruction Decode

- $IDEK.JPC = IFID.NPC + \text{SignedExtend}(IFID.IR[31-12])$
- $IDEK.DPC = IFID.DPC$
- $IDEK.imm = IFID.IR[31-20]$
- $IDEK.func = IFID.IR[14-12]$
- $IDEK.rdl = IFID.IR[11-7]$
- $IDEK.CW = \text{controller}(IFID.IR[6-0])$
- If $IDEK.CW.RegRead$ Then
 - Then If $IDEK.CW.RegRead$ Then
 $IDEK.rs2 = IFID.IR[31-20]$
 - Else If $IDEK.CW.RegRead$ Then
 $IDEK.rs2 = GPR[IFID.IR[24-20]]$
- If $IDEK.CW.ALUSrc$
 - Then If $IDEK.CW.RegRead$ Then
 $IDEK.rs2 = IFID.IR[31-20]$
 - Else If $IDEK.CW.RegRead$ Then
 $IDEK.rs2 = GPR[IFID.IR[24-20]]$

Instruction Execution

- $ALUSelect = ALUControl(IDEK.CW.ALUOp, IDEK.func)$
- $EXMO.ALUOUT = ALU(ALUSelect, IDEK.rs1, IDEK.rs2)$
- $ALUZeroFlag = (IDEK.rs1 == IDEK.rs2)$
- $EXMO.CW = IDEK.CW$
- $EXMO.rs2 = IDEK.rs2$
- If $IDEK.CW.Branch$ and $ALUZeroFlag$
 - Then $PC = (IDEK.imm << 1) + IDEK.NPC$
 - Else $PC = PC + 4$
- If $IDEK.CW.Jump$ Then $PC = IDEK.JPC$

Memory Access

- If $EXMO.CW.MemWrite$ Then
 - $DM[EXMO.ALUOUT] = EXMO.rs2$
- If $EXMO.CW.MemRead$ Then
 - $MOWB.LDOUT = DM[EXMO.ALUOUT]$
- $MOWB.ALUOUT = EXMO.ALUOUT$
- $MOWB.CW = EXMO.CW$

Register Write

- If $MOWB.CW.RegWrite$ Then
 - If $MOWB.CW.Mem2Reg$
 - Then $GPR[MOWB.rdl] = MOWB.LDOUT$
 - Else $GPR[MOWB.rdl] = MOWB.ALUOUT$

Summary Simple Five Stages of Computation

Instruction Fetch

- Program counter
- Instruction Access

Instruction Decode

- Determine operand, opcode, addressing mode, control word
- Read register file

Execution

- Memory reference: Effective Address Calculation
- Register-Register ALU Instruction
- Register-Immediate ALU Instruction
- Conditional Branch

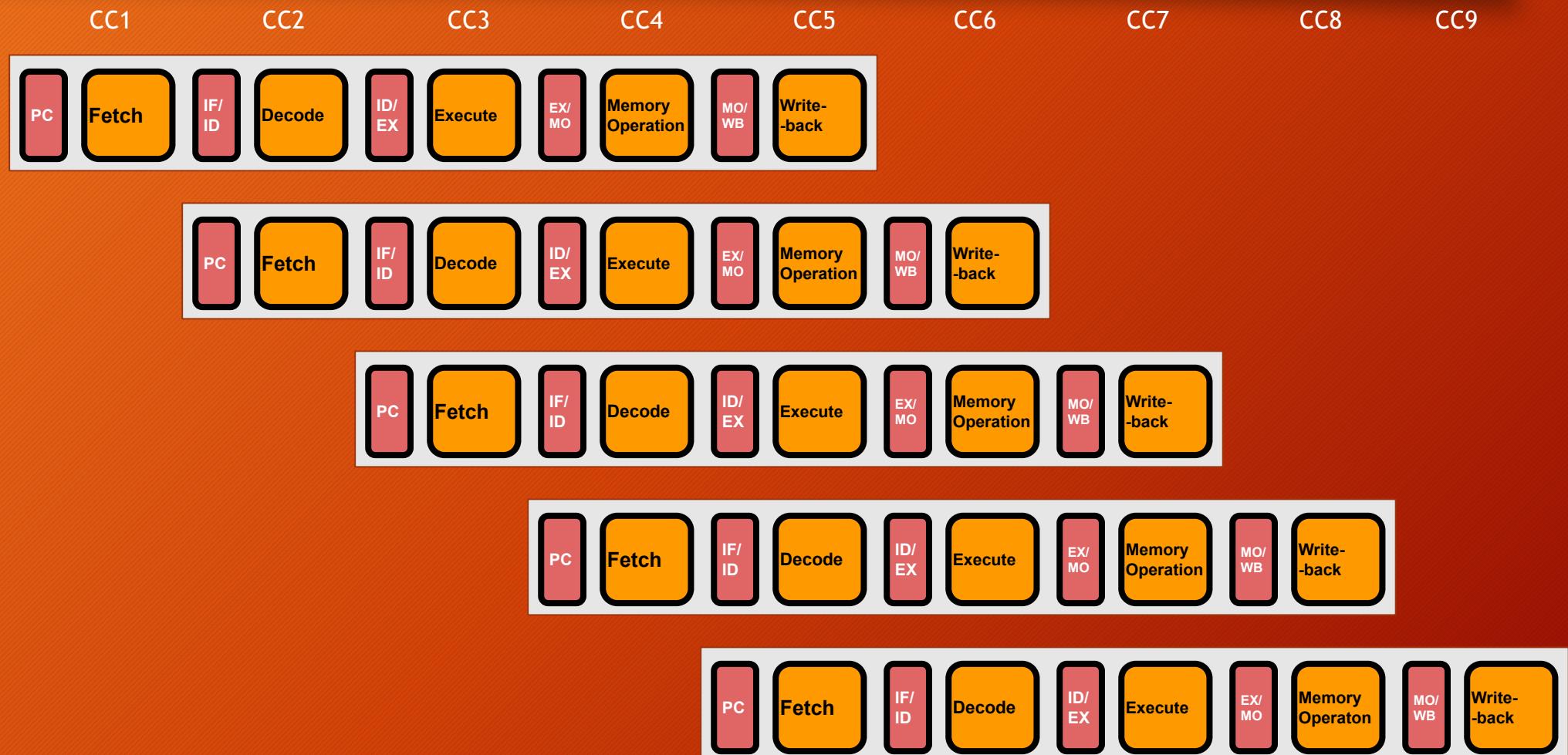
Memory Access

- Data access: Read for Load and Write for Store

Register Writeback

- Write register files for ALU and Load Instructions

Instruction Execution in 5 Stage Instruction Pipeline



Performance of a K-Stage Pipeline

Non-Pipelined System

- t_r = Delay of intermediate registers
- Latency = $t_{np} + t_r$
- Number of cycles for n-instructions = n
- Throughput = IPC = 1
- Frequency = $f = 1 / \text{Latency}$

N-Stage Pipelined System

- Latency per stage = $t_s = (t_{np}/k) + t_r$
- t_r = Delay of intermediate registers
- Total Latency = $kt_s = t_{np} + kt_r$
- Number of cycles for n instructions = $n+k-1$
- Throughput = IPC = $n/(n + k - 1)$
IPC close to 1 for n very large than k
- Frequency = $f = 1 / \text{Latency per stage}$
- Total time = $n / (\text{IPC} * f)$

Problems with Pipelined Systems

Frequency

$$f = 1/t_p$$

Dynamic Power

$$P = 0.5CV^2f$$

Dynamic Voltage and Frequency Scaling (DVFS)

Heat Sinks

Speedup

Can it be 1?

