CPU Design

Debiprasanna Sahoo
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

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

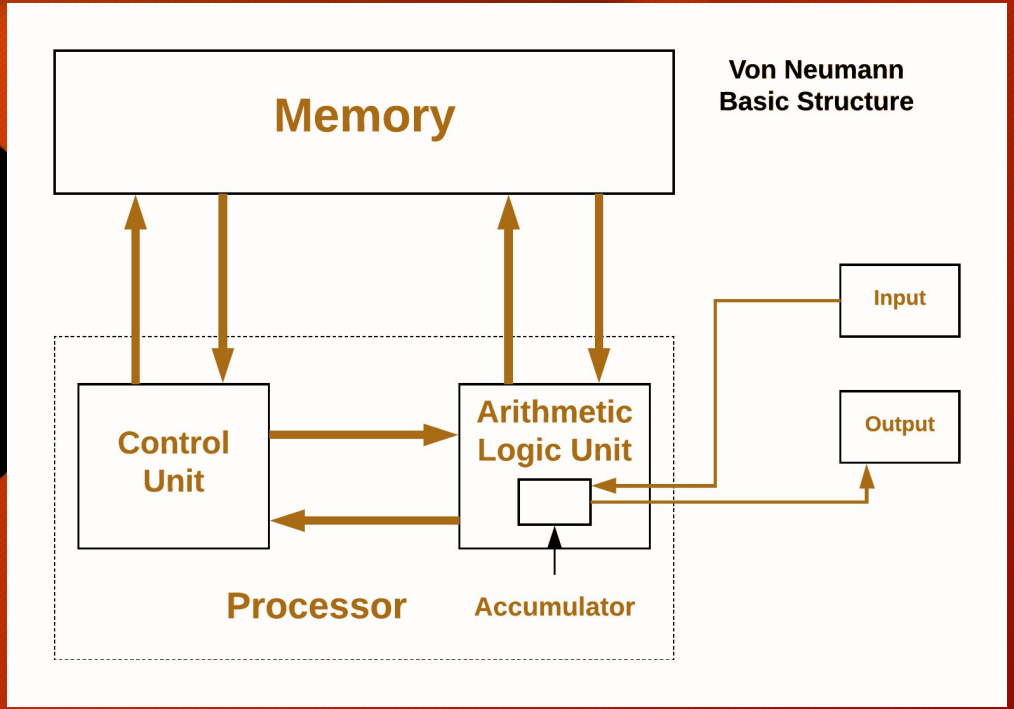
Andrew Waterman and Krste Asanovi

*Image from the book and manual unless specified

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.

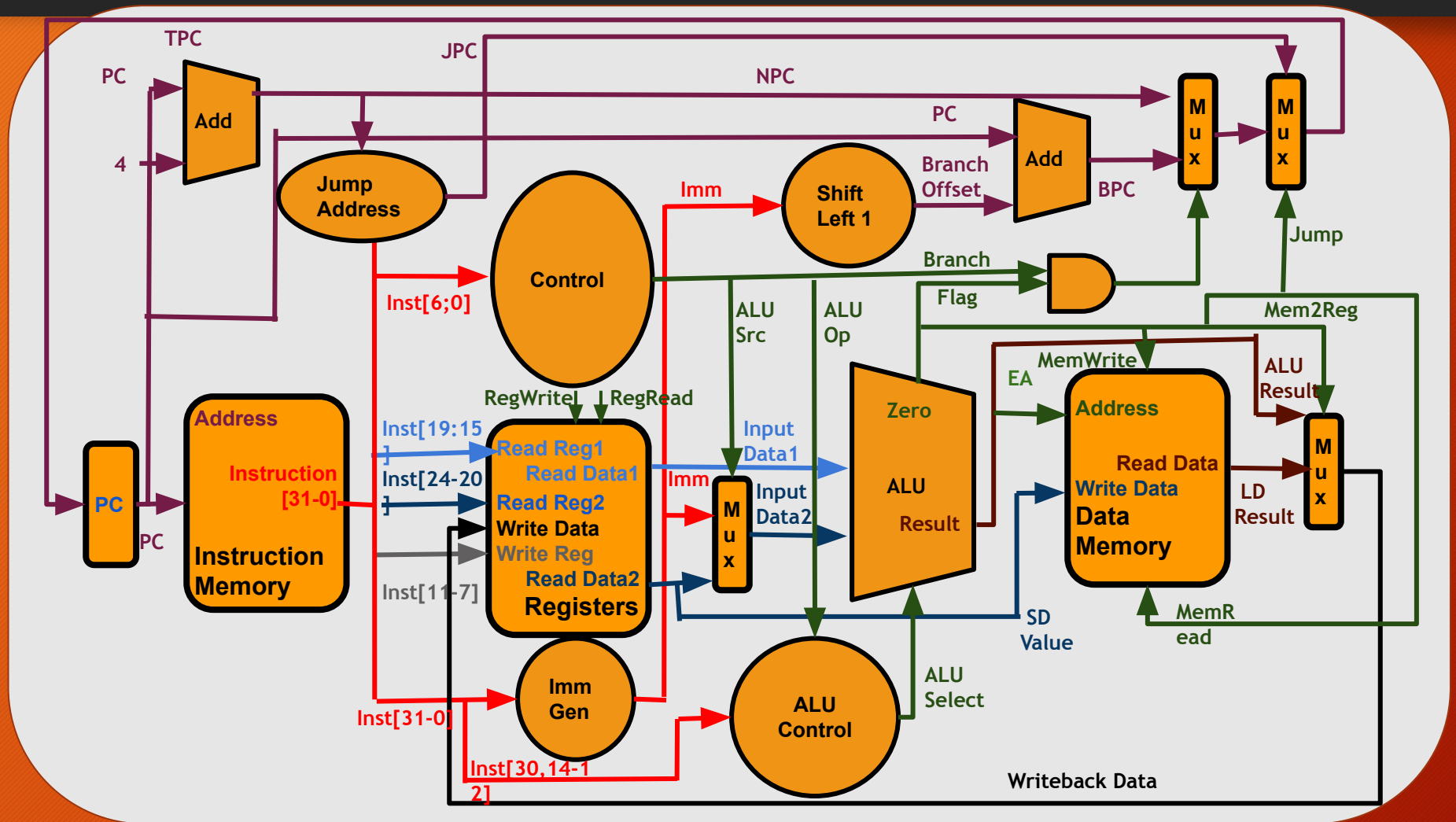


Geeks for Geeks

Agenda: A Basic Design of RISC-V

- 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.

Overview of Implementation (Control and Data Path)

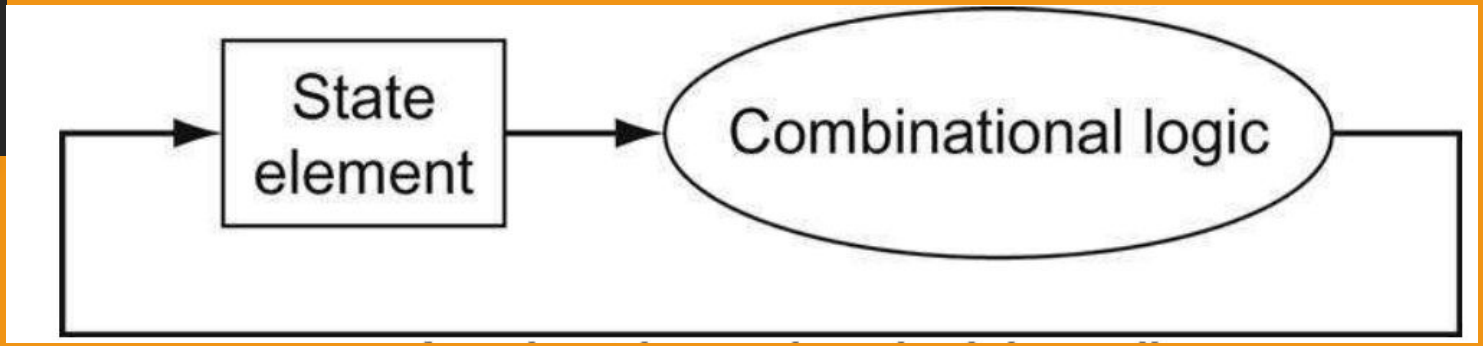


Control Path: The signals that drive the computation

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

Digital Logic Fundamentals

- Combinational Circuits
- Sequential Circuits
- State Elements or Memory
- Clocks

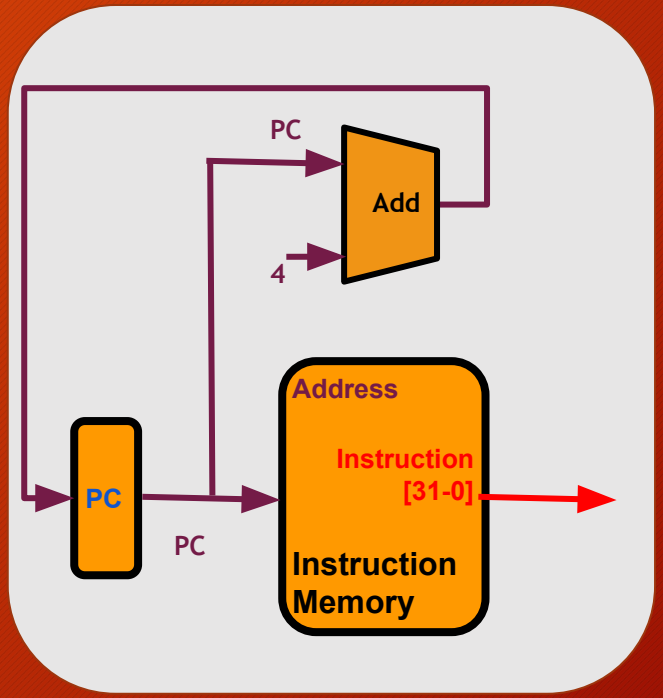


Instruction Fetch

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
  - instruction = IM[PC]
  - PC = PC + 4

Register File Design

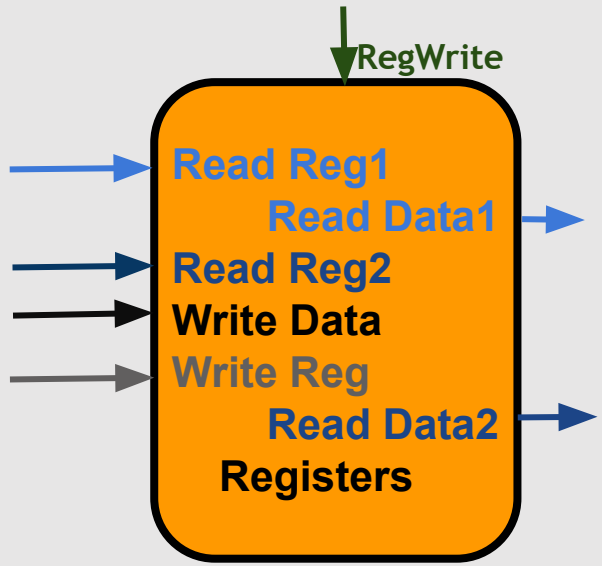
Read Registers: Register index specified in an instruction to access an input register

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 = instruction[19-15]
  - rsl2 = instruction[24-20]
  - rd = instruction[11-7]
  - rs1 = GPR[rsl1]
  - rs2 = GPR[rsl2]
  - GPR[rd] = result

Data Memory

Address: Effective memory address to be read

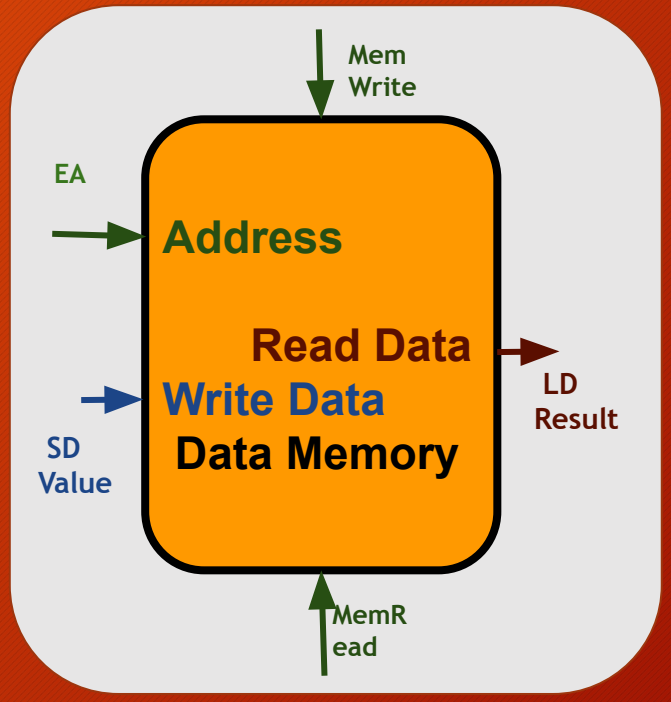
Read Data: Output data of the memory address

