Pipelining

Prof. James L. Frankel
Harvard University

Version of 11:11 PM 1-Dec-2021

Single-Cycle CPU Implementation

- The clock cycle must be long enough to accommodate the longest instruction
- There is no overlapping of the execution of instructions
- With time-consuming instructions (floating point, memory access), the penalty is high

Pipelining

- Decreases the average number of cycles per instruction (CPI) – throughput
- Does not decrease the time for any single instruction – latency

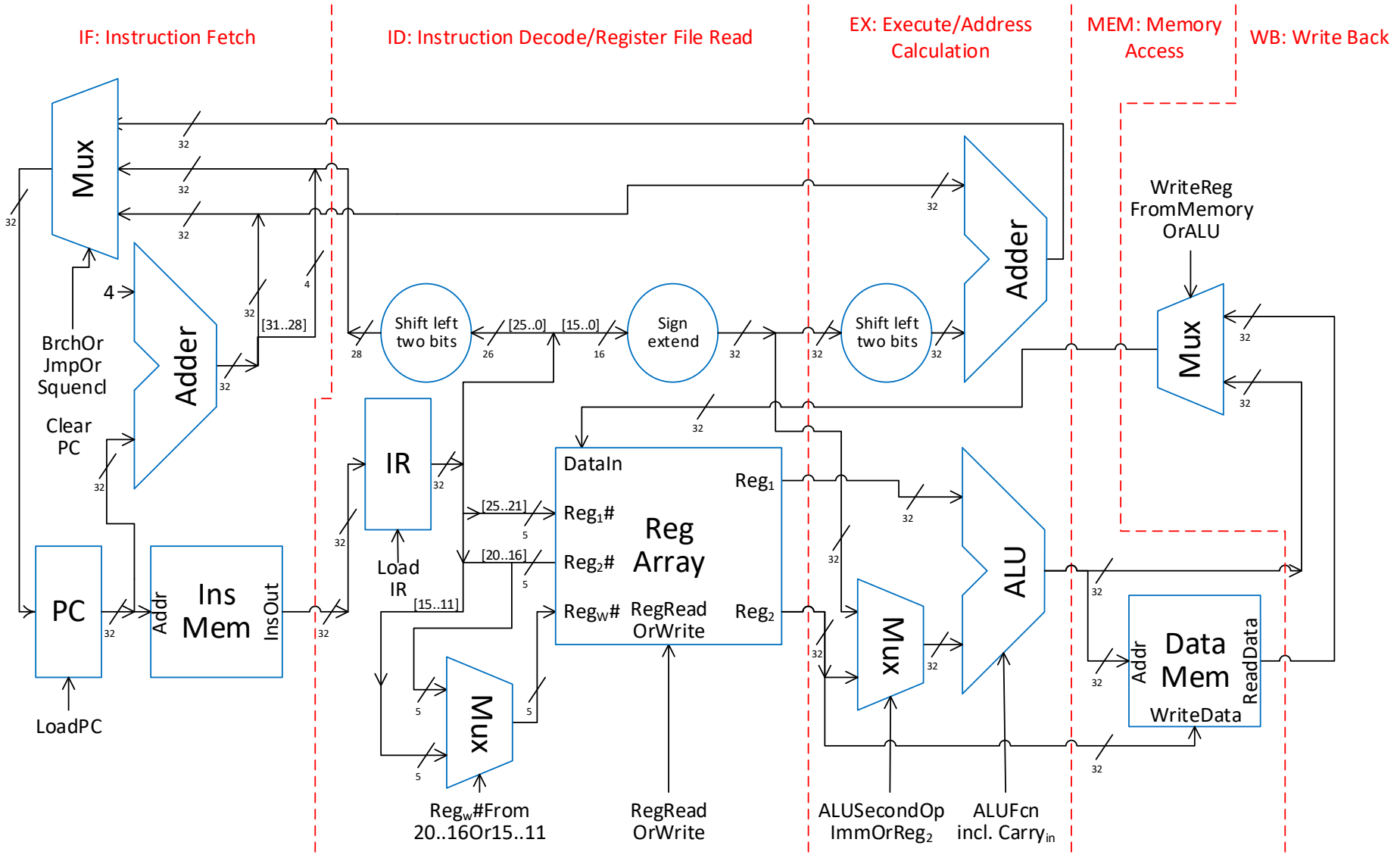
Memory Organization

- We will allow the assumption that there are “two” memories – one for instructions and one for data
- This is not *really* the case
- We will have an instruction cache and a data cache
  - These will create the illusion that we have two separate memories