1

Structural Hazards

2

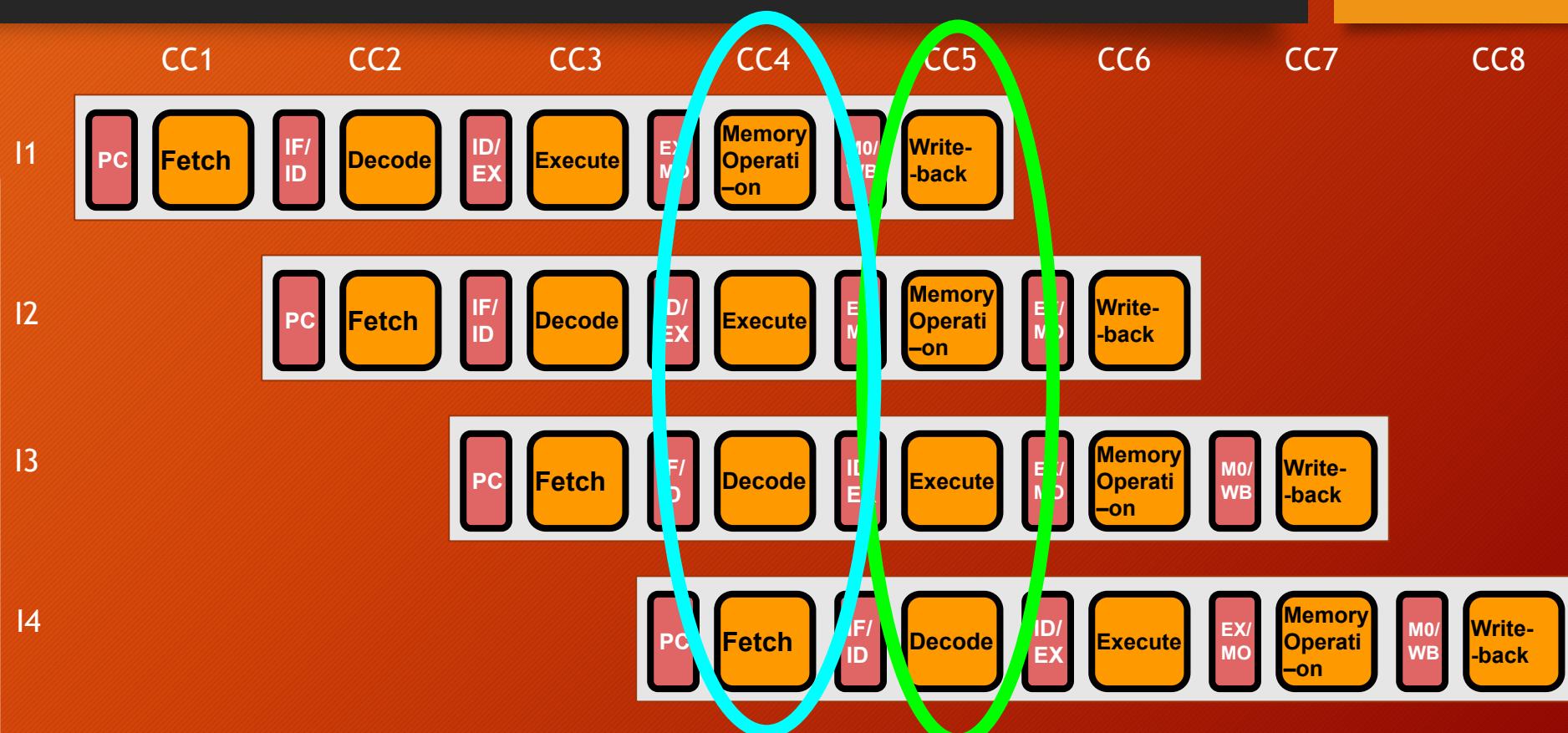
Data Hazards

3

Control Hazards

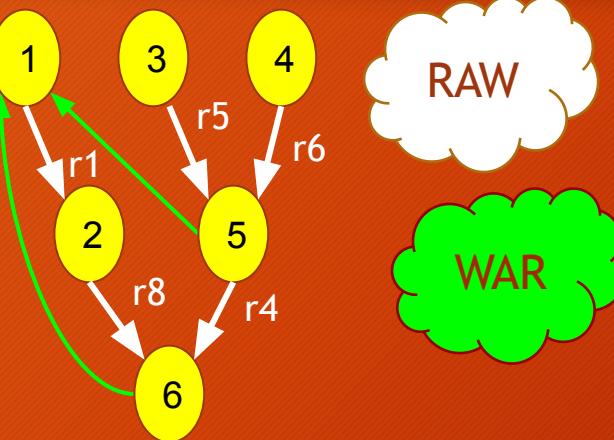
Structural Hazards

- Structural Hazard occurs when two stages of a single pipeline access the same resource.
- Reading and writing into register file in the same cycle.
- Access memory for instruction and data in the same cycle.



Data Flow Analysis (Graph Representation) and Data Hazards

```
(1) r1 = r4 / r7
(2) r8 = r1 + r2
(3) r5 = r5 + 1
(4) r6 = r6 - r3
(5) r4 = r5 + r6
(6) r7 = r8 * r4
```



```
(1) r1 = r4 / r7
(2) r8 = r1 + r2
(3) r5 = r5 + 1
(4) r6 = r6 - r3
(5) r4 = r5 + r6
(6) r7 = r8 * r4
```

- Data flow analysis is done via data flow graphs
- Nodes are instructions
- Edges are register data

- No WAW and WAR for pipelined system
- Neither we are parallel nor we go back in time!