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

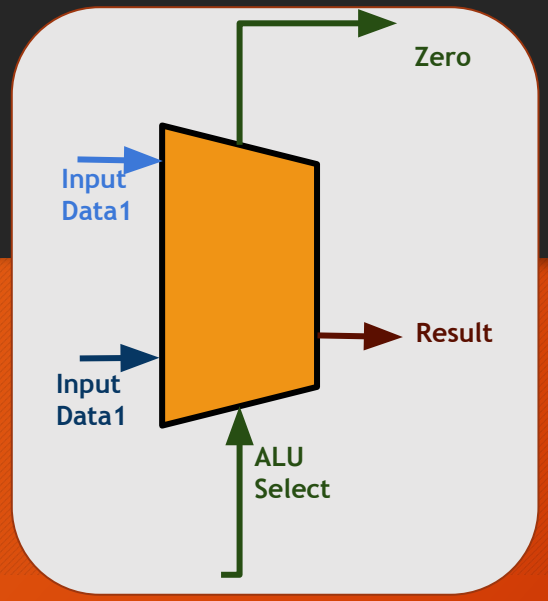
MemRead: Read Signal

MemWrite: Write Signal

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

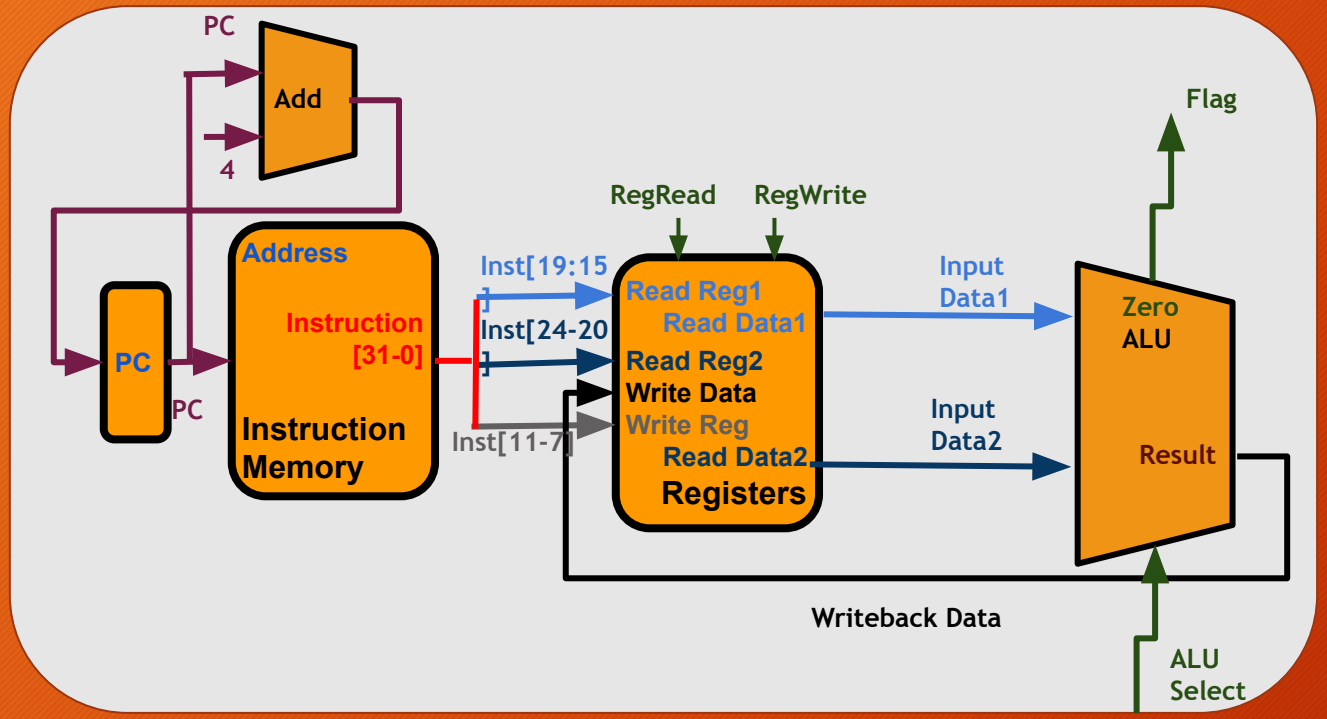


ALU Design



- Input rs1 and rs2
  - From the registers (All Instructions)
- Output rd
  - To register (For Non-LS Instructions)
  - To memory (For LS instructions)
- Status Signals
  - Flags (E.g., Zero)
- Control Signal
  - ALU Operation

Data-path Design for R-Type Instruction



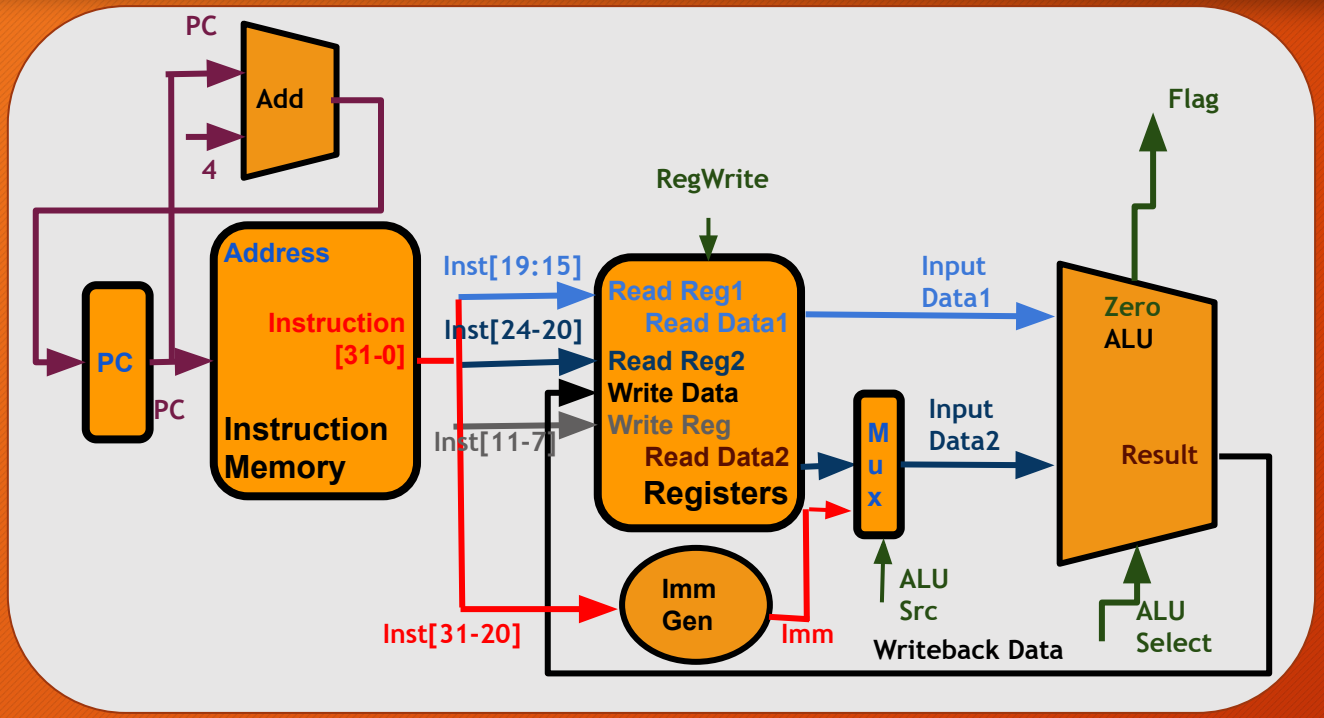
- instruction = IM[PC]
- PC = PC + 4
- rsl1 = instruction[19-15]
- rsl2 = instruction[24-20]
- rd = instruction[11-7]
- If RegRead Then rs1 = GPR[rsl1]
- If RegRead Then rs2 = GPR[rsl2]
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- func7 = instruction[31-25]
- ALUresult = ALU(ALUSelect, 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]

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd [11-7]	Opcode [6-0]

Data-path Design after Adding I-Type Instruction



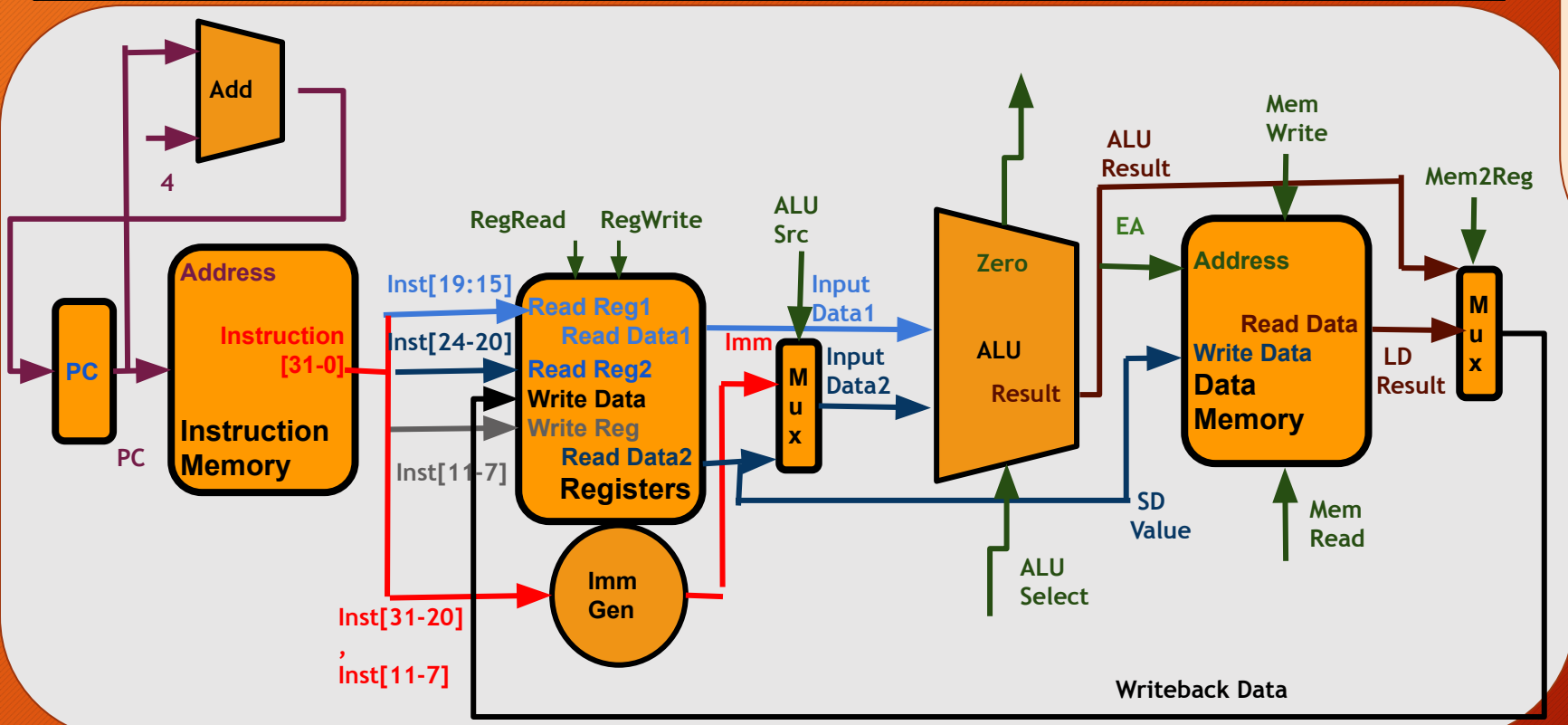
- instruction = IM[PC]
- PC = PC + 4
- imm = instruction[31-20]
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rsl1 = instruction[19-15]
- rsl2 = instruction[24-20]
- rd = instruction[11-7]
- rs1 = If RegRead Then GPR[rsl1]
- *If ALUSrc*
- *Then rs2 = imm*
- *Else* If RegRead Then *rs2 =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

- 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

Data-path Design after Adding Memory Instructions

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd[11-7]	Load [6-0]

immediate[31-25]	rs2[24-20]	rs1 [19-15]	func3 [14-12]	imm[11-7]	Store[6-0]

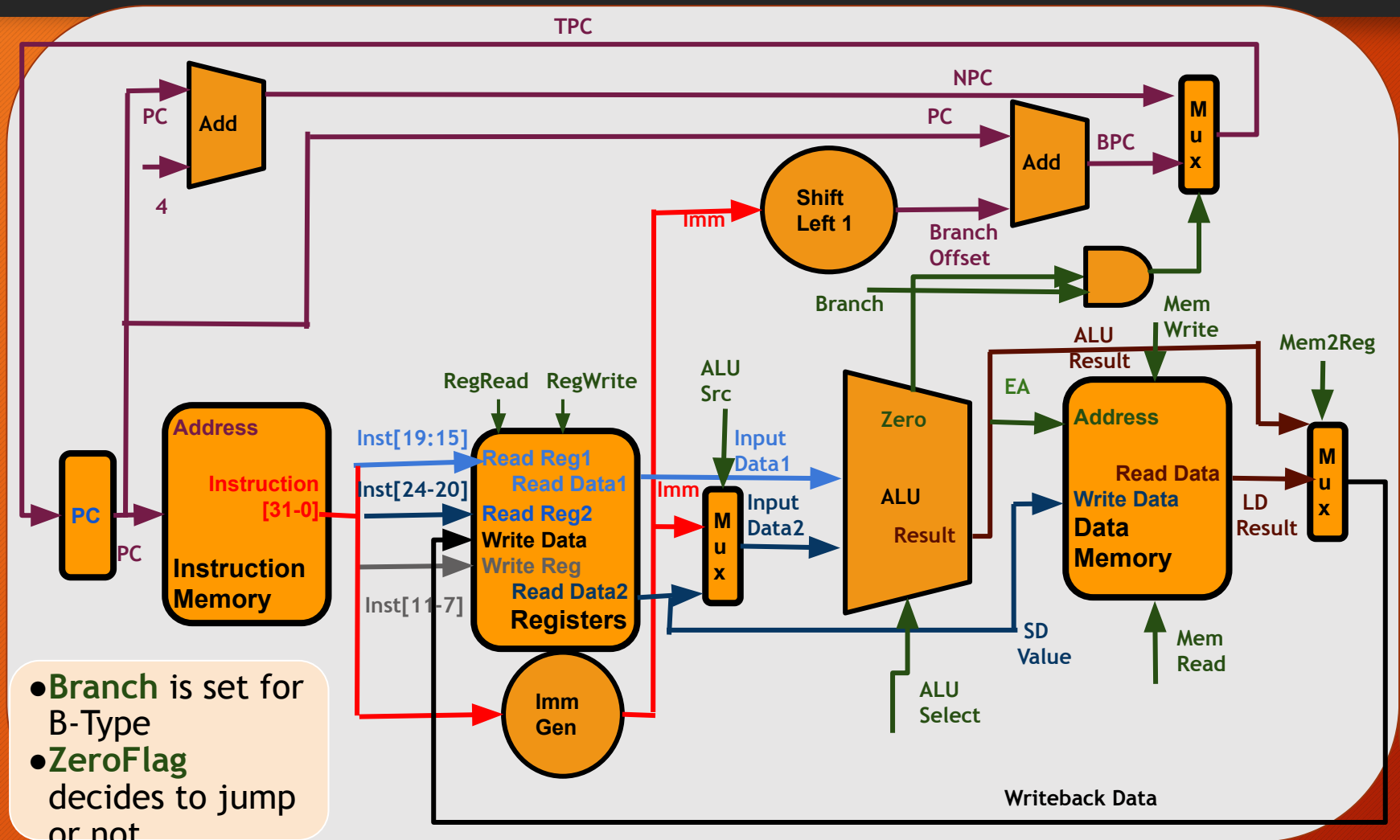


- instruction = IM[PC]
- PC = PC + 4
- imm=ImmGen(instruction[31-20],instruction[11-7])
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rsl1 = instruction[19-15]
- rsl2 = instruction[24-20]
- rd = instruction[11-7]
- If RegRead Then rs1 = GPR[rsl1]
- If ALUSrc
- Then rs2 = imm
- Else If RegRead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

immediate[31-25]	rs2[24-20]	rs1 [19-15]	func3 [14-12]	immediate[11-7]	Opcode [6-0]