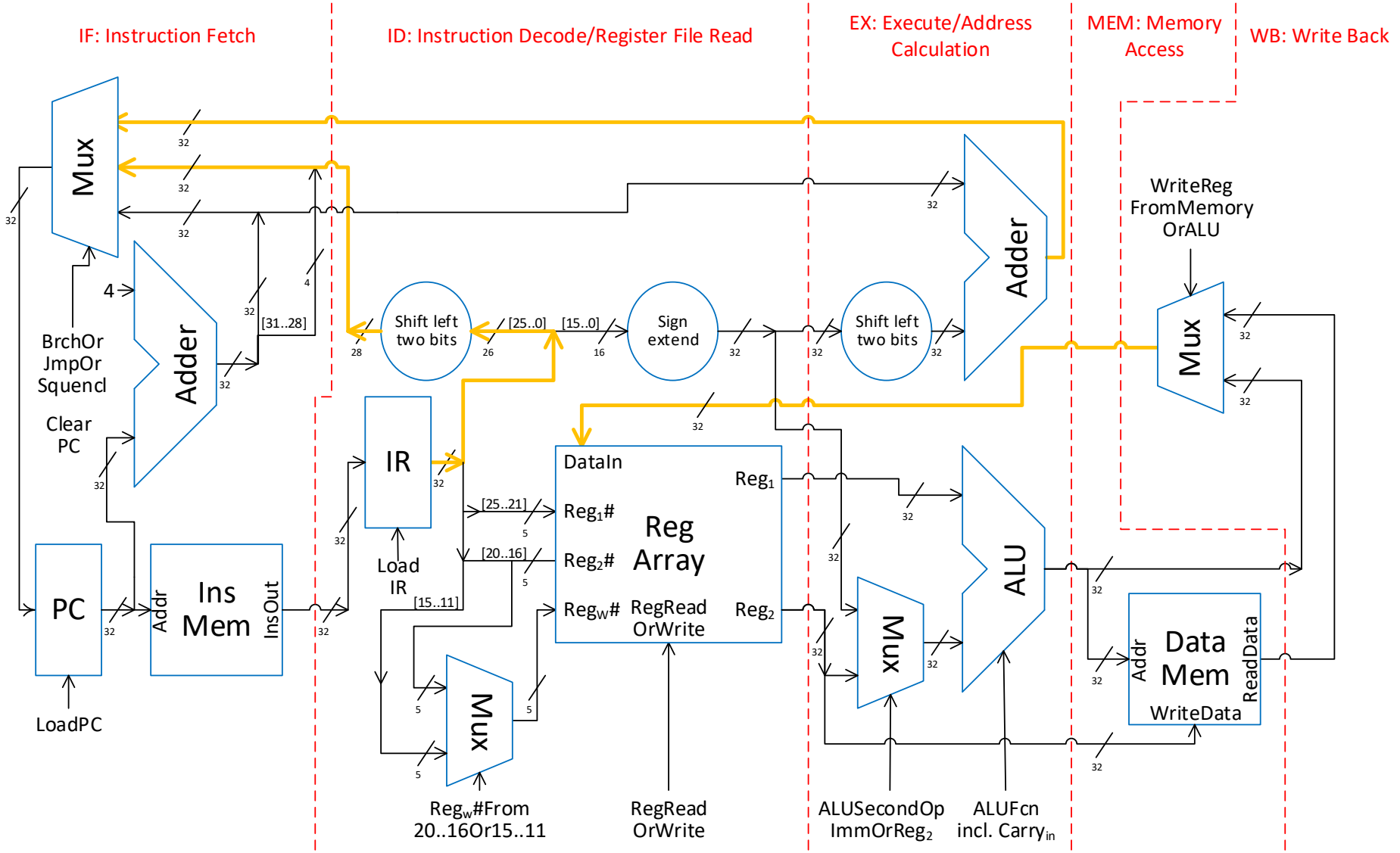
Instruction Set Design for Pipelining

- All instructions are the same length
  - x86 ISA has 1 to 15 bytes per instruction
- Small number of instruction formats
  - Operands can be fetched from registers before determining the instruction type
- Load/store architecture means that the EX (ALU) stage can compute the memory address
  - If operands could come from memory, then the memory access would need to precede the EX stage
- All memory accesses are aligned
  - Only a single memory access is required