1
Read after Write (RAW) Hazards

2
Write after Read (WAR) Hazards

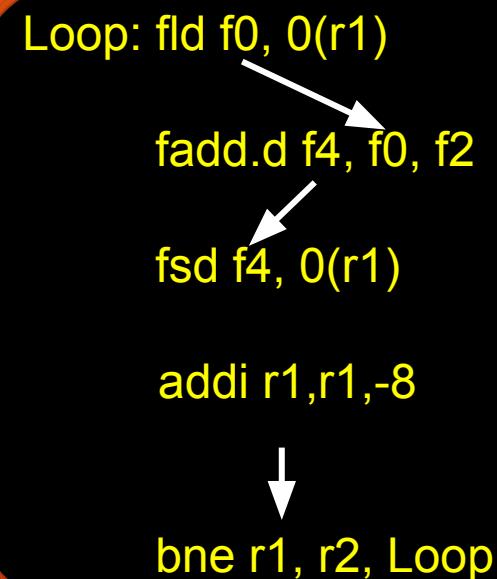
3
Write after Write (WAW) Hazards

- Data dependency can convey three things:
 - Possibility of hazard
 - Orders in which results are calculated
 - Upper-bound on how much parallelism can possibly be exploited

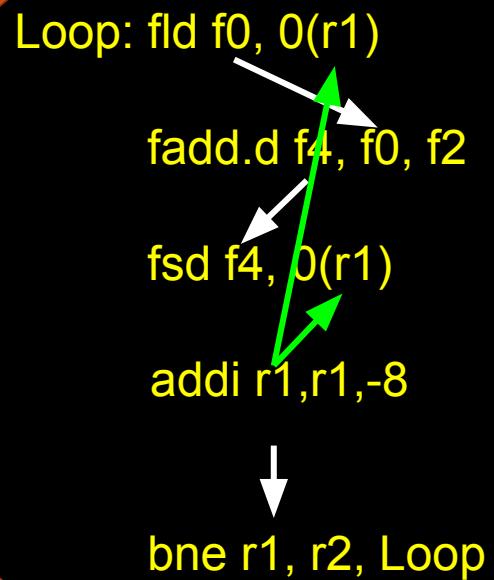
- Data dependent values can flow between instructions via:
 - Registers: Easy to find dependency
 - Memory location: Difficult to detect dependency. Why? -> Effective address calculation is data-dependent

True Data Dependency

- ❑ True Dependency or Read-after-write hazard
- ❑ Instruction i produces a result that may be used by instruction j
- ❑ Instruction j is data-dependent on instruction k , and k is data-dependent on instruction i .



Name Dependency: Anti-dependency



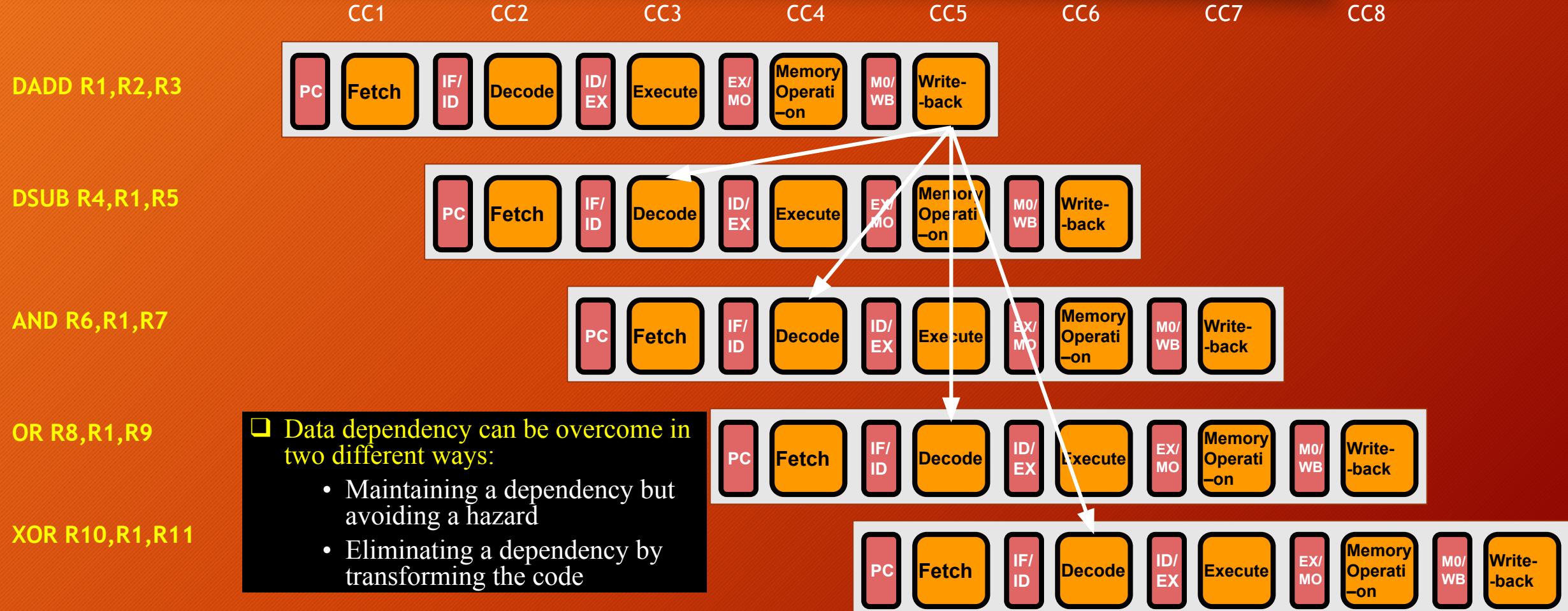
- ❑ **Anti-dependency:** i precedes j but instruction j writes a register or memory that instruction i reads
- ❑ No problem with in-order pipeline execution.
- ❑ What is program order? What is in-order?
- ❑ What will happen if we start executing instructions in parallel or out-of-order.

Name Dependency: Output Dependency

```
fld f1, 0(r1)  
fadd.d f3, f1, f2  
fadd.d f1, f4, f5
```

- ❑ **Output-dependency:** i precedes j but instruction j writes a register or memory that instruction i writes
- ❑ No anti-dependency or true dependency exists between I0 and I2
- ❑ What about latency of I2 v I0? L1 latency is around 4 cycles and integer adder takes 1 cycle
- ❑ What will happen when we start executing them in parallel?

Data Hazard: Example



Control Hazard

Loop: `fld f0, 0(r1)`

`fadd.d f4, f0, f2`

`fsd f4, 0(r1)`

`addi r1,r1,-8`

`bne r1, r2, Loop`

`add f4, f4, f2`

`bne r1, r2, elsepart`

`fld f0, 0(r1)`

`elsepart: fld f0, 0(r2)`

- Until we **execute**, the outcome of the branch is not known.
- The PC goes-on incrementing and pulls unwanted instructions into the pipeline.