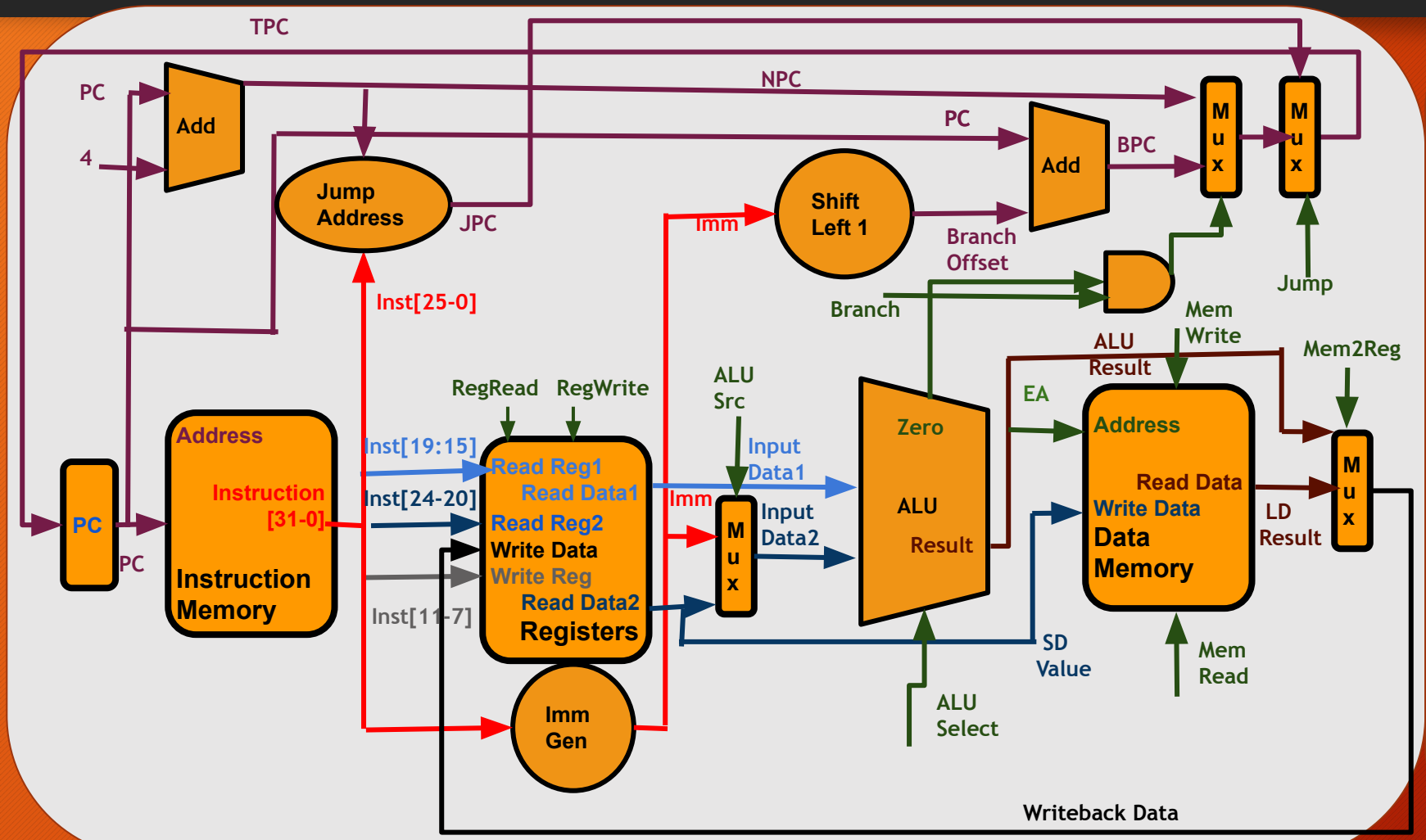
Data-path Design after Adding B-Type Instructions



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

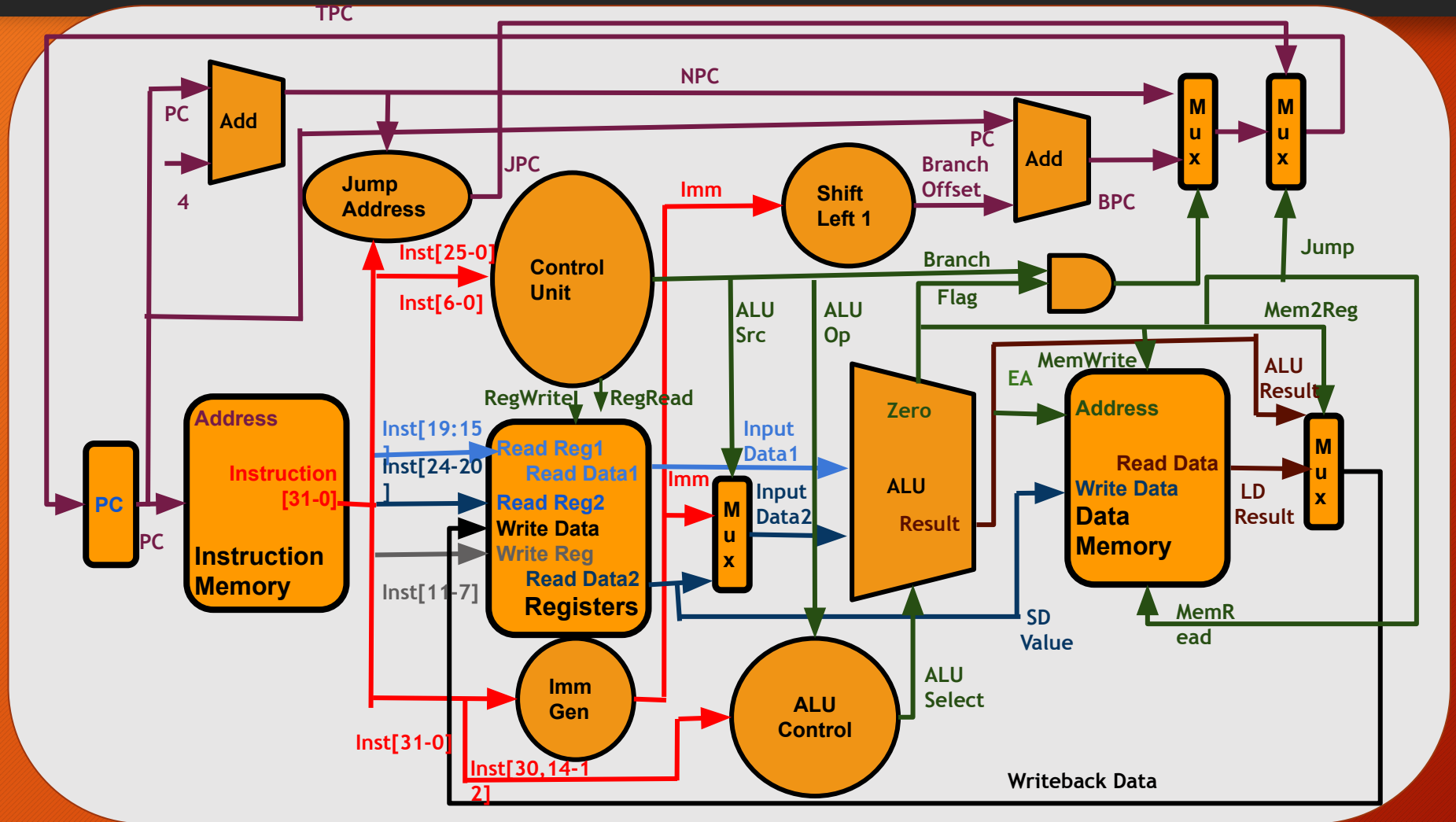
- instruction = IM[PC]
- *NPC = PC + 4*
- imm=ImmGen(instruction[31-20],instruction[11-7]
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rs1 = instruction[19-15]
- rs2 = instruction[24-20]
- rd = instruction[11-7]
- rs1 = GPR[rs1]
- If ALUSrc
- Then If RegRead Then rs2 = imm
- *Else If RegRead Then rs2 = GPR[rs2]*
- 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[31-28],instruction[25-0]<<2*
- imm=ImmGen(instruction[31-20],instruction[11-7])
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rs1 = instruction[19-15]
- rs2 = instruction[24-20]
- rd = instruction[11-7]
- rs1 = GPR[rs1]
- If ALUSrc
- Then If RegRead Then rs2 = imm
- Else If RegRead Then rs2 = GPR[rs2]
- 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[31-28],instruction[25-0] <<2
- 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 TA = BPC
  - Else TA = NPC
- If Jump Then TPC = JPC
- If RegWrite Then
  - If Mem2Reg
    - Then GPR[rd] = LDresult
    - Else GPR[rd] = ALUresult
- PC = TPC

ControlWord

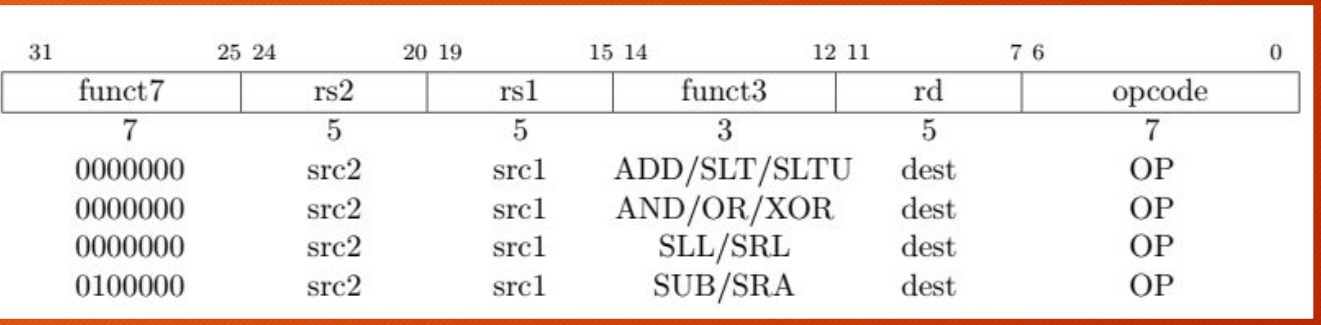
- RegRead enabled to read GPR
- ALUSrc: Selects between rs2 and Imm
- ALUOp: Operation to be executed on the ALU
- MemWrite is used for Store
- MemRead is used for loads
- RegWrite enabled to write GPR
- Mem2Reg selects between ALUResults and LDResults
- Branch and Jump selects between Branch PC, Jump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



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

Func7

Func3

Solve K-map?

ALU Control Signal

Generation Control Signal: Opcode Handling

Instruction	ALU Src	Mem-Reg	Req-Write	Mem-Read	Mem-Write	Branch	ALU Op1	ALU Op2
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

Debiprasanna Sahoo
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

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/Length of Clock Cycle

Clock Cycle Time Analysis

Critical Path of a Circuit: Longest path between two registers if the combinational circuit between the resisters 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 Time for Each Instruction

Instruction Class	Fetch	Decode	Execute	Memory	Writeback	Total
ld	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

Pipelining: Concept

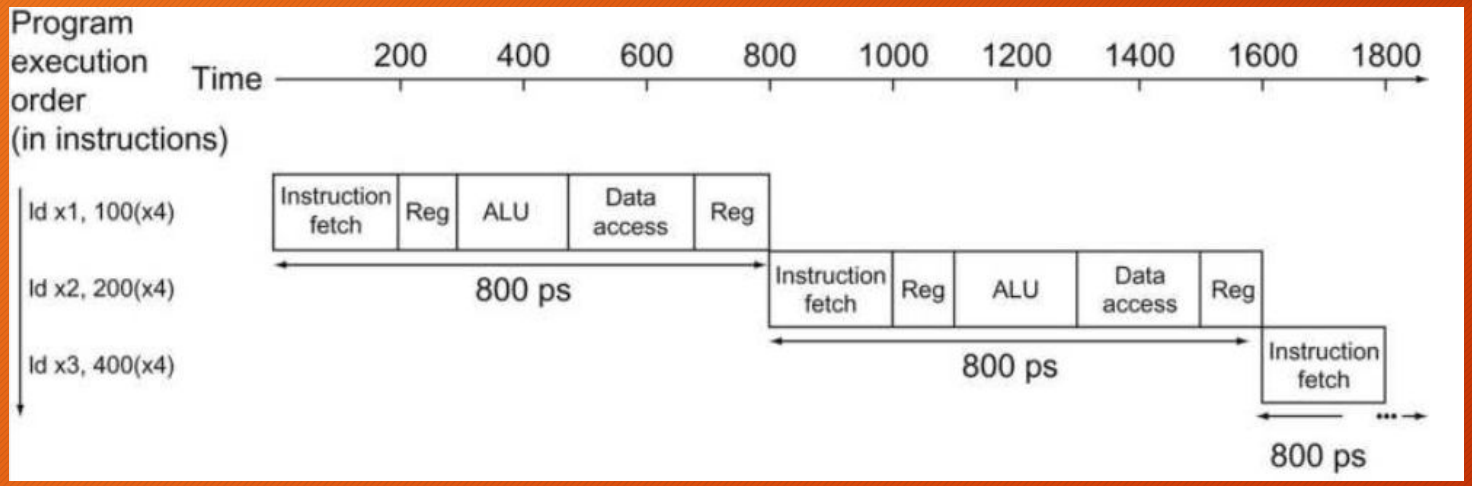
Assembly line: Overlapping execution of multiple instruction in parallel

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.