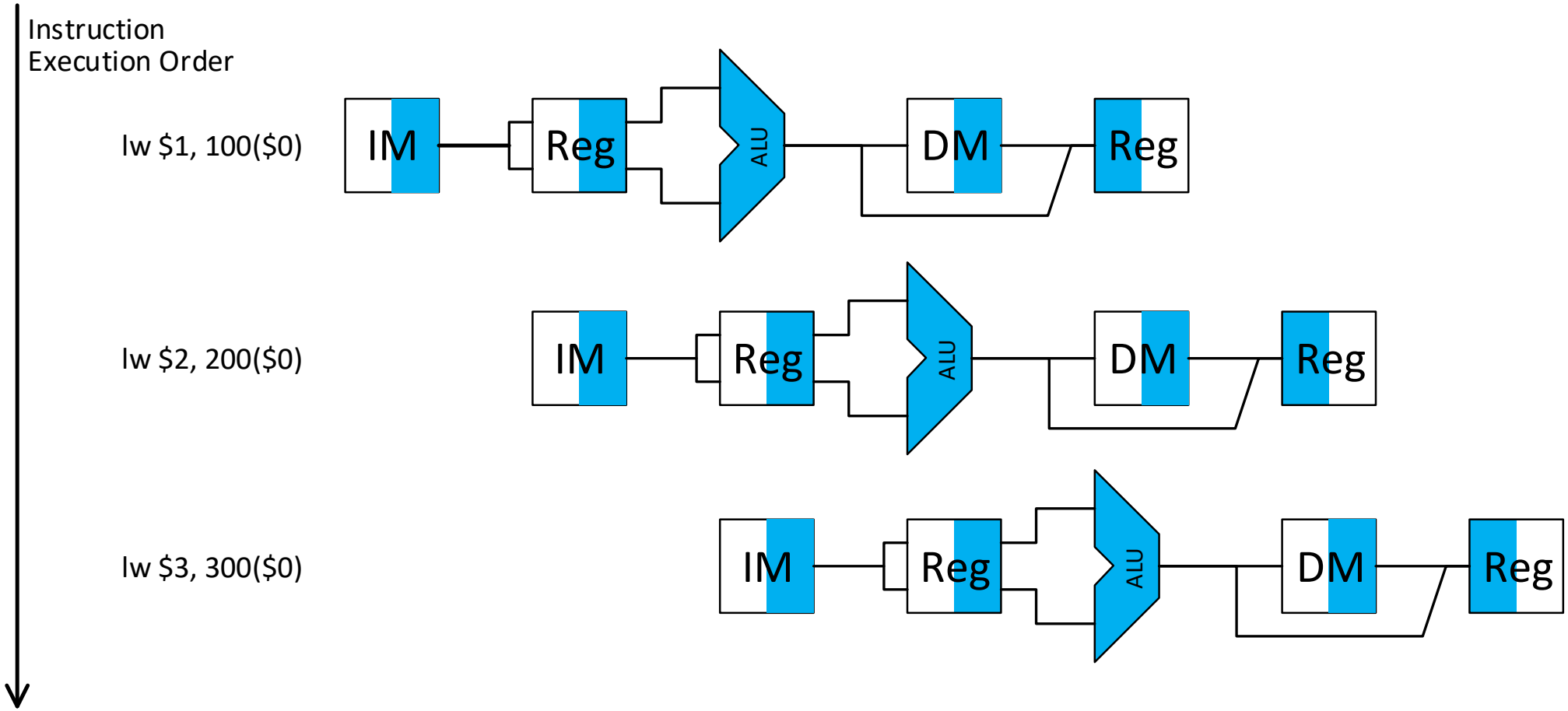
Data Path Split into Pipeline Stages


IF: Instruction Fetch
ID: Instruction Decode/Register File Read
EX: Execute/Address Calculation
MEM: Memory Access
WB: Write Back
Mux
Adder
4
BrchOr
JmpOr
Squencl
Clear
PC
[31..28]
PC
LoadPC
Addr
Ins
Mem
InsOut
Shift left two bits
[25..0]
[15..0]
Sign extend
IR
Load
IR
[25..21]
[20..16]
[15..11]
DataIn
Reg1#
Reg2#
Regw#
Reg
Array
RegRead
OrWrite
Reg1
Reg2
Mux
Regw#From
20..16Or15..11
RegRead
OrWrite
Shift left two bits
Adder
ALU
Mux
ALUSecondOp
ImmOrReg2
ALUFcn
incl. Carryin
WriteReg
FromMemory
OrALU
Mux
Addr
Data
Mem
ReadData
WriteData