Branch Not Taken

CC1

CC2

CC3

CC4

CC5

CC6

CC7

CC8

CC9

Branch Instruction PC



PC + 4



PC + 8



PC + 12



PC + 16



Control Hazard: Example

Branch Taken

CC1

CC2

CC3

CC4

CC5

CC6

CC7

CC8

CC9

Branch Instruction PC



PC + 4



Can't
Perform
these actions

BPC



BPC + 4

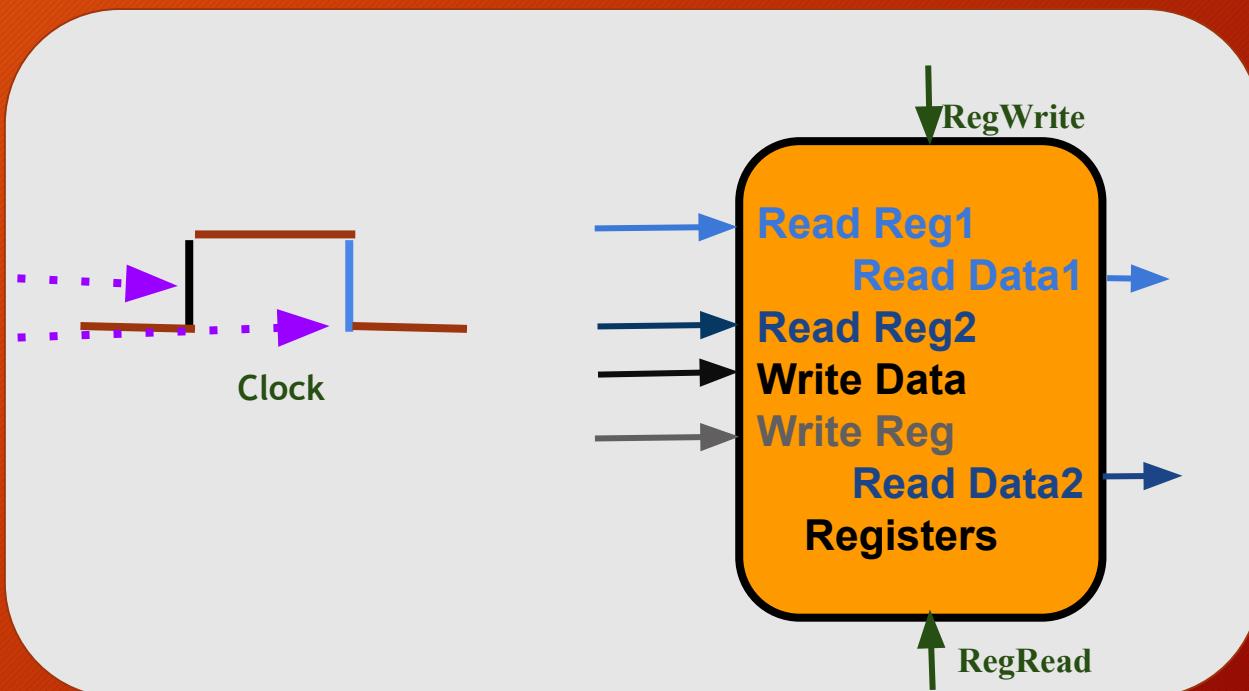


BPC + 8



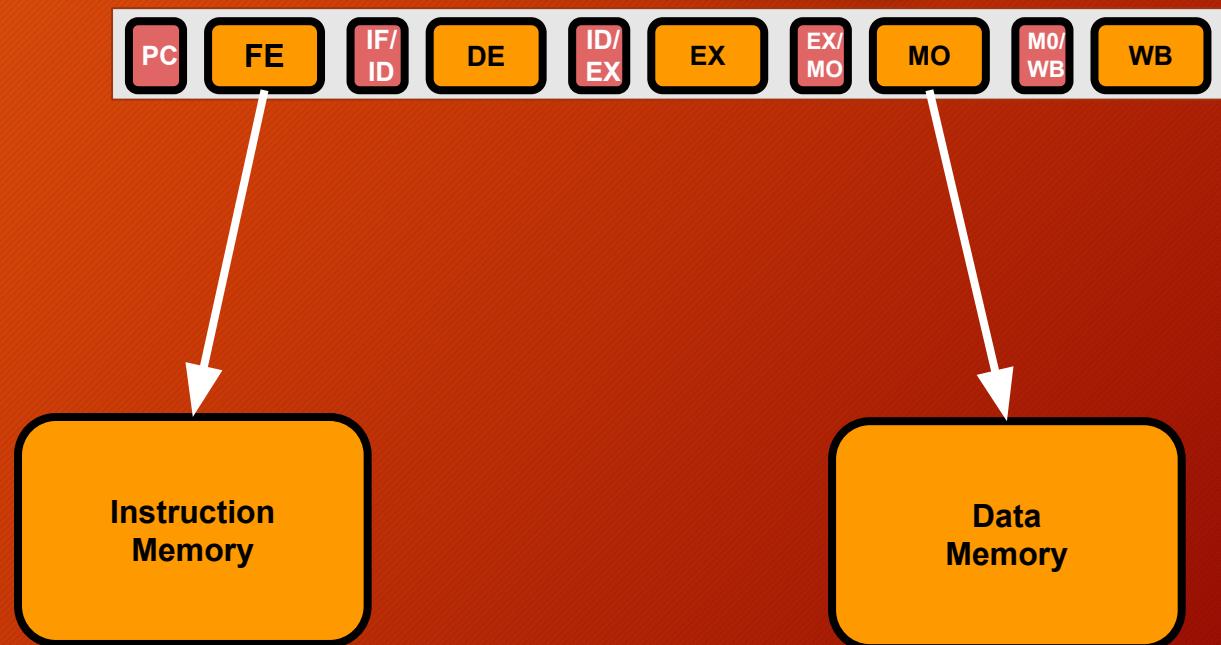
Structural Hazard Solution for Register Read/Write in Same Cycle

- Edge-triggered flip-flops are used to design registers.
- Write in the positive edge of a clock.
- Read in the negative edge of a clock.
- Writeback and decode can happen in a single cycle.
- Decoding has long latency. Hence, reading at negative edge will allow the circuit to complete all the gate operations before reading registers.



Structural Hazard Solution for Accessing Instruction and Data Memory

- **Harvard Architecture:** Use separate instruction and data memory.
- Fetch unit interacts with the instruction memory
- Memory stage interacts with the data memory
- Von-Neumann discussed about a single memory module.
- Do you remember the memory view of an application? Different segments of an applications: Code/Data/Heap/Stack.
- If we have different memory unit, we can optimize different units separately.



Data and Control Hazard Solution: Pipeline Interlock Concept



Pipeline Interlock Detection and Resolution

- Add ready bit in the output general purpose register to detect true data dependency
- The ready bit acts as a lock to reserve the data for the instruction that will update the output GPR.
- Check the ready bit while decoding and copying the input registers.
- Mark the bit busy for the output register while decoding.
- Mark the bit ready for the output register after writeback.
- Use signals to communicate between the pipeline stages.

ADD R1, R2, R3

ADD R1, R4, R5

ADD R6, R1, R7

Should we use ready bit or instruction id which is writing into the output last?