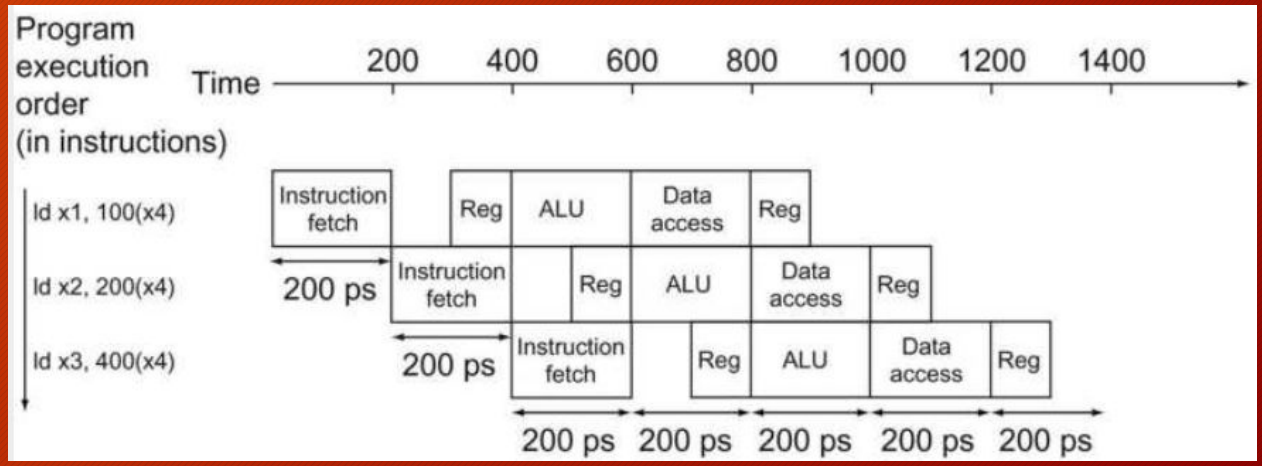
Critical Path Pipelined System: **The gate delays of the critical path between two intermediate register accesses.**

Instruction Time: Non-pipelined vs Pipelined

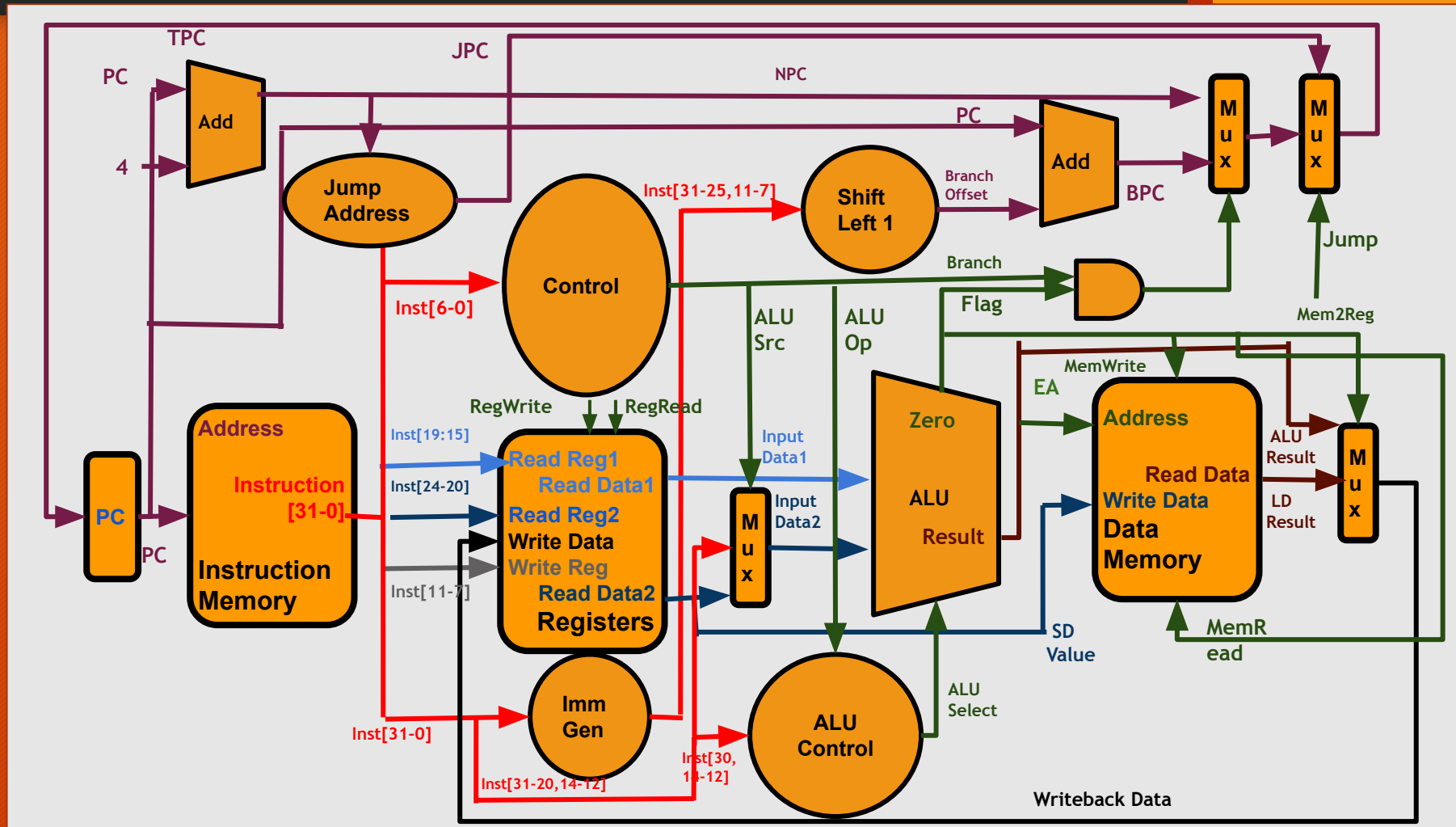


Non-Pipelined System

Pipelined System



Non-Pipelined Core: Design



Non-Pipelined Core: Implementation

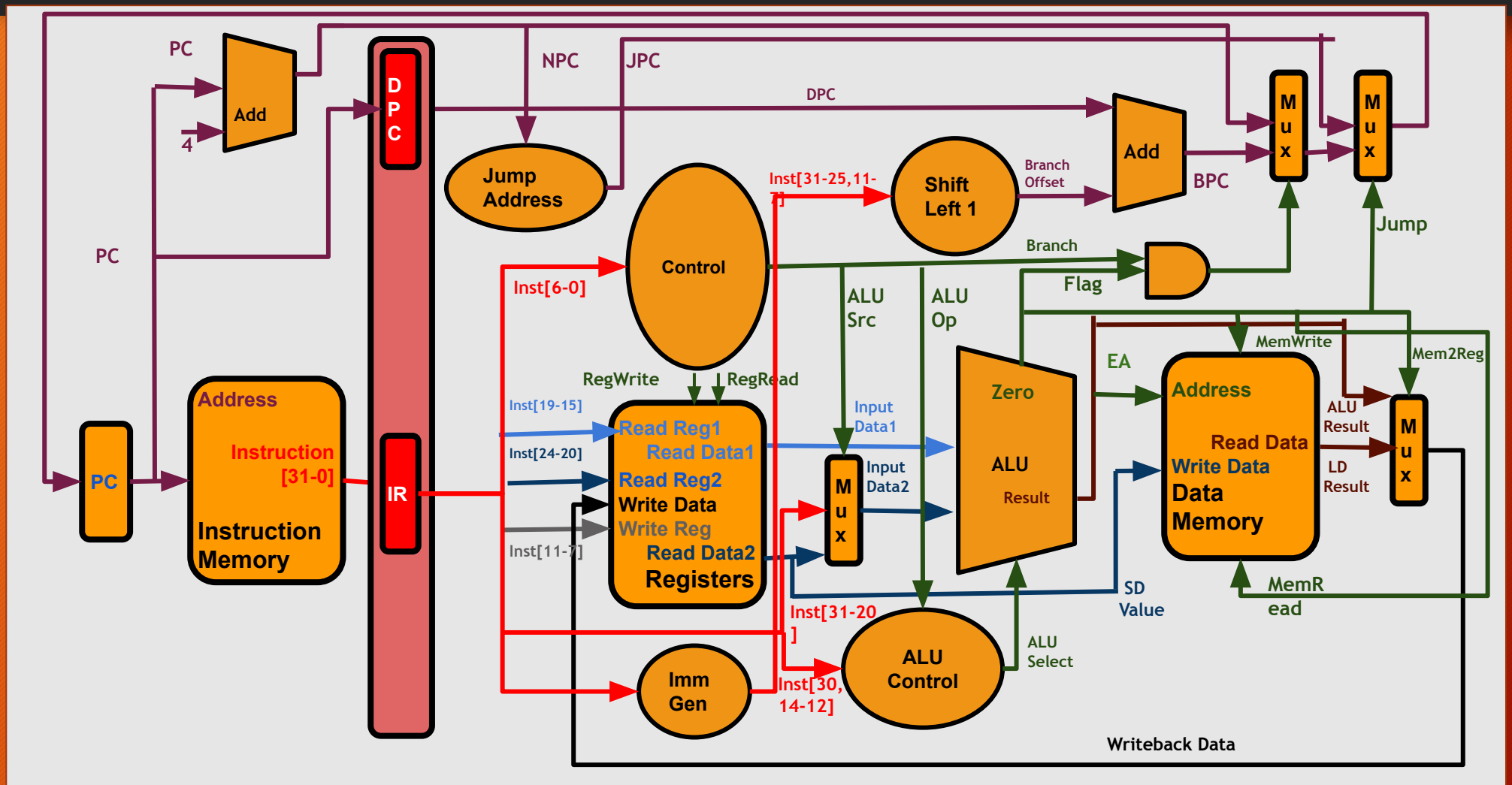
- instruction = IM[PC]
- NPC = PC + 4
- JPC=NPC[31-28],instruction[25-0]<<2
- imm = instruction[31-20]
- opcode = instruction[6-0]
- func3 = instruction[14-12]
- rsl1 = instruction[19-15]
- rsl2 = instruction[24-20]
- rd = instruction[11-7]
- *ControlWord = controller(opcode, func)*
- If RegRead Then 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 TA = BPC
  - Else TA = 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

2-Stage Instruction Pipeline: Design



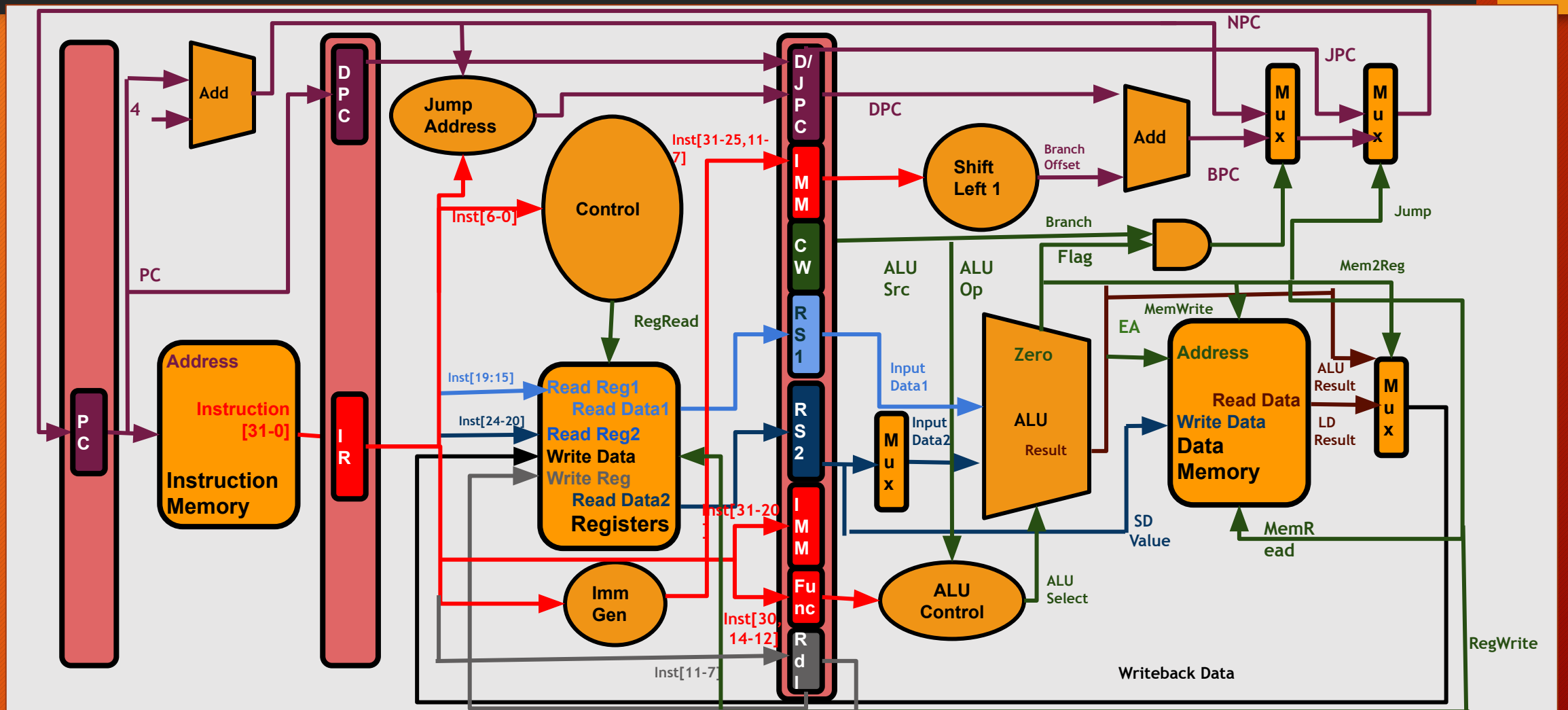
2-Stage Instruction Pipeline: Implementation

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

- JPC=NPC[31-28],instruction[25-0]<<2
- 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, func)
- 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) + PC
- If CW.Branch and ALUZeroFlag
  - Then PC = BPC
  - Else PC = NPC
- 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



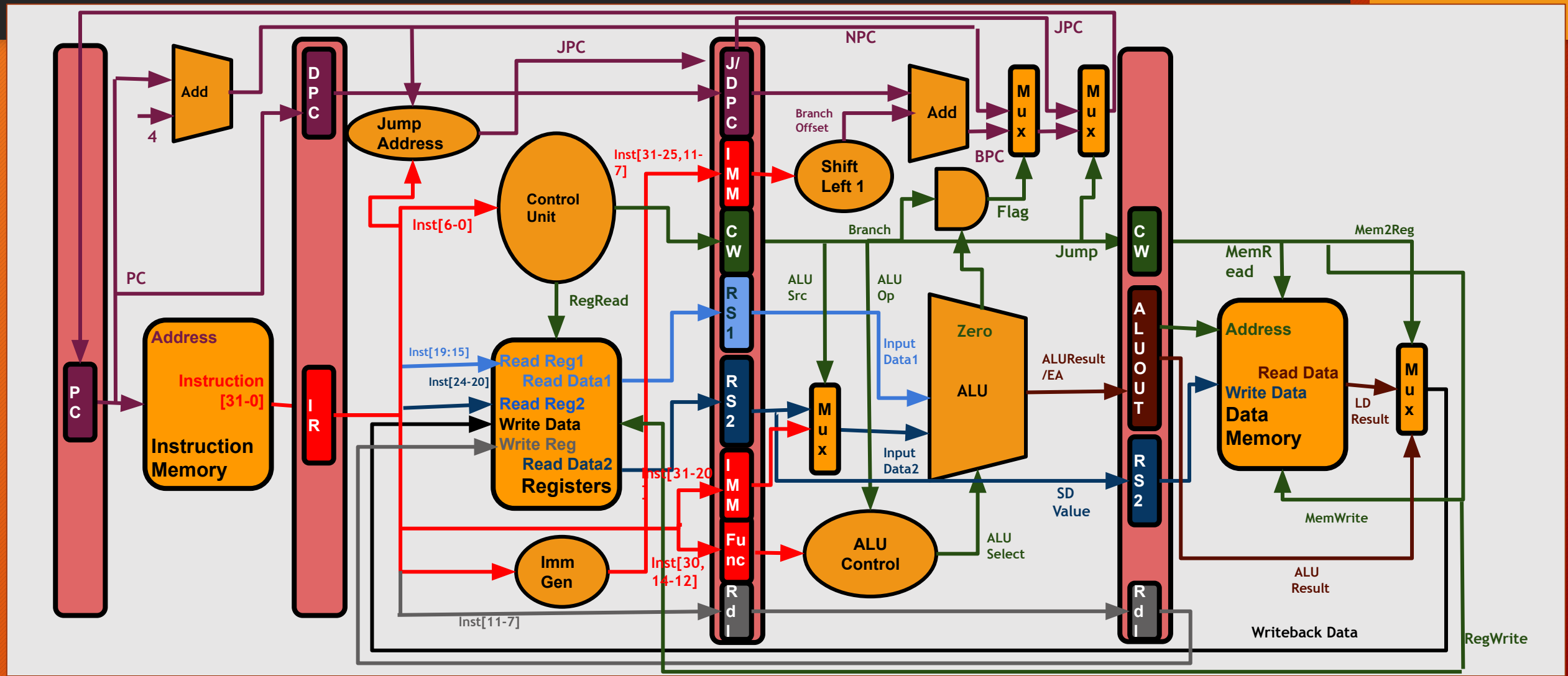
3-Stage Instruction Pipeline: Implementation