Right to Left Data Paths are Problematic

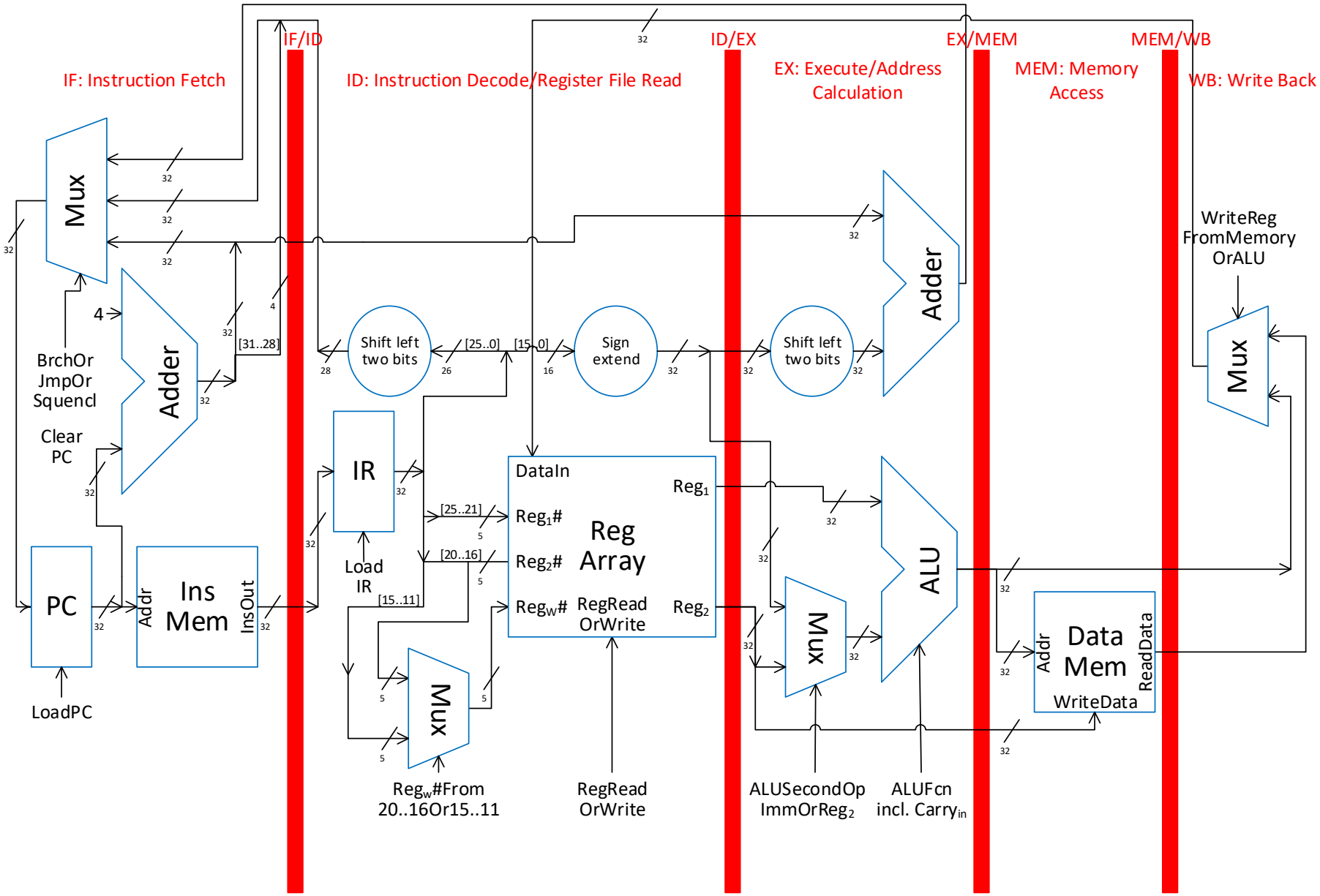

IF: Instruction Fetch
ID: Instruction Decode/Register File Read
EX: Execute/Address Calculation
MEM: Memory Access
WB: Write Back
Mux
Adder
4
BrchOr
JmpOr
Squencl
Clear
PC
[31..28]
PC
LoadPC
Addr
Ins
Mem
InsOut
IR
Load
IR
Shift left
two bits
[25..0]
[15..0]
Sign
extend
Shift left
two bits
Adder
WriteReg
FromMemory
OrALU
Mux
DataIn
Reg1
Reg1#
Reg2#
Regw#
RegRead
OrWrite
Reg2
Reg
Array
[25..21]
[20..16]
[15..11]
Mux
Regw#From
20..16Or15..11
RegRead
OrWrite
Mux
ALU
ALUSecondOp
ImmOrReg2
ALUFcn
incl. Carryin
Addr
Data
Mem
ReadData
WriteData

Pipelining Three lw Instructions

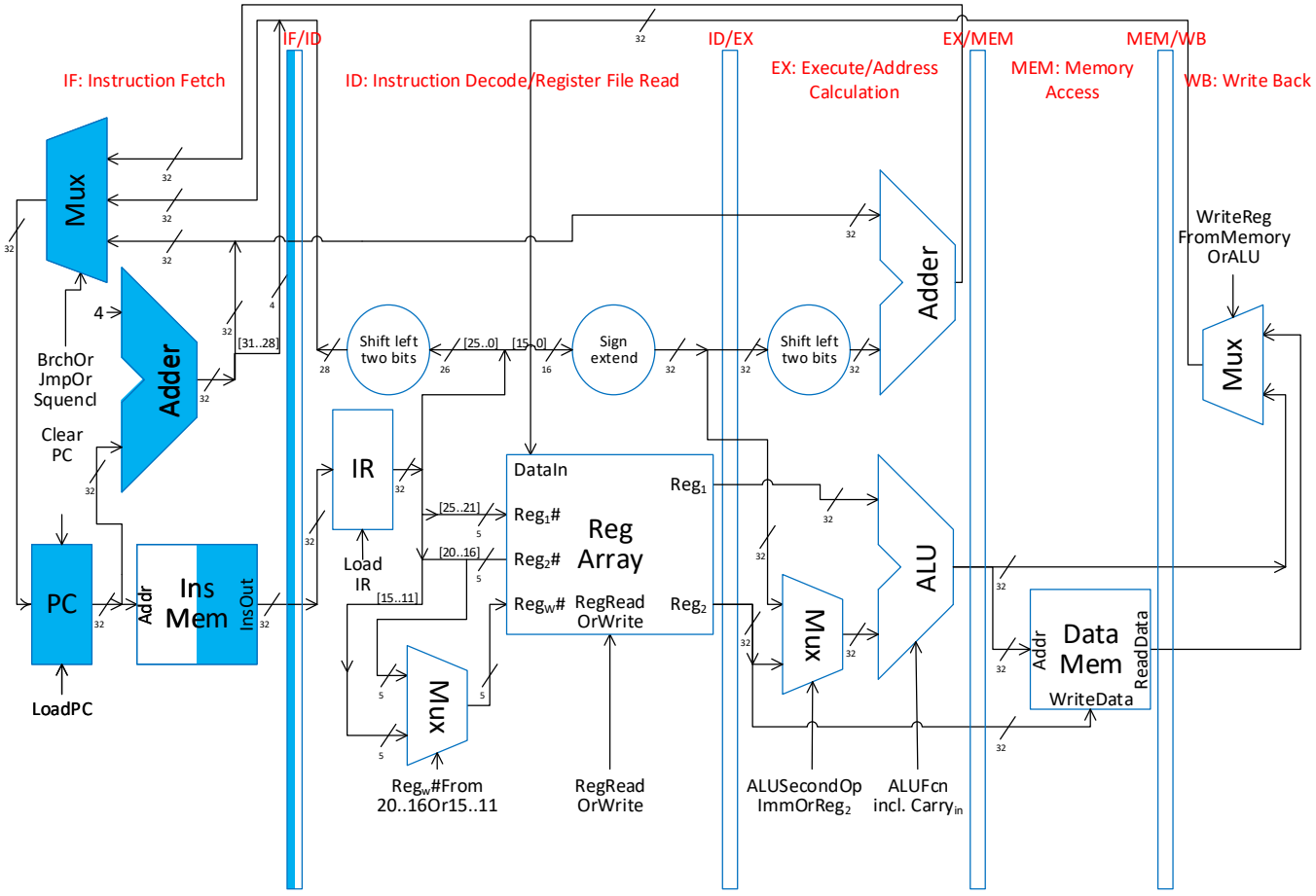


Staging the Pipelined Data

- Add pipeline registers between stages
- Allows the pipeline stages to run independently within the same clock cycle