Implementation of Guards, Pipeline Interlock, Correcting Branches and Jumps

- If IFID.stall or !PC.Valid Then return
- IFID.IR = IM[PC.value]
- IFID.DPC = PC
- NPC = PC.value + 4
- IFID.Valid = true

- If IDEX.stall or !IFID.Valid Then return
- IDEX.JPC=IFID.NPC + SignedExtend(IFID.IR[31-12])
- IDEX.DPC = IFID.DPC
- IDEX.imm = IFID.IR[31-20]
- IDEX.func = IFID.IR[14-12]
- IDEX.rdl = IFID.IR[11-7]
- IDEX.CW = controller(IFID.IR[6-0])
- If IDEX.CW.RegRead Then
- If GPR[IFID.IR[19-15]].ins = -1
- Then IDEX.rs1 = GPR[IFID.IR[19-15]].value
- Else IFID.stall = true return
- If IDEX.CW.ALUSrc
- Then If IDEX.CW.RegRead Then
- IDEX.rs2 = IFID.IR[31-20]
- Else If IDEX.CW.RegRead Then
 - If GPR[IFID.IR[24-20]].ins = -1
 - Then IDEX.rs2 = GPR[IFID.IR[24-20]].value
 - Else IFID.stall = true return
- GPR[INDEX.rdl].ins = CW.ins
- IFID.stall = false
- INDEX.Valid = true

- If EXMO.stall or !IDEX.Valid Then return

- ALUSelect = ALUControl(IDEX.CW.ALUOp, IDEX.func)
- EXMO.ALUOUT = ALU(ALUSelect, IDEX.rs1, IDEX.rs2)
- ALUZeroFlag = (IDEX.rs1 == IDEX.rs2)
- EXMO.CW = IDEX.CW
- If IDEX.CW.Branch
- Then ALUZeroFlag
 - Then PC = (IDEX.imm << 1) + IDEX.DPC
 - IFID.Valid = false
 - IDEX.Valid = false
 - PC.Valid = true
- If IDEX.CW.Jump
- Then PC = IDEX.JPC
- IFID.Valid = false
- IDEX.Valid = false
- PC.Valid = true
- IDEX.stall = false
- EXMO.Valid = true

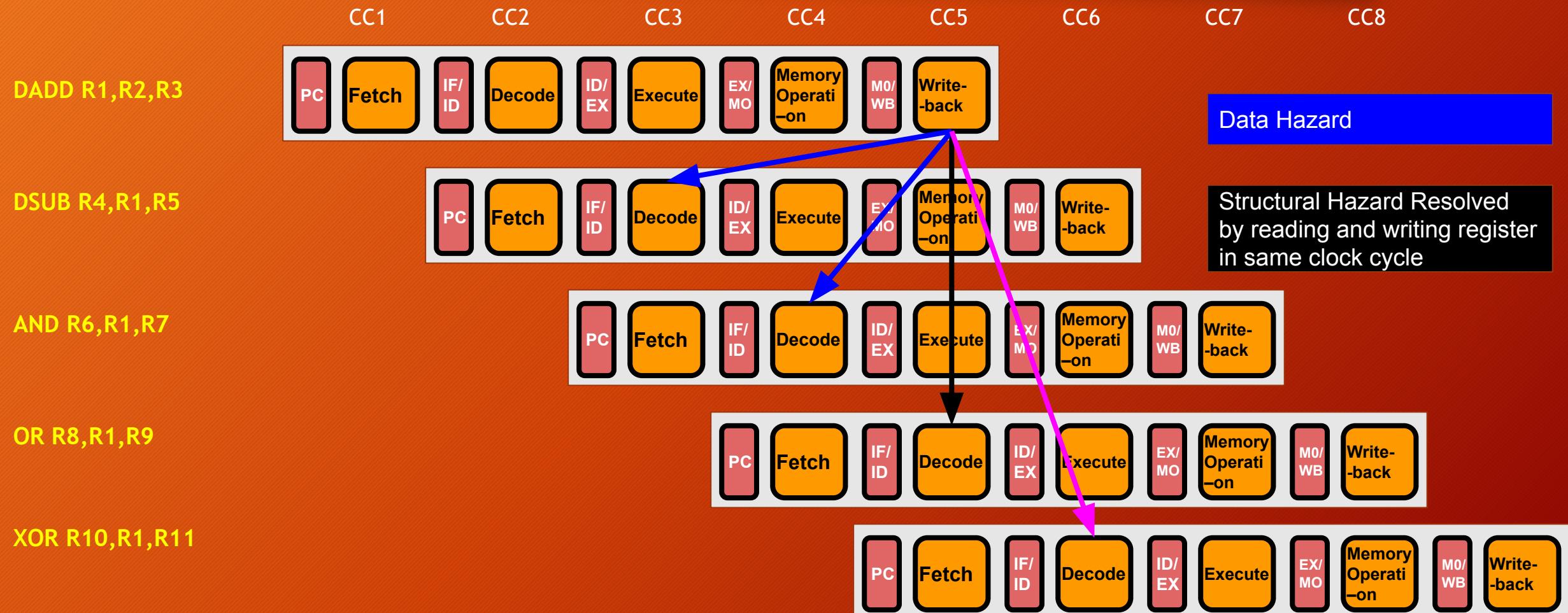
- If MOWB.stall or !EXMO.Valid Then return

- If EXMO.CW.MemWrite Then
 - DM[EXMO.ALUOUT] = IDEX.rs2
- If EXMO.CW.MemRead Then
 - MOWB.LDOUT = DM[EXMO.ALUOUT]
- MOWB.ALUOUT = EXMO.ALUOUT
- MOWB.CW = EXMO.CW
- If IDEX.CW.Branch or IDEX.CW.Jump
- Then PC.value = EXMO.TPC
- PC.Valid = True
- EXMO.stall = false
- MOWB.Valid = true