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

- IDEX.JPC=IFID.NPC[31-28],IFID.IR[25-0]<<2
- 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
- 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]]

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

4 Stage Instruction Pipeline: Design



4-Stage Instruction Pipeline: Implementation

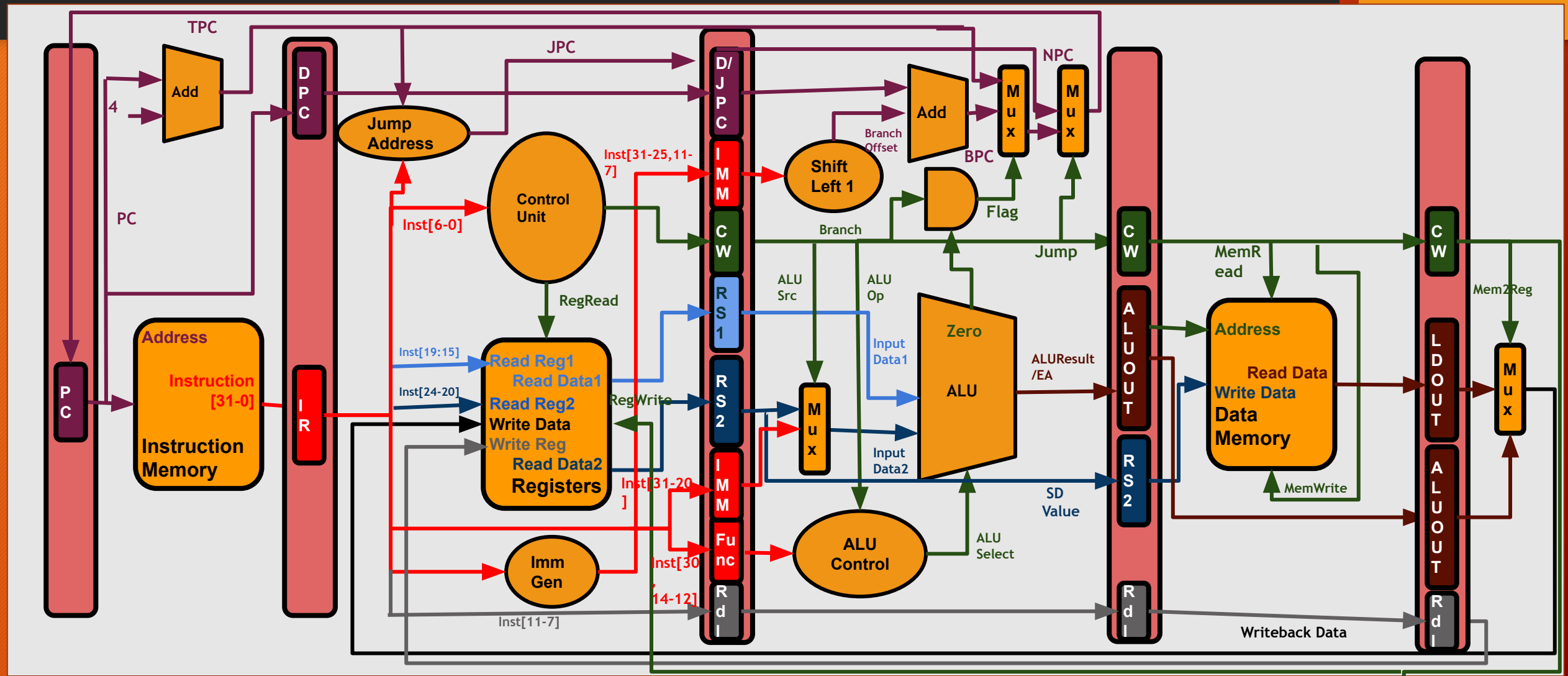
- IFID.IR = IM[PC]
- IFID.DPC = PC
- NPC = PC + 4

- IDEX.JPC=IFID.NPC[31-28],IFID.IR[25-0]<<2
- 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
- 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]]

- 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 and ALUZeroFlag
  - Then PC = (IDEX.imm << 1) + IDEX.DPC
  - Else PC = NPC
- If IDEX.CW.Jump Then PC = IDEX.JPC

- If IDEX.CW.MemWrite Then
  - DM[EXMO.ALUOUT] = IDEX.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

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

- IDEX.JPC=IFID.NPC[31-28],IFID.IR[25-0]<<2
- 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
- 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]]

- 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 and ALUZeroFlag
  - Then PC = (IDEX.imm << 1) + IDEX.NPC
  - Else PC = NPC
- If IDEX.CW.Jump Then PC = IDEX.JPC

- 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 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

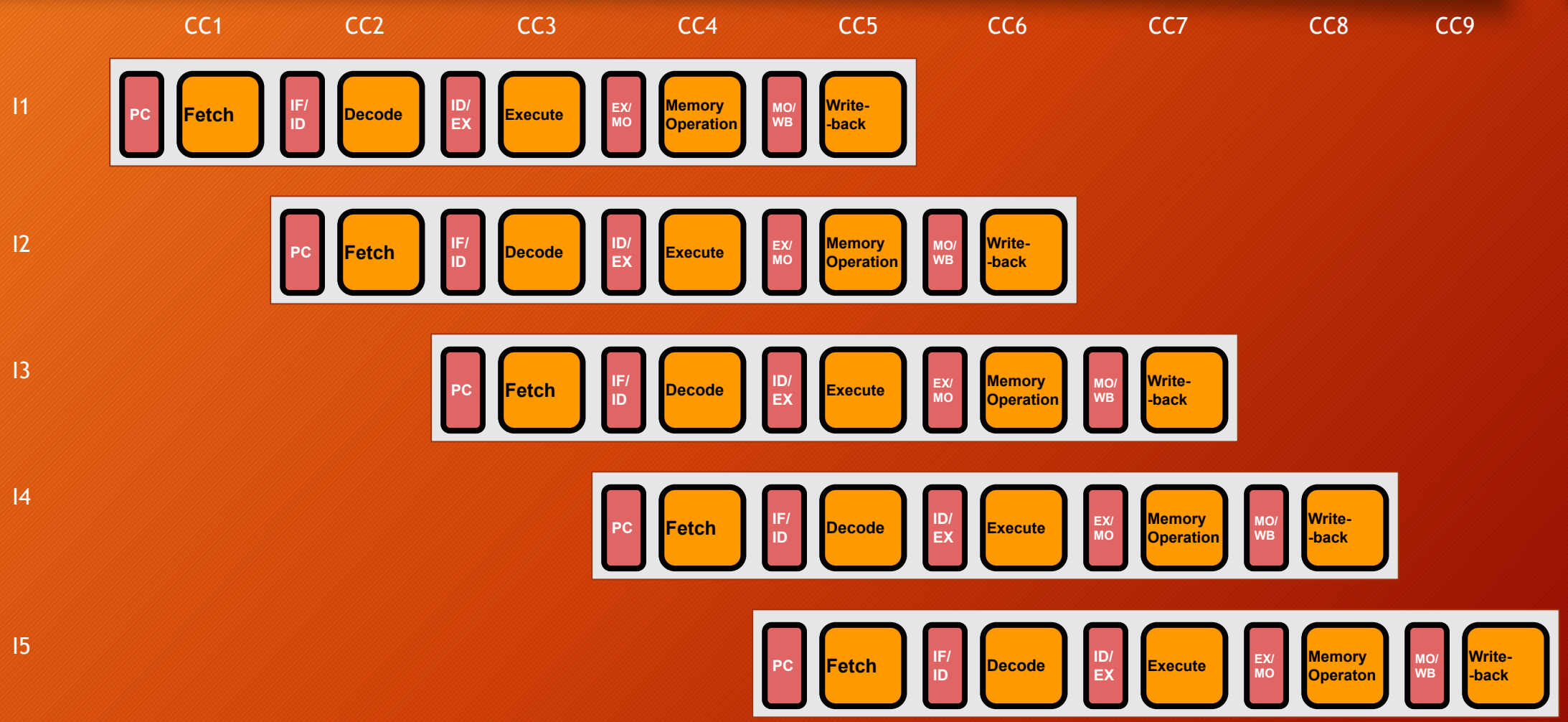
Memory Access

- 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 N-Stage Pipeline

Latency: **The gate delays of the critical path between two intermediate register accesses.**

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).

Non-Pipelined System

- **Latency = t_{np}**
- **Throughput: CPI = 1**

N-Stage Pipelined System

- **Latency = $t_p = (t_{np}/N) + t_r$**
- **t_r = Delay of intermediate registers**
- **Throughput: CPI close to 1 for large number of instructions**

Speedup = Execution Time$_{old}$/Execution Time$_{new}$ = $t_{np}/ t_p = t_{np}/ ((t_{np}/N) + t_r)$ comes close to N

Problems with Pipelined Systems

Frequency $f = 1/t_p$

Dynamic Power $P = nCV^2fA$

Dynamic Voltage and Frequency Scaling (DVFS)

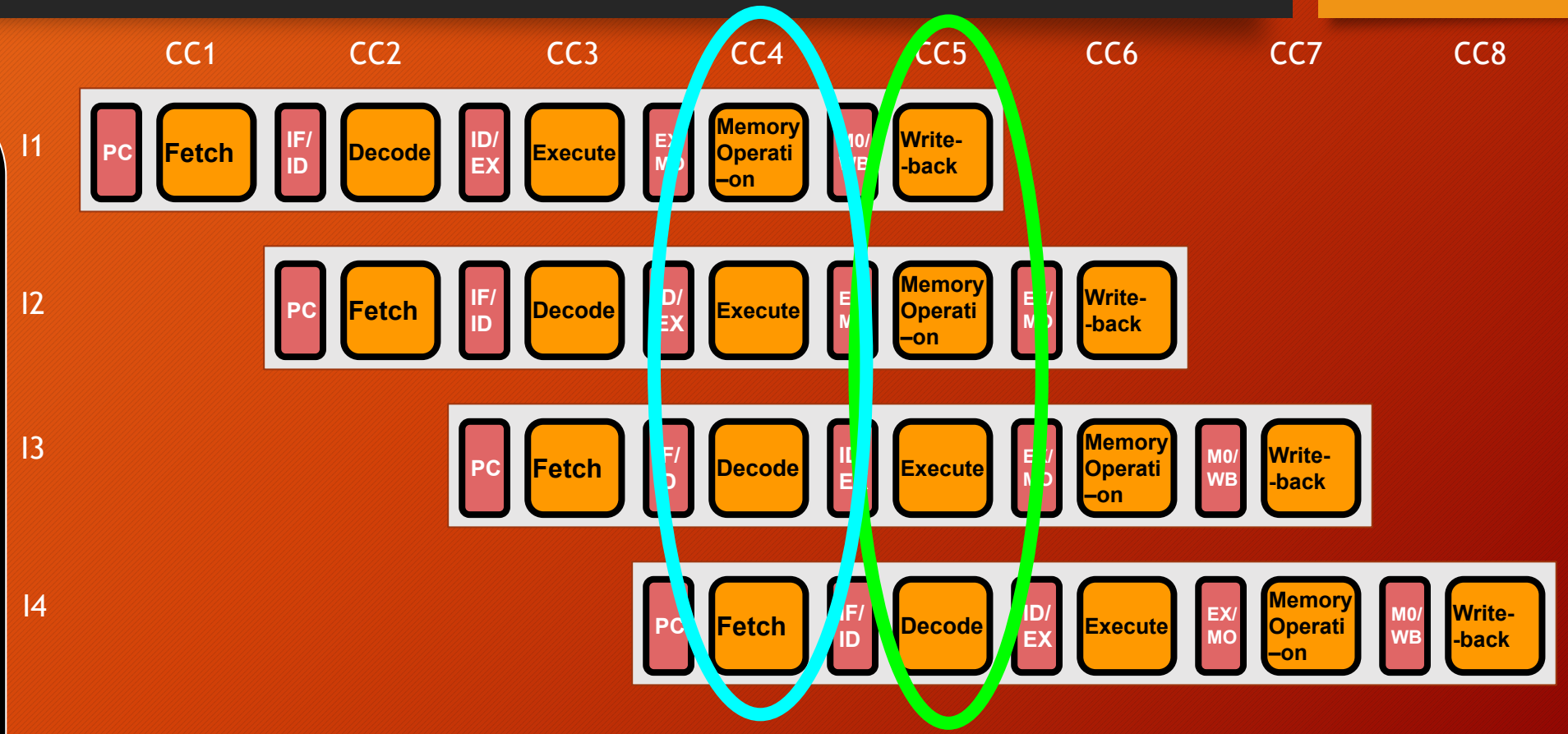
Heat Sinks

Speedup Can it be 1?