IF/ID
ID/EX
EX/MEM
MEM/WB
IF: Instruction Fetch
ID: Instruction Decode/Register File Read
EX: Execute/Address Calculation
MEM: Memory Access
WB: Write Back
Mux
Adder
4
BrchOr
JmpOr
Squencl
Clear
PC
[31..28]
Shift left two bits
[25..0]
[15..0]
Sign extend
Shift left two bits
Adder
WriteReg
FromMemory
OrALU
Mux
IR
Load
IR
DataIn
Reg1
Reg1#
Reg2#
Regw#
RegRead
OrWrite
Reg2
Reg Array
[25..21]
[20..16]
[15..11]
PC
LoadPC
Addr
Ins
Mem
InsOut
Mux
Regw#From
20..16Or15..11
RegRead
OrWrite
ALU
Mux
ALUSecondOp
ImmOrReg2
ALUFcn
incl. Carryin
Addr
Data
Mem
ReadData
WriteData

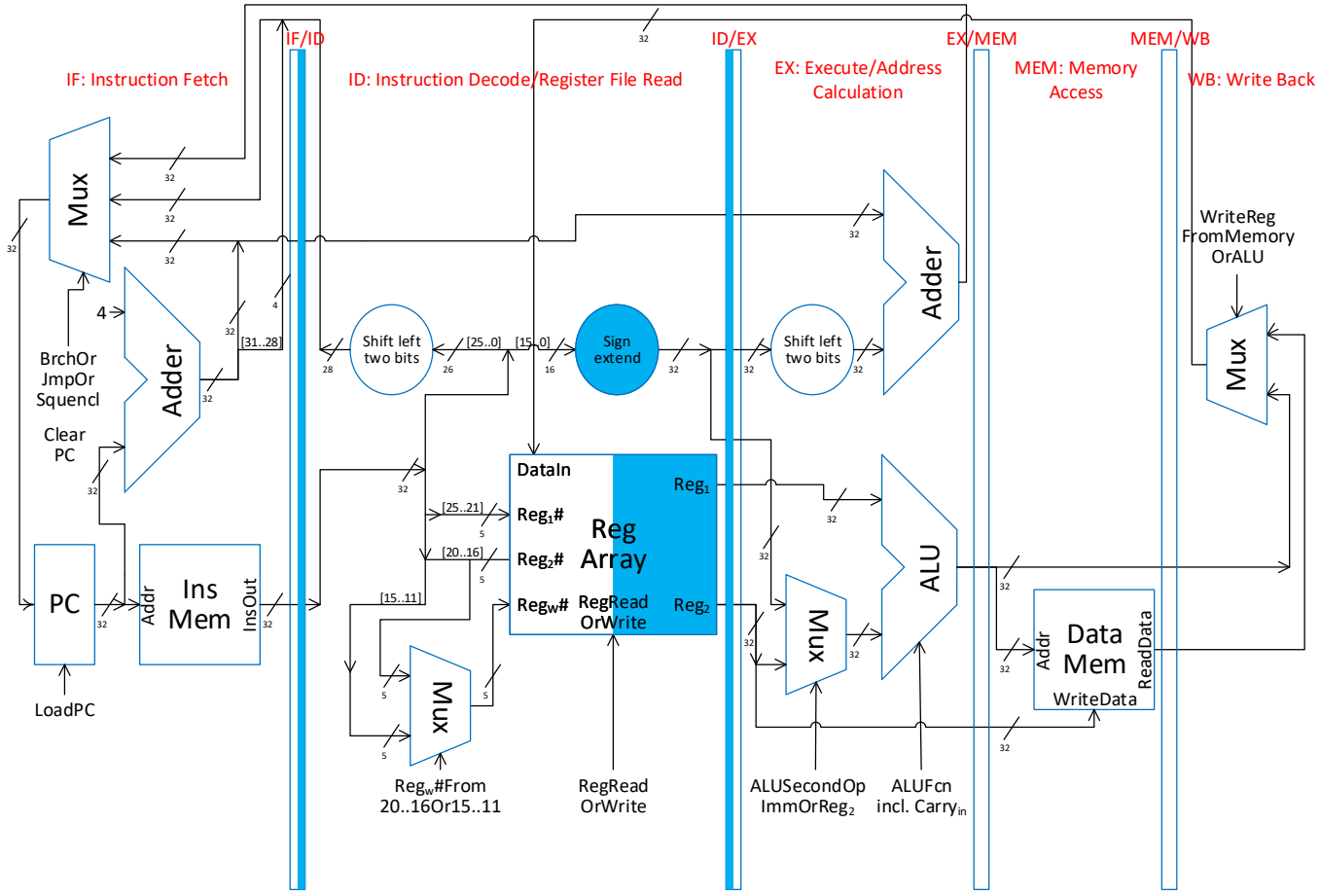
lw Instruction in IF Pipeline Stage



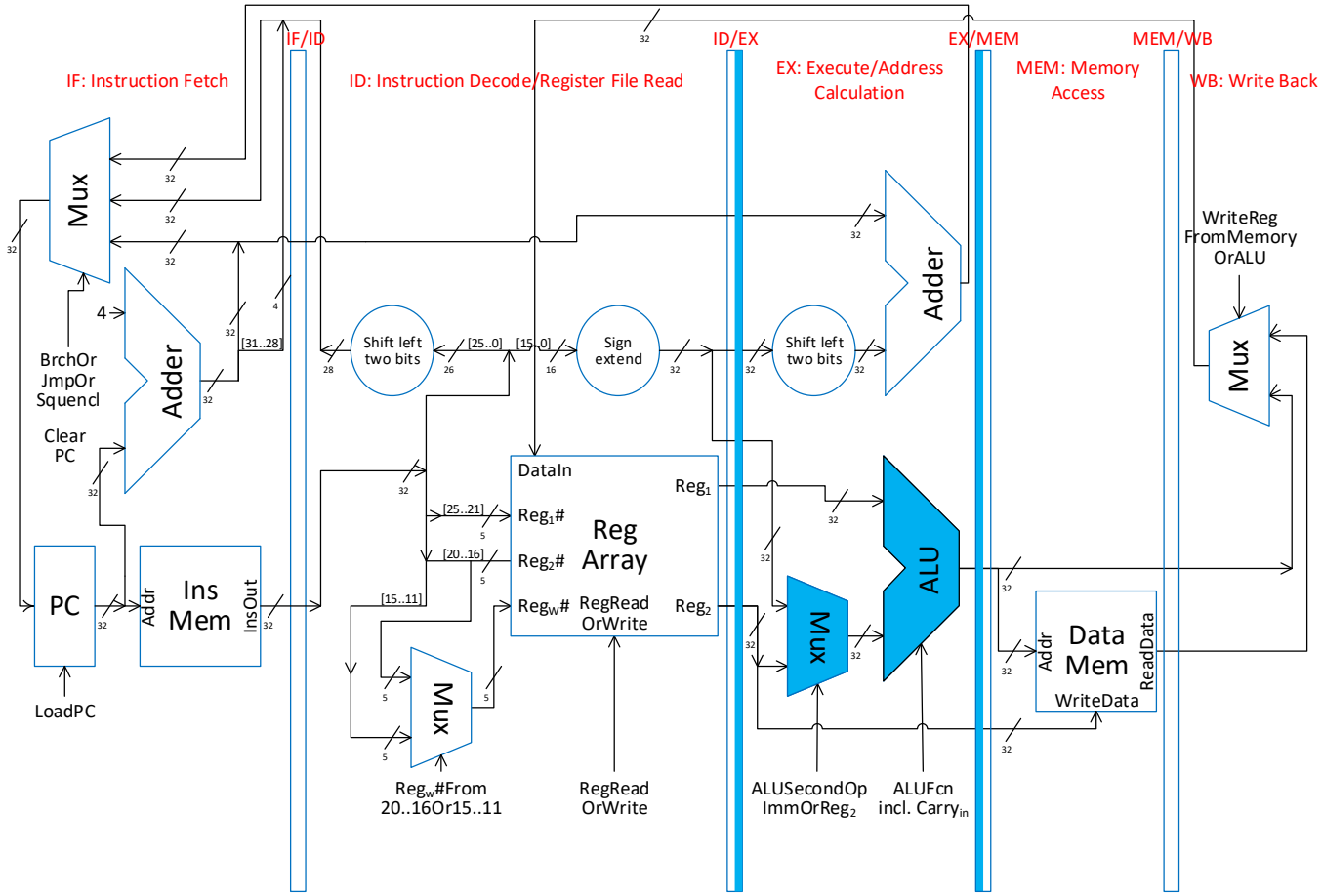
ID (Instruction Decode) Stage

- No need to keep the IR as a register because the current instruction is stored in the IF/ID pipeline register