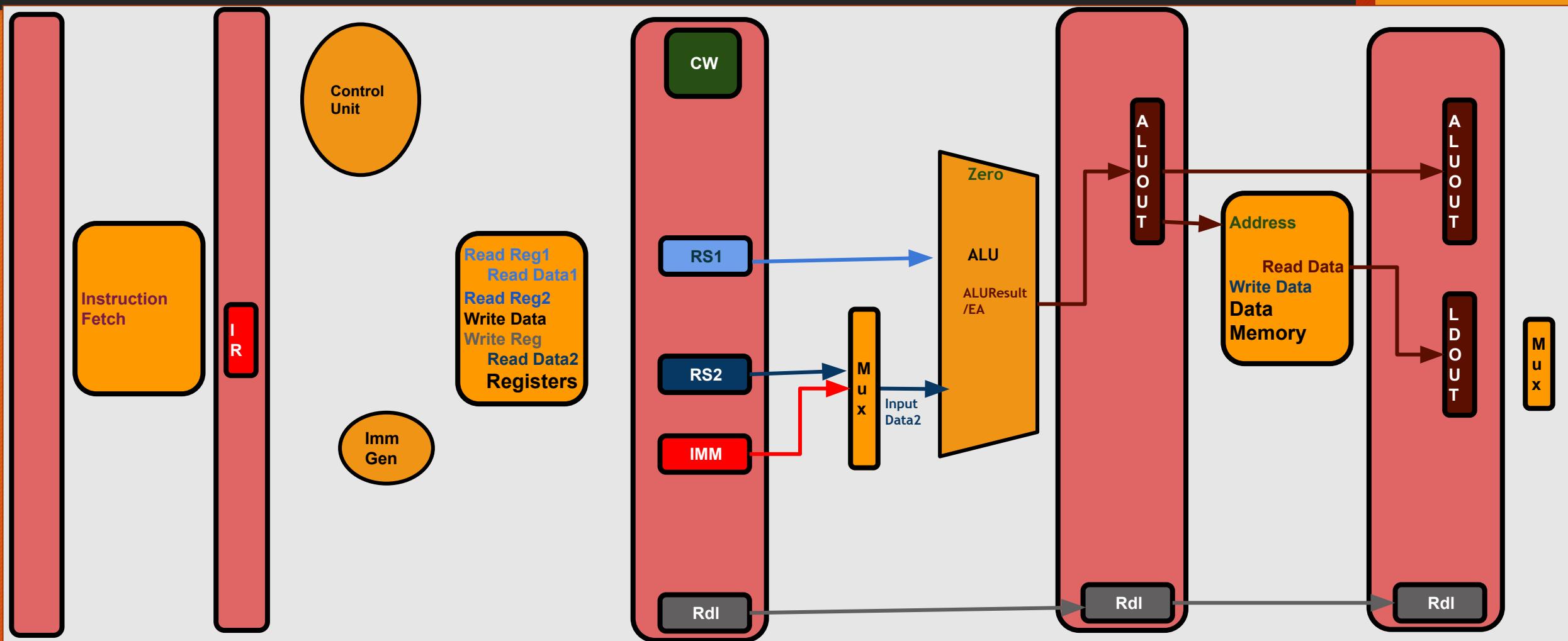
- If !MOWB.Valid Then return

- If !MOWB.CW.RegWrite Then Return
- Else If MOWB.CW.Mem2Reg and GPR[MOWB.rdl].ins = CW.ins
 - Then GPR[MOWB.rdl].value = MOWB.LDOUT
 - Else GPR[MOWB.rdl].value = MOWB.ALUOUT
 - GPR[MOWB.rdl].ins = -1
 - MOWB.stall = false

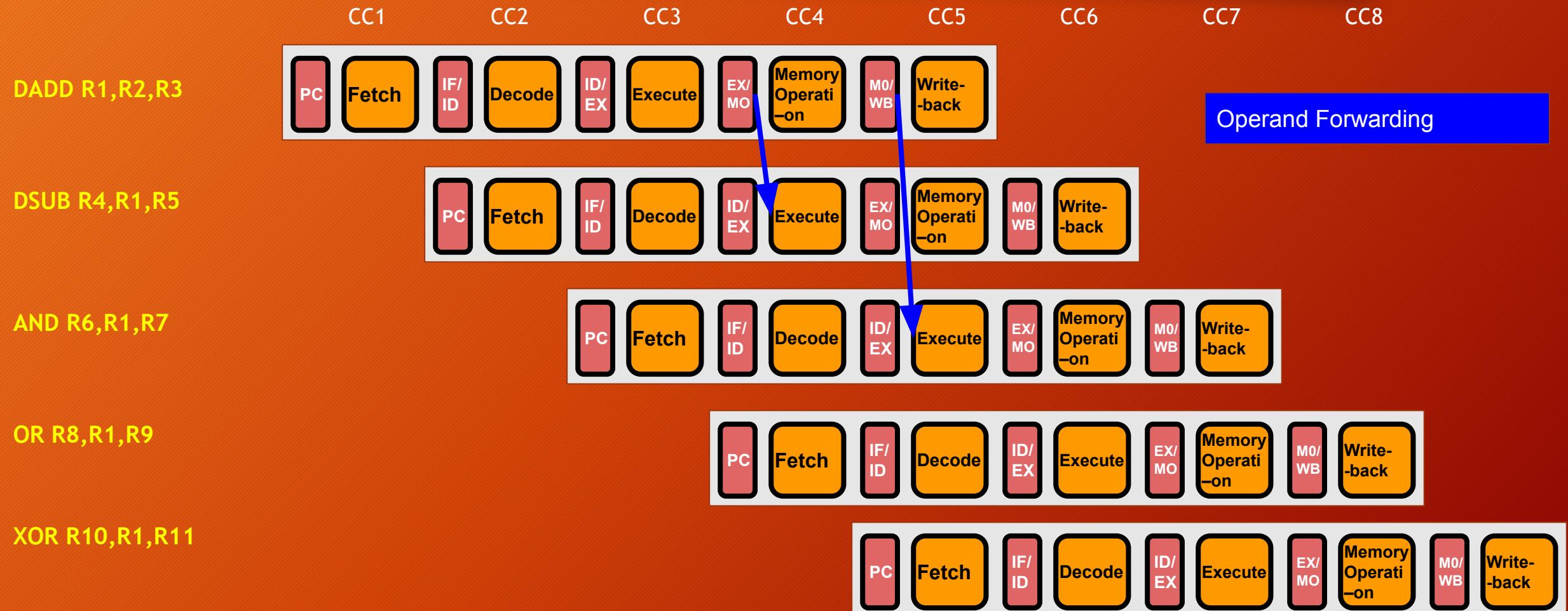
Operand Forwarding: Concept and Example Problem



Review of 5 Stage Pipeline



Operand Forwarding: Concept and Example Solution



Operand Forwarding: Detection and Resolution

Source Pipeline Register Set	Source Opcode	Destination Opcode	Destination for Forwarding	Comparison and Condition
EXMO	R-R ALU or R-I ALU	Any	ALU Input 1	EXMO.rdl == IDEX.rsl1
EXMO	R-R ALU or R-I ALU	R-R ALU	ALU Input 2	EXMO.rdl == IDEX.rsl2
MOWB	Except SD	Any	ALU Input 1	MOWB.rdl == IDEX.rsl1
MOWB	Except SD	R-R ALU	ALU Input 2	MOWB.rdl == IDEX.rsl2