1. **Structural Hazards**
2. **Data Hazards**
3. **Control Hazards**

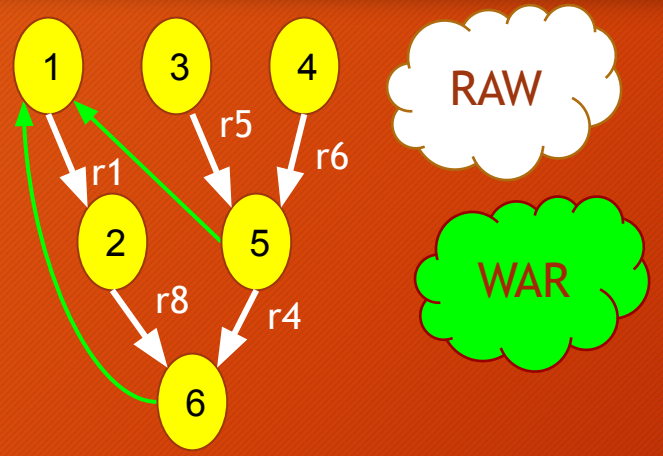
Structural Hazards

- 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
```


1	2	3
Read after Write (RAW) Hazards	Write after Read (WAR) Hazards	Write after Write (WAW) Hazards

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

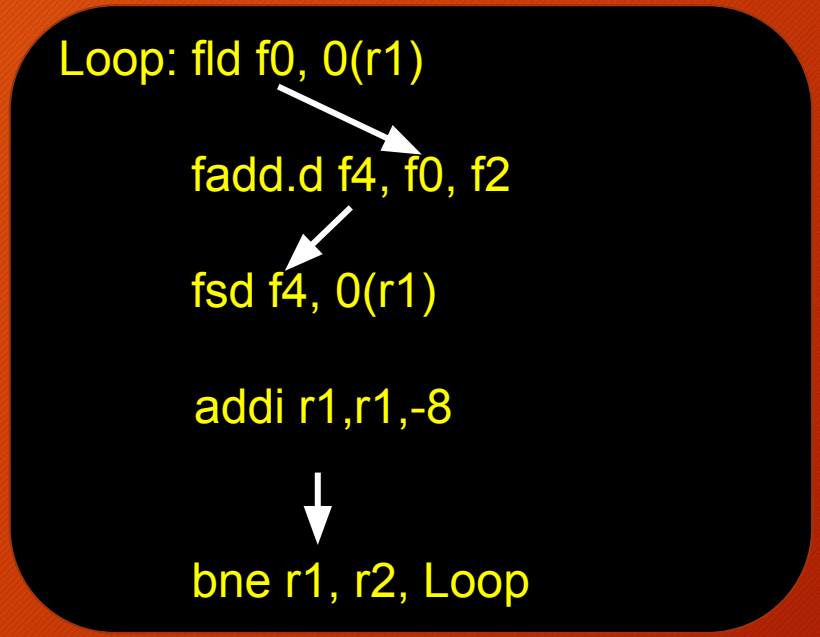
- ❑ Data dependency can convey three things:
  - Possibility of hazard
  - Orders in which results are calculated
  - Upper-bound on how much parallelism can possibly be exploited

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

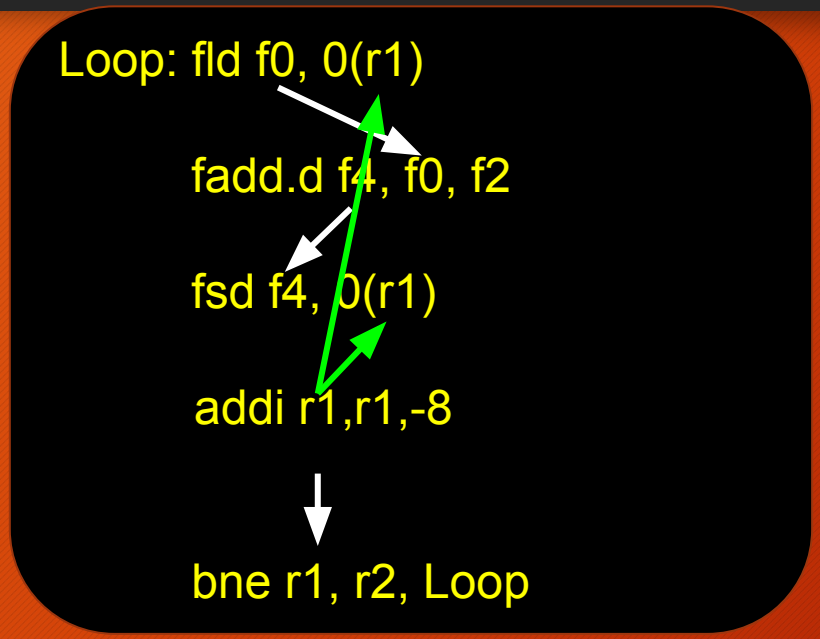
- ❑ Data dependent values can flow between instructions via:
  - Registers: Easy to find dependency
  - Memory location: Difficult to detect dependency. Why? -> Effective address calculation is data-dependent

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

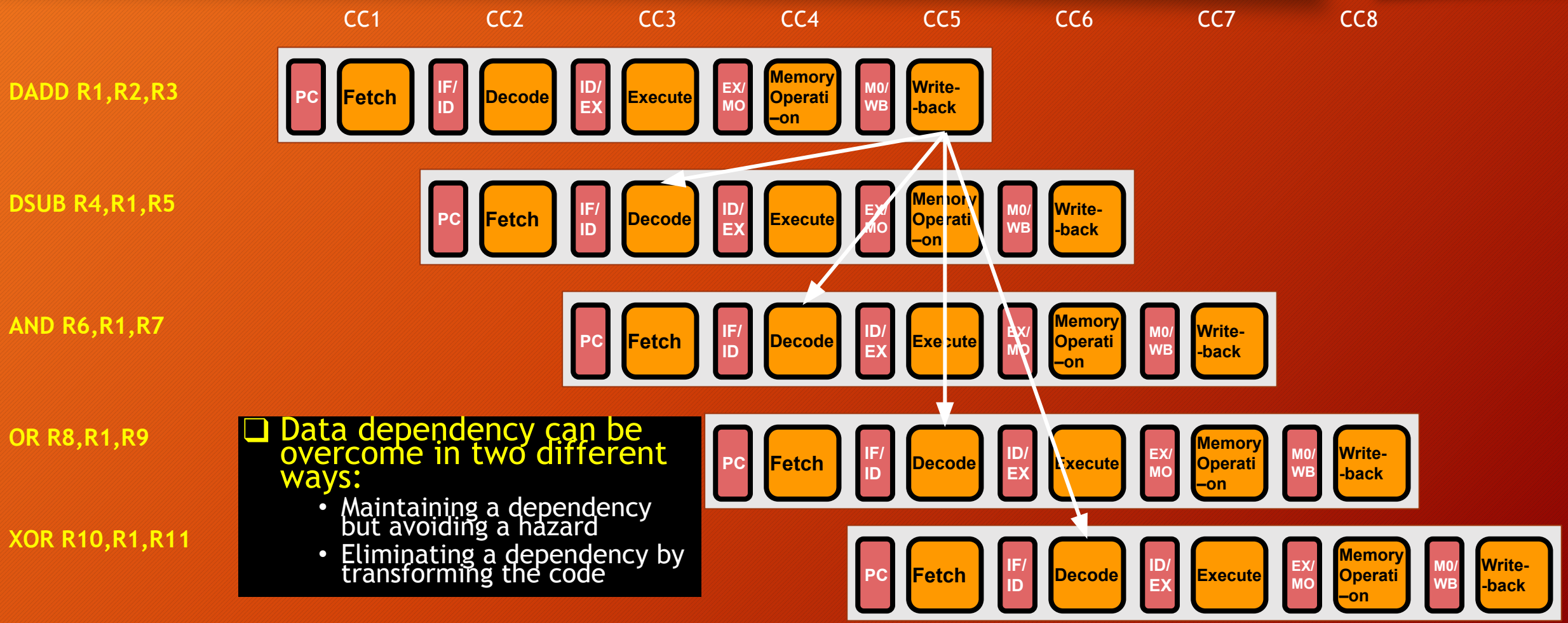


- Anti-dependency: *i* precedes *j* but instruction *j* writes a register or memory that instruction *i* reads
- No problem with in-order pipeline execution.
- What is program order? What is in-order?
- What will happen if we start ***executing*** instructions in parallel or out-of-order

Name Dependency: Output Dependency

- Output-dependency: *i* precedes *j* but instruction *j* writes a register or memory that instruction *i* writes 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



- Data dependency can be overcome in two different ways:
  - Maintaining a dependency but avoiding a hazard
  - Eliminating a dependency by transforming the code

Control Hazard

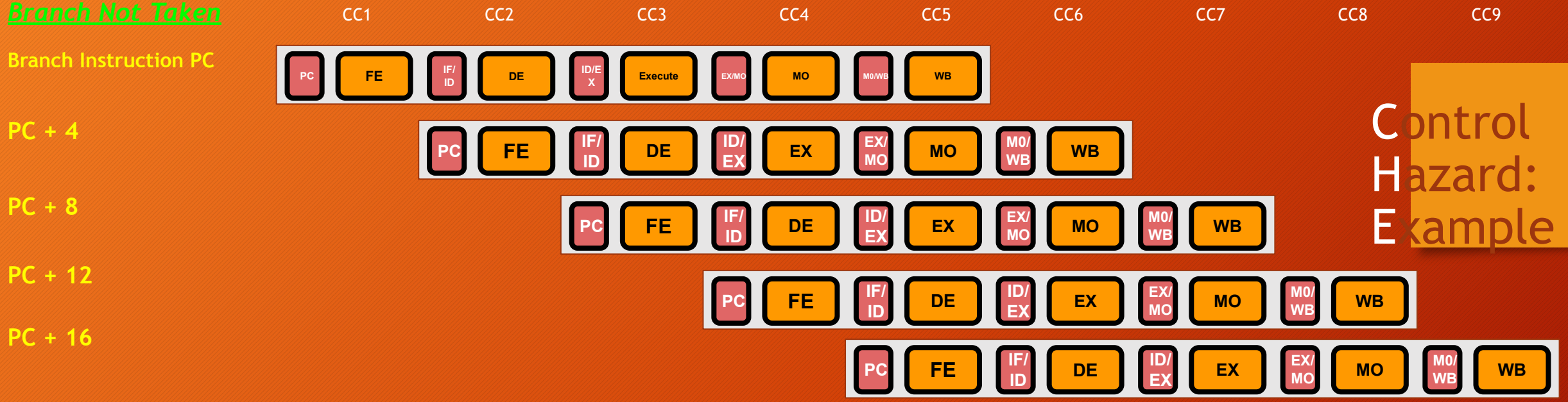
```
Loop: fld f0, 0(r1)
      fadd.d f4, f0, f2
      fsd f4, 0(r1)
      addi r1,r1,-8
      bne r1, r2, Loop
      add f4, f4, f2
```