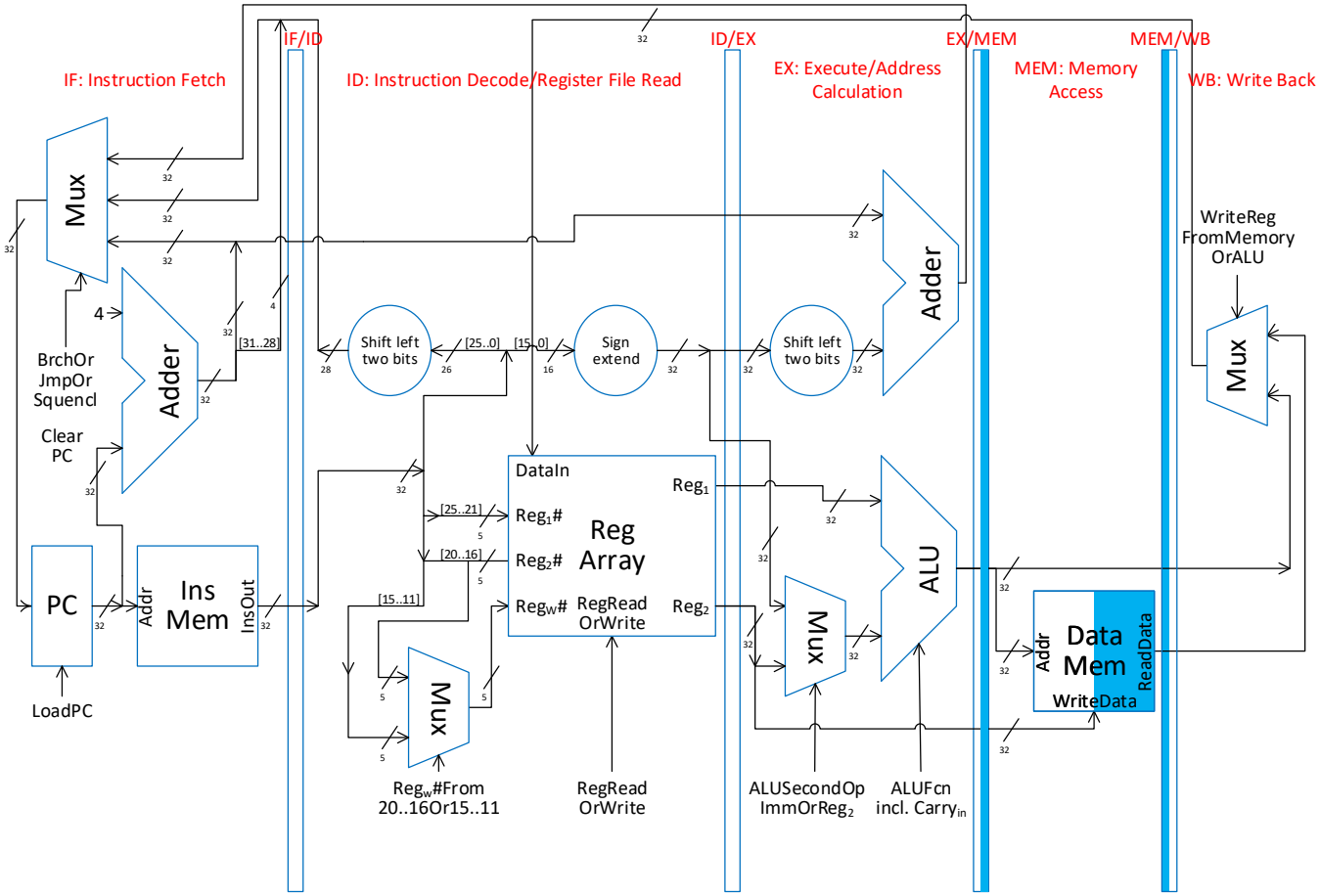
lw Instruction in ID Pipeline Stage



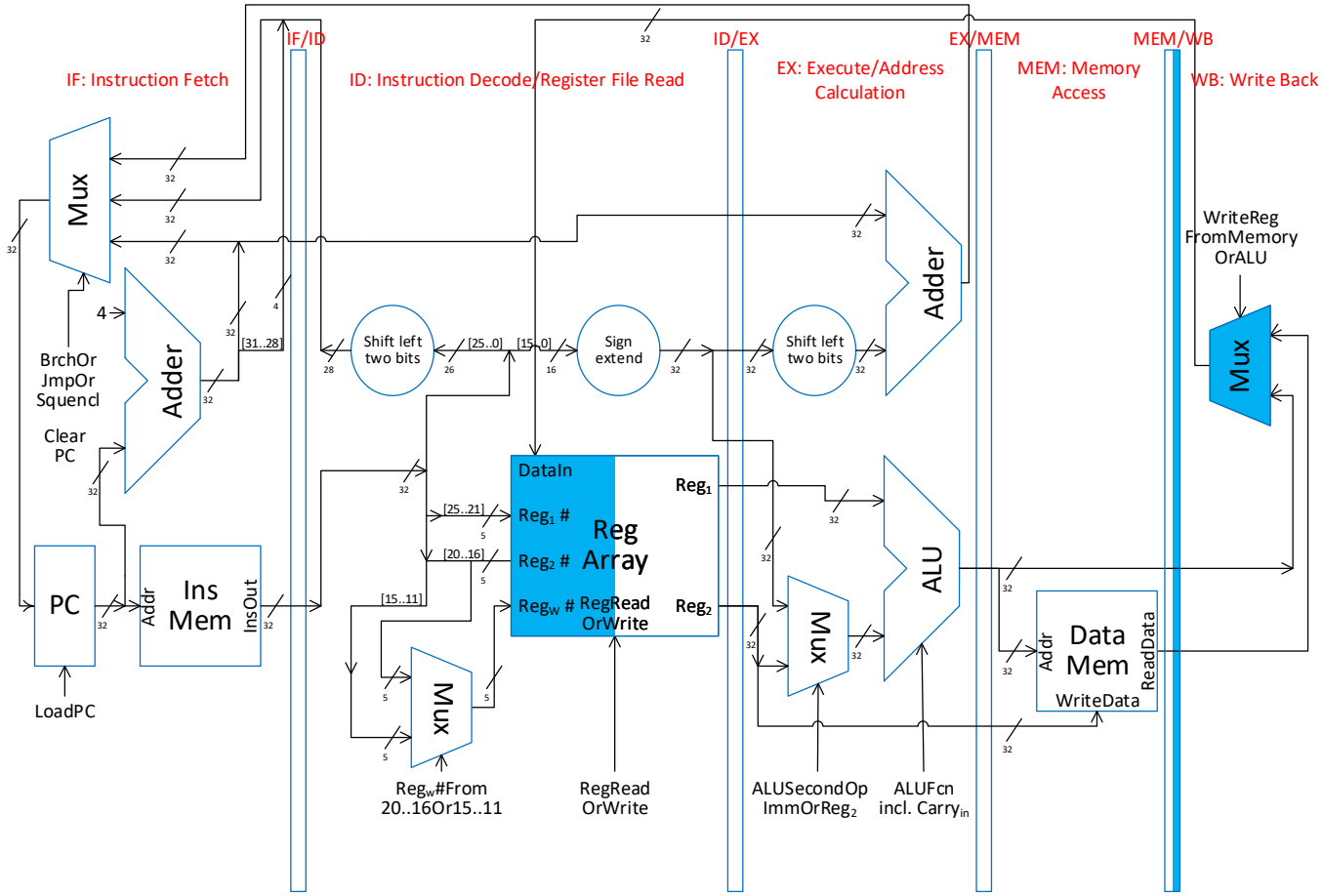
lw Instruction in EX Pipeline Stage



lw Instruction in MEM Pipeline Stage



lw Instruction in WB Pipeline Stage

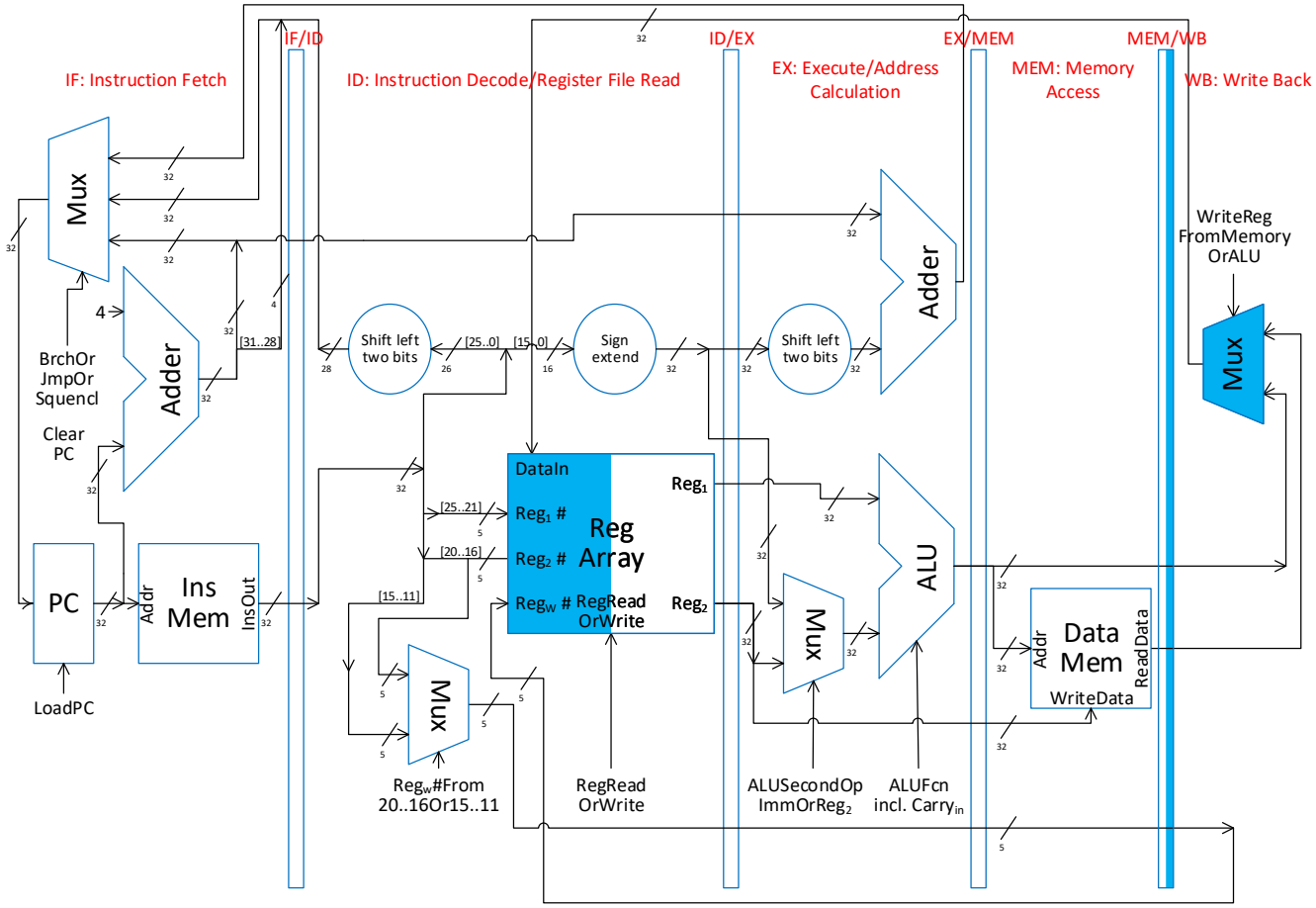