Execution Stage

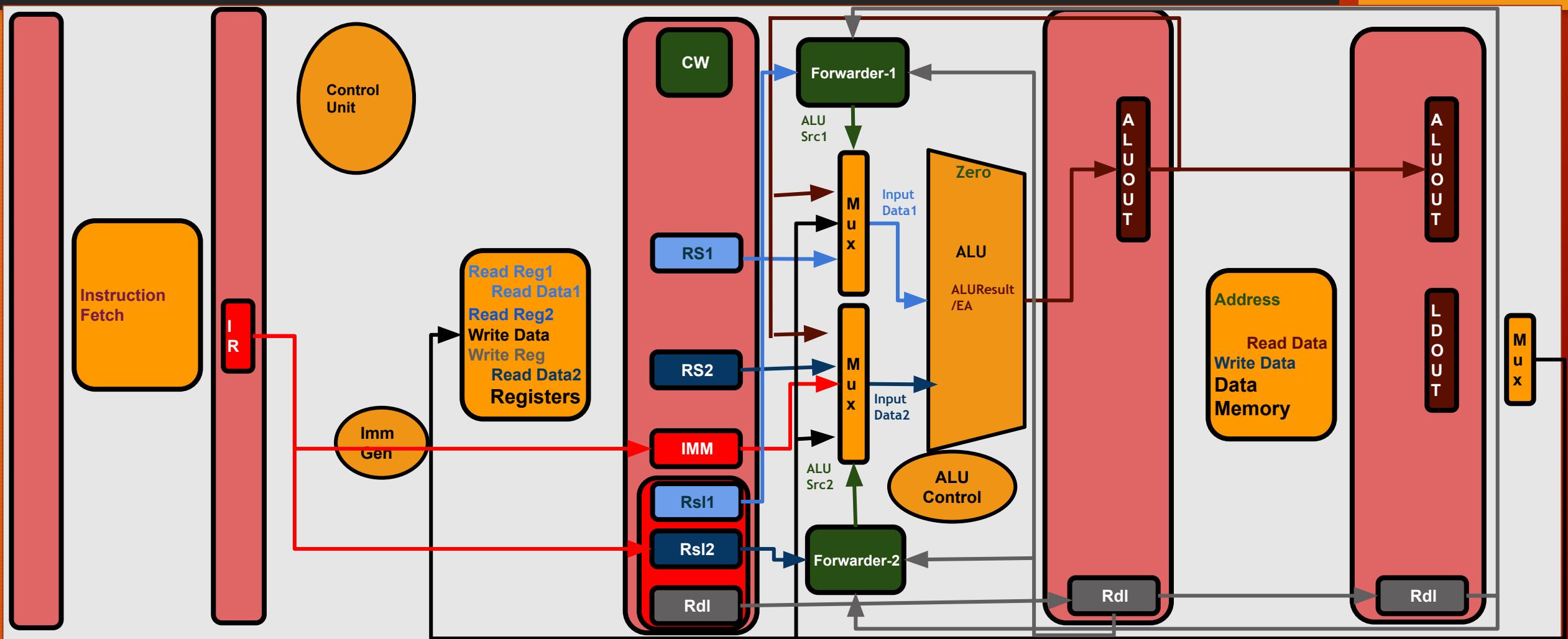
- If EXMO.rdl = IDEX.rsl1
 - then input1 = EXMO.ALUOutput
- Else If MOWB.rdl = IDEX.rsl1
 - then if MOWB.CW.MemRead
 - then input1 = MOWB.LDOutput
 - else input1 = MOWB.ALUOutput
 - Else input1 = IDEX.rs1
- Repeat the same for input2

Something Fishy about 2nd Operand of Store

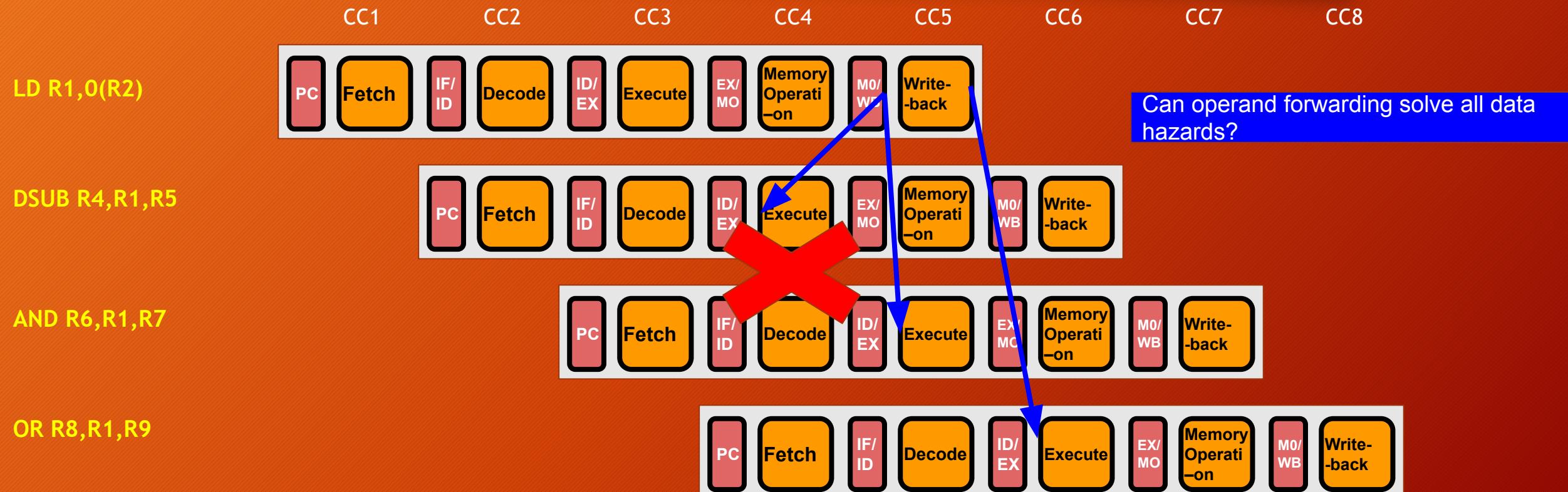
Decode Stage

- If GPR[rsl1].INS = 0
 - then IDEX.rs1 = GPR[rsl1].value
 - IDEX.rsl1 = INVALID
- else IDEX.rsl1 = rsl1
 - IDEX.rs1 = INVALID
- If GPR[rsl2].INS = 0
 - then IDEX.rs2 = GPR[rsl2].value
 - IDEX.rsl2 = INVALID
- else IDEX.rsl2 = rsl2
 - IDEX.rs2 = INVALID
- GPR[rdl].INS = currentIns