```
      bne r1, r2, elsepart
      fld f0, 0(r1)
elsepart:  fld f0, 0(r2)
```

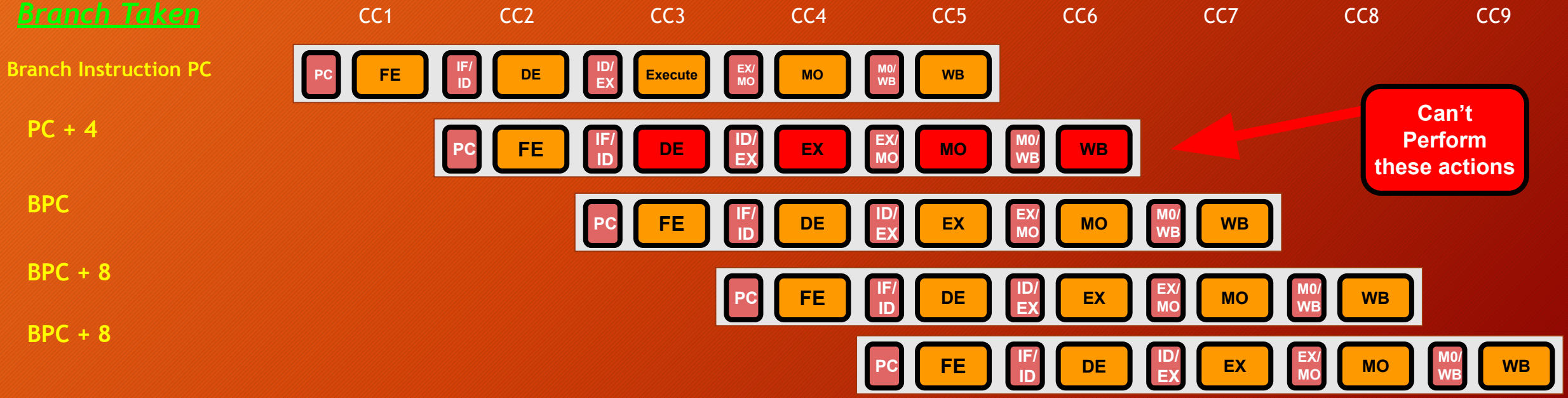
- Until we execute, the outcome of the branch is not known.
- The PC goes-on incrementing and pulls unwanted instructions into the pipeline.

Control Hazard: Example

Branch Not Taken

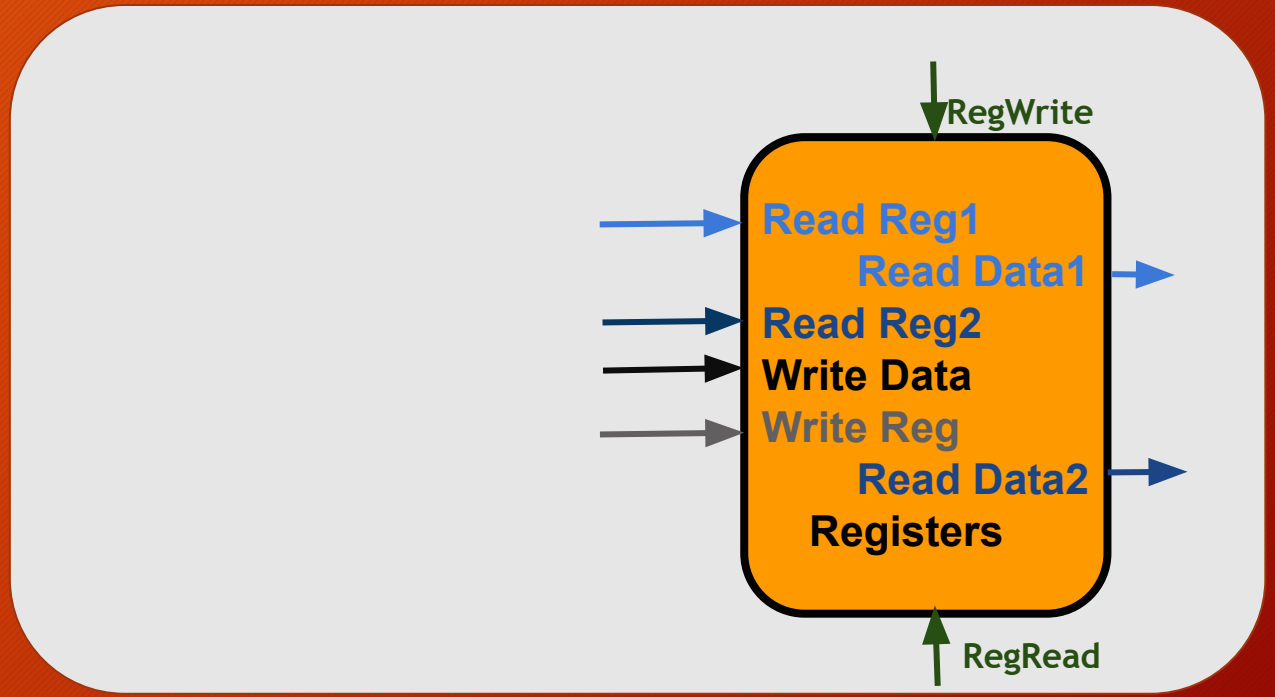


Branch Taken



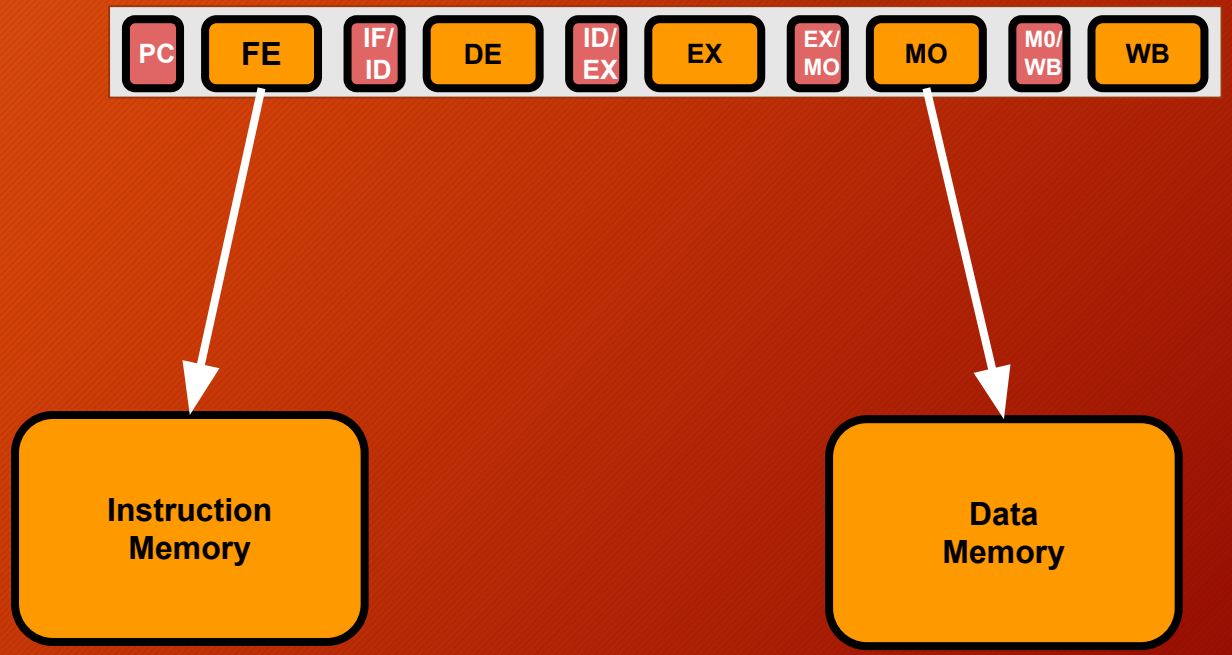
Structural Hazard Solution for Register Read/Write in Same Cycle

- 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.

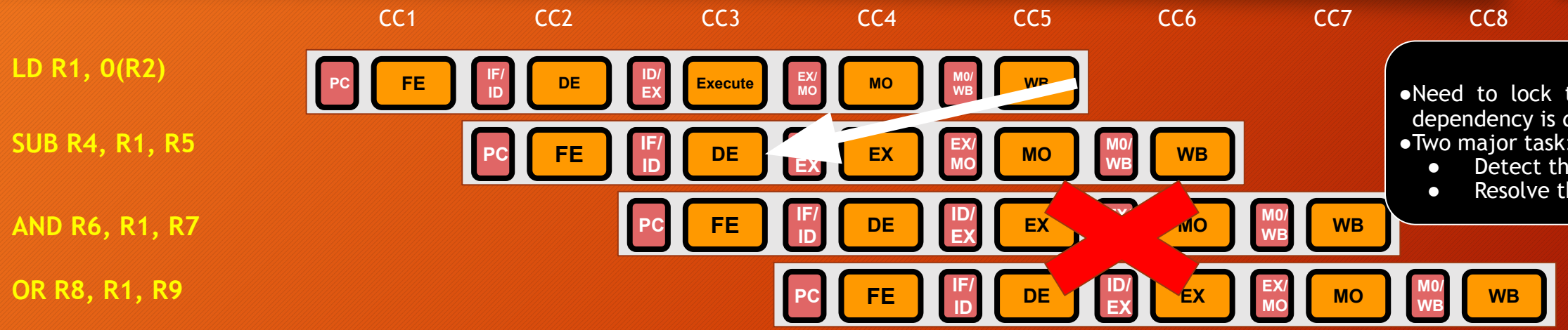


Structural Hazard Solution for Accessing Instruction and Data Memory

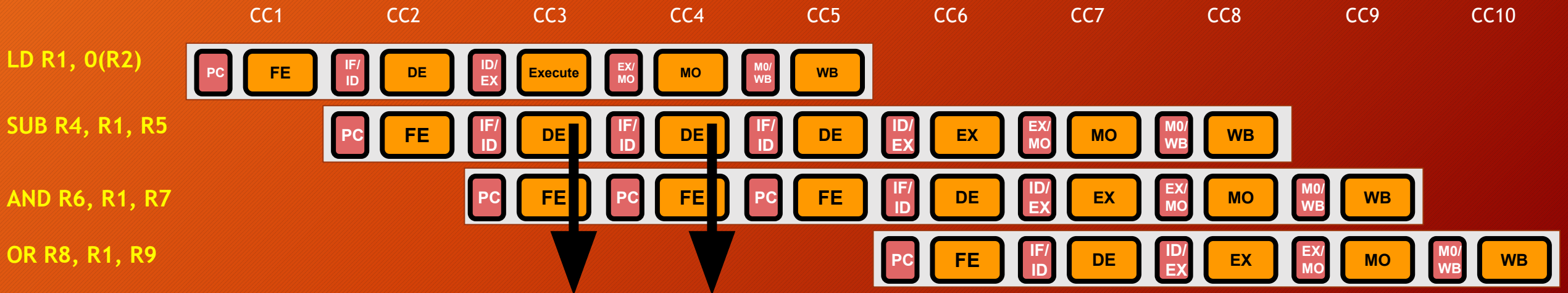
- Harvard Architecture: Use separate instruction and data memory.
- Fetch unit interacts with the instruction memory
- Memory stage interacts with the data memory
- Von-Neumann discussed about a single memory module.
- Do you remember the memory view of an application? Different segments of an applications: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



Data and Control Hazard Solution: Pipeline Interlock Concept



- Need to lock the pipeline when dependency is detected.
- Two major task:
  - Detect the dependency
  - Resolve the dependency



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*

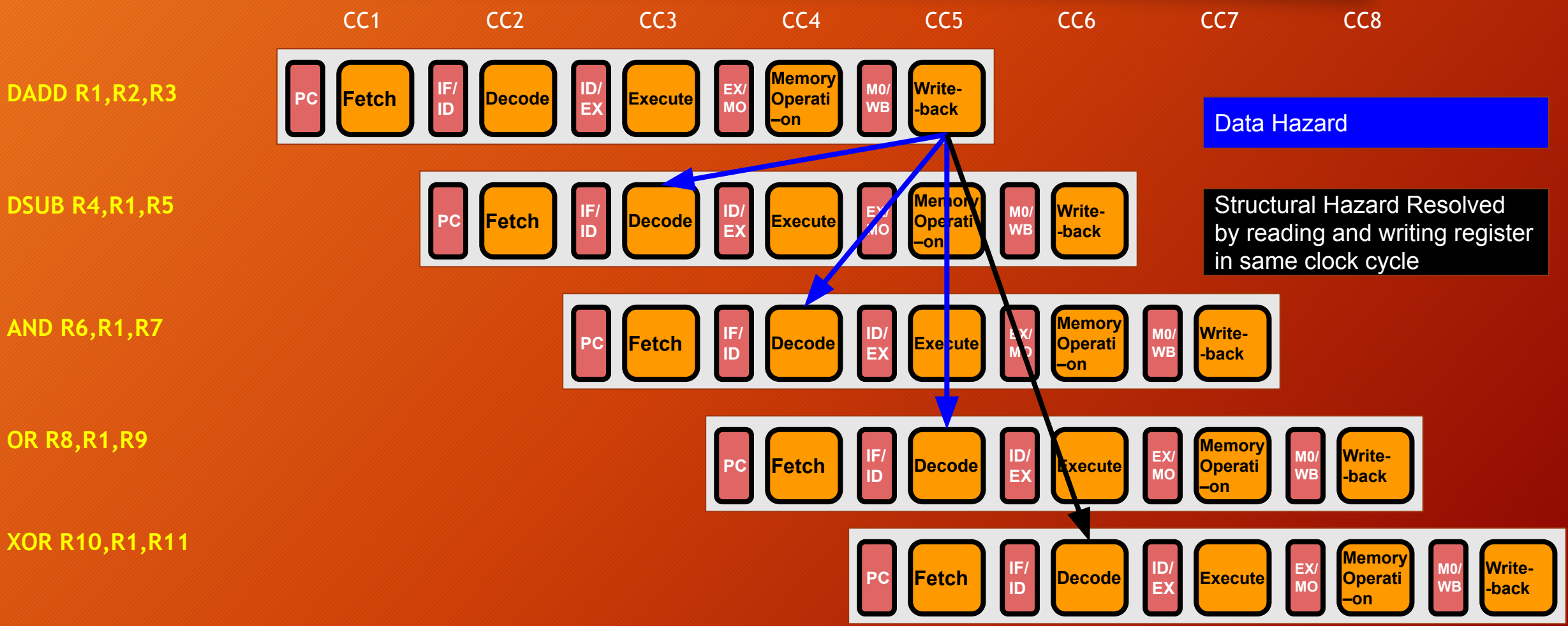
- *If IDEX.stall or !IFID.Valid Then return*
- IDEX.JPC=IFID.NPC[31-28],IFID.IR[25-0]<<2
- IDEX.DPC = IFID.DPC
- IDEX.imm = IFID.IR[31-20]
- IDEX.func = IFID.IR[14-12]
- IDEX.rdl = IFID.IR[11-7]
- IDEX.CW = controller(IFID.IR[6-0])
- If IDEX.CW.RegRead Then
- *If GPR[IFID.IR[19-15]].ins = -1*
- *Then 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[IDEX.rdl].ins = CW.ins*
- *IFID.stall = false*
- *IDEX.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 = false*
- If IDEX.CW.Jump
- Then PC = IDEX.JPC
- *IFID.Valid = false*
- *IDEX.Valid = false*
- *PC.Valid = false*
- *IDEX.stall = false*
- *EXMO.Valid = true*

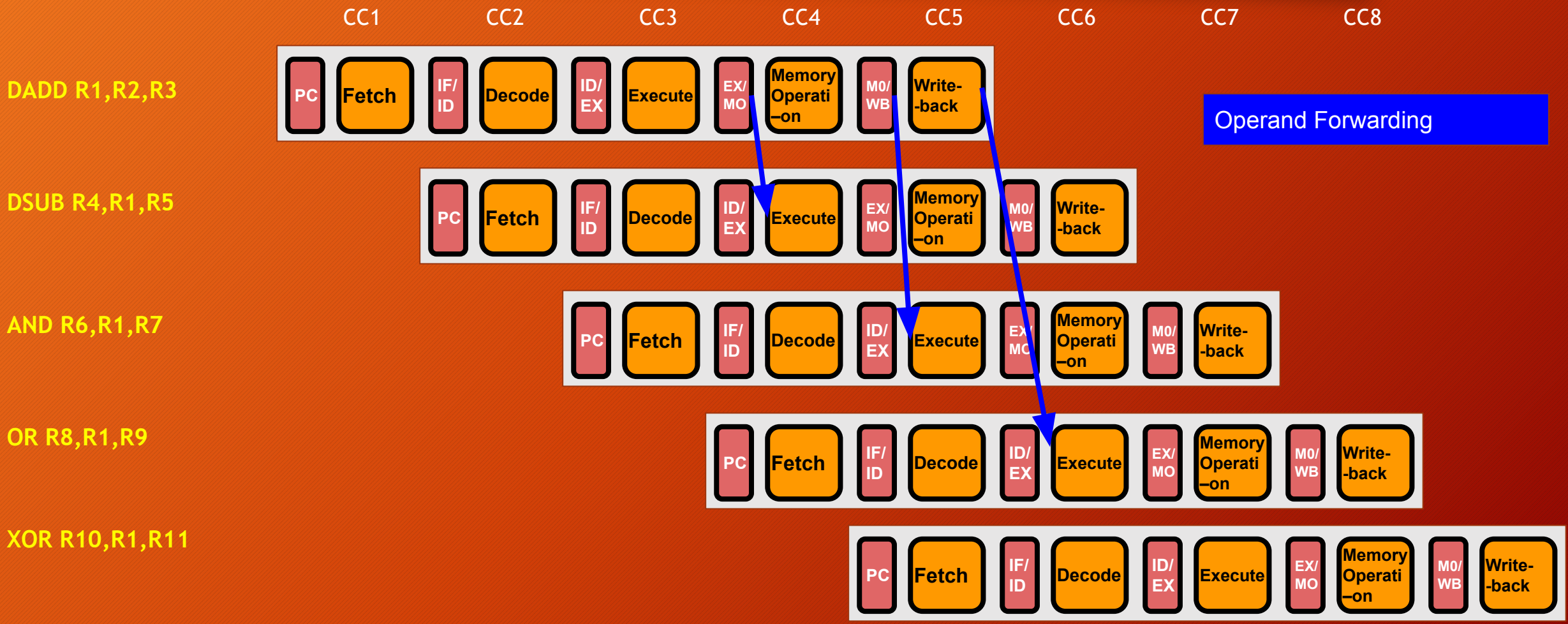
- *If !MOWB.Valid Then return*
- If !MOWB.CW.RegWrite Then Return
- Else *If MOWB.CW.Mem2Reg and GPR[MOWB.rdl].ins = CW.ins*
  - Then GPR[MOWB.rdl].value = MOWB.LDOUT
  - Else GPR[MOWB.rdl].value = MOWB.ALUOUT
  - *GPR[MOWB.rdl].ins = -1*
  - *MOWB.stall = false*

- *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*

Operand Forwarding: Concept and Example Problem



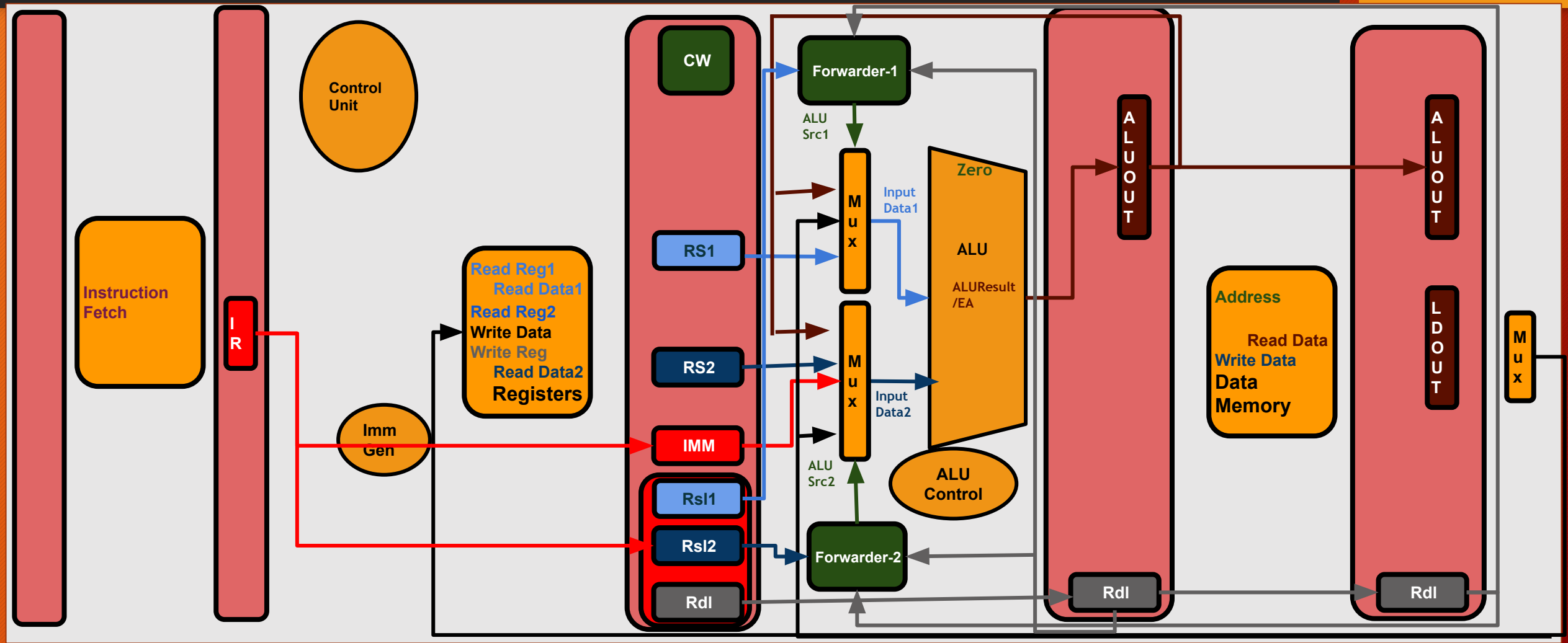
Operand Forwarding: Concept and Example Solution



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

Operand Forwarding: Design



Operand Forwarding: Implementation

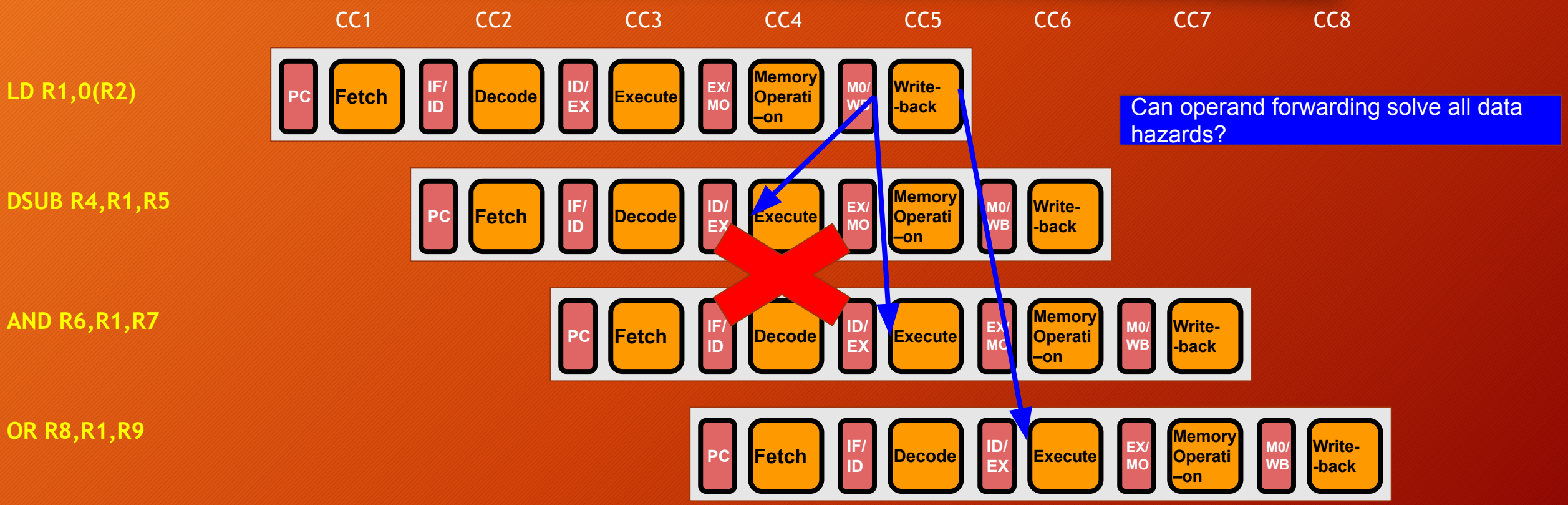
- If IDEX.stall or !IFID.Valid Then return
- IDEX.JPC=IFID.NPC[31-28],IFID.IR[25-0]<<2
- IDEX.DPC = IFID.DPC
- IDEX.imm = IFID.IR[31-20]
- IDEX.func = IFID.IR[14-12]
- IDEX.rdl = IFID.IR[11-7]
- IDEX.CW = controller(IFID.IR[6-0])