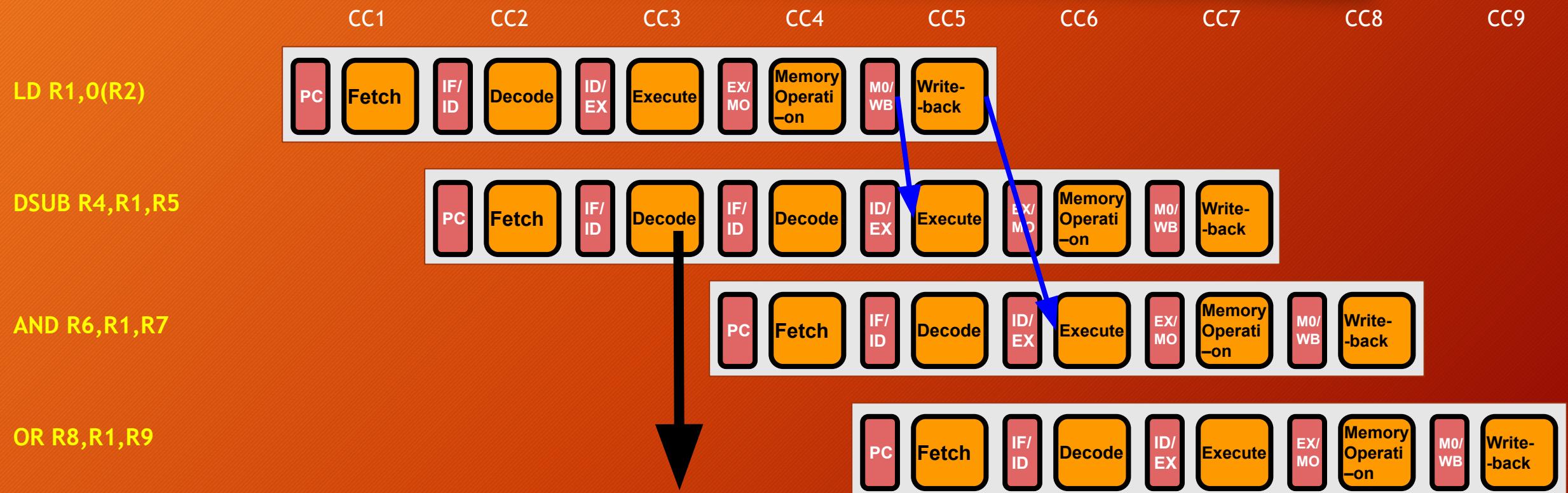
Operand Forwarding: Design



Operand Forwarding: Example-2

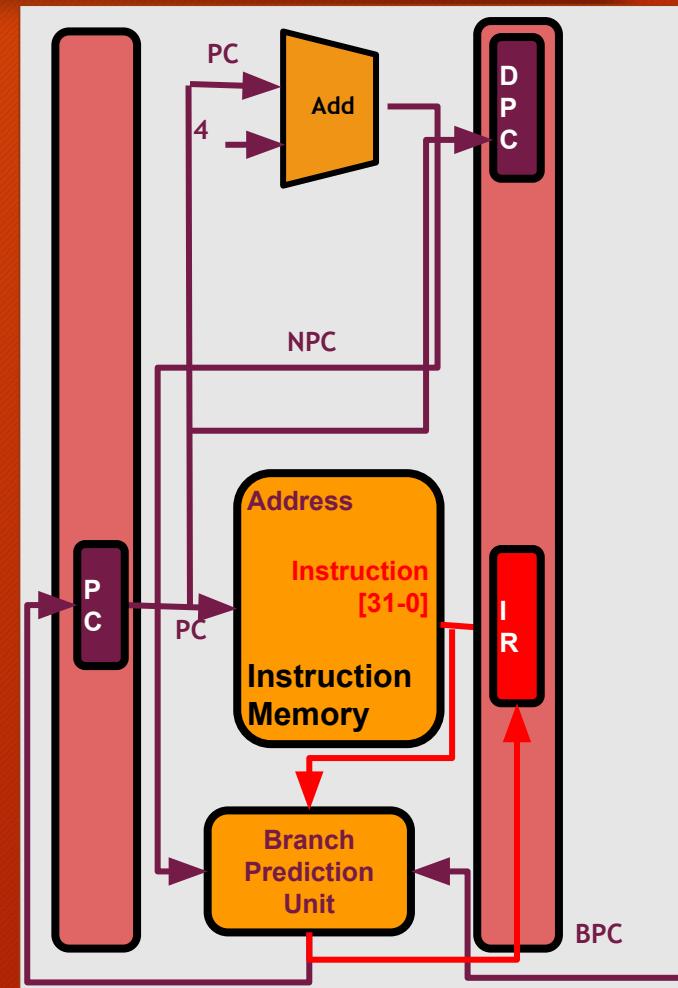


Stall Solves All Hazards!



Handling Pipeline Branch Penalties: Freeze Pipeline on Branch

- **Freeze** -> Don't allow any instructions to enter the pipeline. How?
- Add a circuit to manage the branch instructions in the pipeline to the **fetch unit**. Why to fetch unit?
 - Need a small decoder in the fetch unit to detect that it is a branch instruction. If we wait for decode unit to detect branch, it is too late!
 - Ensure that fetch unit stalls such that no instruction enters the pipeline.
 - Alternative, fetch dummy instruction (NOP) and fool the pipeline from the next cycle until branch is resolved.



Handling Pipeline Branch Penalties: Predict and Flush on Mispredict

- There are four different options:
 - Predict Always Branch Taken
 - Predict Always Branch Not Taken
 - Static Prediction (using compiler's help) for Branch Taken/Not-taken
 - Dynamic Prediction (use hardware-based AI?)
- Flushing involves invalidating the intermediate registers upon a mispredict. Already discussed!

Predicted	Actual Outcome	Action	Hardware Change
Not-taken	Not-taken	Already on-track	No change
Taken	Taken	Already on-track	Determine the target BPC
Not-taken	Taken	Flush and Load BPC	No change
Taken	Not-taken	Flush and Load DPC	Determine the target BPC and saving DPC

Pipeline Stages of Well-Known Intel Processors

Year	Model Name	#Pipelined Stages	Max. Frequency (in MHz)	Process Technology (in nm)
1978	8086	2	5	3000
1982	80186	2	25	3000
1982	80286	3	25	1500
1985	80386	6	33	1500
1989	80486	5	100	1000
1993	Pentium	5	200	800/600/350
1995	Pentium-II	17	450	500/350/250
1999	Pentium-III	15	1400	250/180/130
2000/2	NetBurst (Pentium-4)	20	2000/3466	180/130
2004	NetBurst	31	3800	90, 65

Year	Model Name	#Pipelined Stages	Max. Frequency (in MHz)	Process Technology (in nm)
2006	Intel Core	14	3000	65
2008	Nehalem	20	3600	45
2011	Sandy Bridge	16	4000	32
2012	Ivy Bridge	16	4100	22
2013	Haswell	16	4400	22
2014	Broadwell	16	3700	14
2015	Skylake	16	5200	14
2019	Cascade/Coffee Lake	14/20	3800/4000	14/10
2020	Ice/Cooper Lake	14/12/20	5300/5500/4300	14/7
2022	Raptor Lake	12	6000	7

“

Designing Pipelined System in the
Fundamental Property of Computing Systems!

”

Can we do better?