- If IDEX.CW.RegRead Then
- IDEX.rs1 = GPR[IFID.IR[19-15]].value
- *IDEX.rsl1 = 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]].value
  - *IDEX.rsl2 = IFID.IR[24-20]*
- GPR[IDEX.rdl].ins = CW.ins
- IFID.stall = false
- IDEX.Valid = true

We removed Stall Logic!

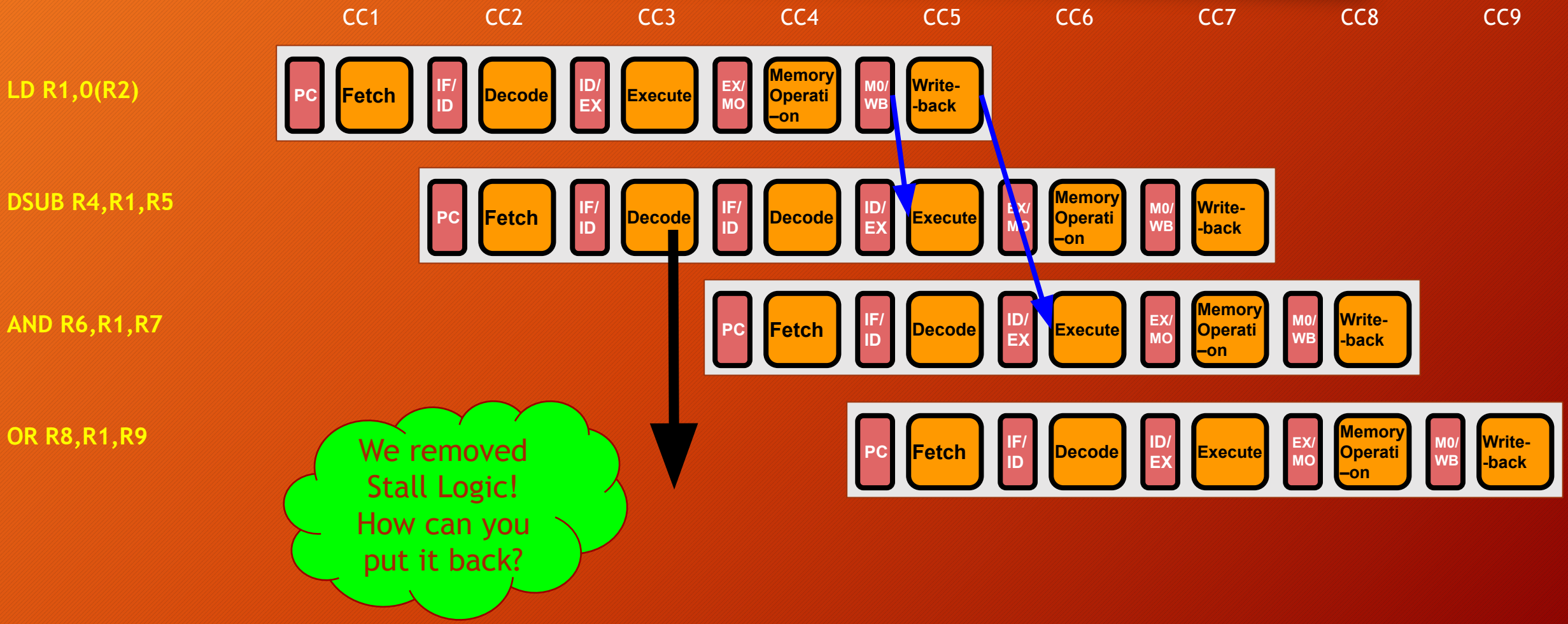
- If EXMO.stall or !IDEX.Valid Then return
- ALUSelect = ALUControl(IDEX.CW.ALUOp, IDEX.func)
- ALUINP1 = Forwarder(IDEX.rs1,IDEX.rsl1,EXMO.rdl,MOWB.rdl)
- ALUINP2 = Forwarder(IDEX.rs2,IDEX.rsl2,EXMO.rdl,MOWB.rdl)
- EXMO.ALUOUT = ALU(ALUSelect, ALUINP1, ALUINP2)
- ALUZeroFlag = (ALUINP1 == ALUINP2)
- 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 = false
- If IDEX.CW.Jump
- Then PC = IDEX.JPC
- IFID.Valid = false
- IDEX.Valid = false
- PC.Valid = false
- IDEX.stall = false
- EXMO.Valid = true

Can you think of the forwarder logic?

Operand Forwarding: Example-2



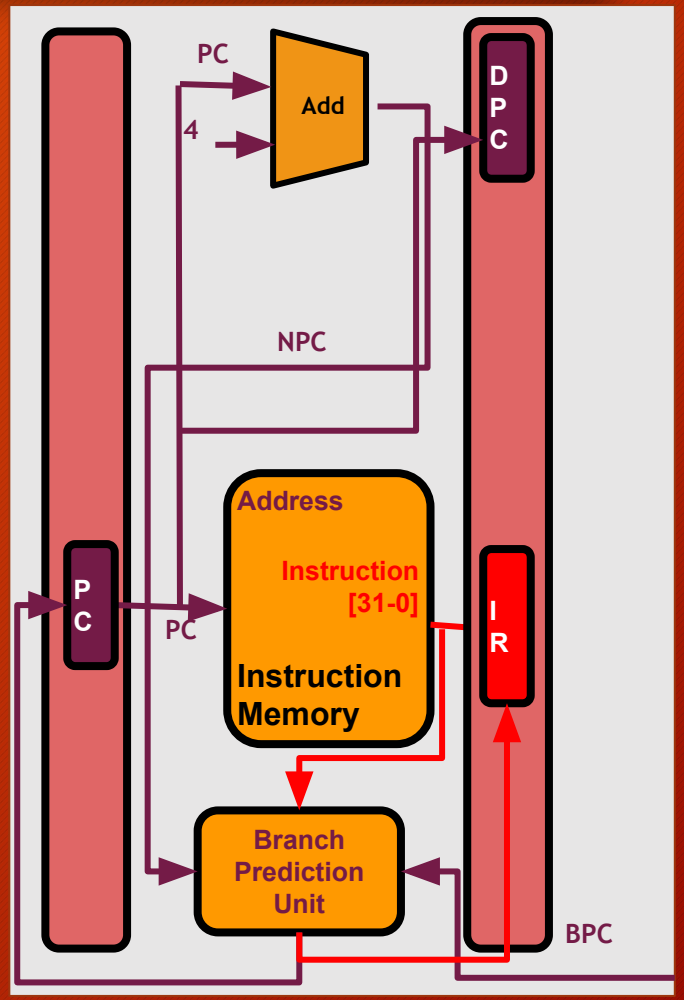
Stall Solves All Hazards!



Handling Pipeline Branch Penalties: Branch Delay Slot

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

Exceptions and Interrupts: Concepts

- Synchronous vs Asynchronous: Synchronous if the exception can be reproduced with same input and states at the same place in the program.
- User Request vs Coerced: User has requested for a particular action as compared to some hardware events
- Maskable vs Un-maskable: In an un-maskable exception/interrupt, the hardware has to take an action.
- Within vs Between Instructions: Is a particular instruction causing the exception/interrupt or it is generated between a set of instructions.
- Resume vs Terminate: What should be done when the exception/interrupt is taken care off by the hardware.

Exceptions and Interrupts: Categories

Types of Event	Synchronous or Asynchronous	Coerced or User request	Maskable or Non-maskable	Within vs Between	Resume vs Terminate	Pipeline Stage to Handle
IO Device Req	Asynchronous	Coerced	Non-maskable	Between	Resume	Any
Invoke OS	Synchronous	User Request	Non-maskable	Between	Resume	Any
Tracing Execution	Synchronous	User Request	Maskable	Between	Resume	Any
Breakpoint	Synchronous	User Request	Maskable	Between	Resume	Any
Integer Arithmetic Underflow or Overflow	Synchronous	Coerced	Maskable	Within	Resume	Execute
FP Underflow or Overflow	Synchronous	Coerced	Maskable	Within	Resume	Execute
Page fault	Synchronous	Coerced	Non-maskable	Within	Resume	Memory Operation or Fetch
Misaligned Memory Access	Synchronous	Coerced	Maskable	Within	Resume	Memory Operation or Fetch
Memory Protection Violation	Synchronous	Coerced	Non-maskable	Within	Resume	Memory Operation or Fetch
Undefined Instruction	Synchronous	Coerced	Non-maskable	Within	Terminate	Decode
Hardware Malfunction	Asynchronous	Coerced	Non-maskable	Within	Terminate	Any
Power Failure	Asynchronous	Coerced	Non-maskable	Within	Terminate	Any

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?