Problem with lw in WB Pipeline Stage

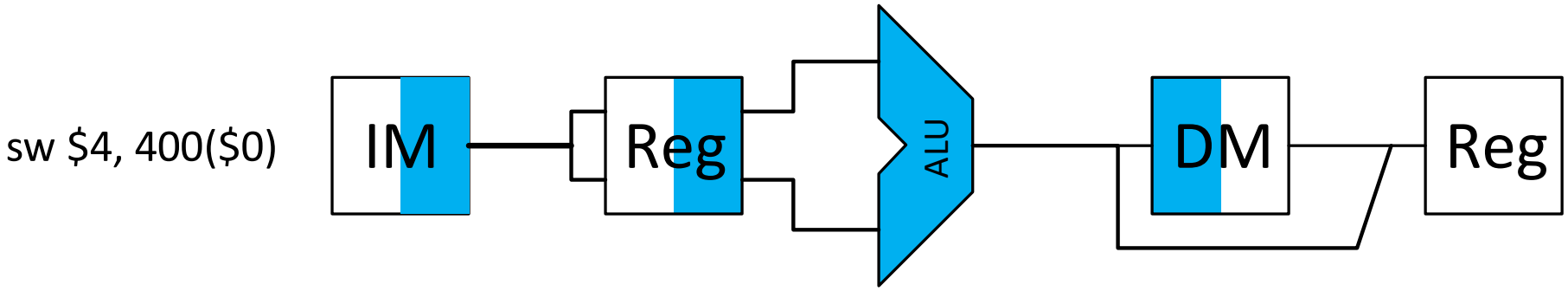
- Unfortunately, the incorrect register is written in the WB stage
- The Reg_w# is coming from the wrong pipeline stage!

- Let's correct the problem by keeping the Reg_w# with the value that will be written to that register

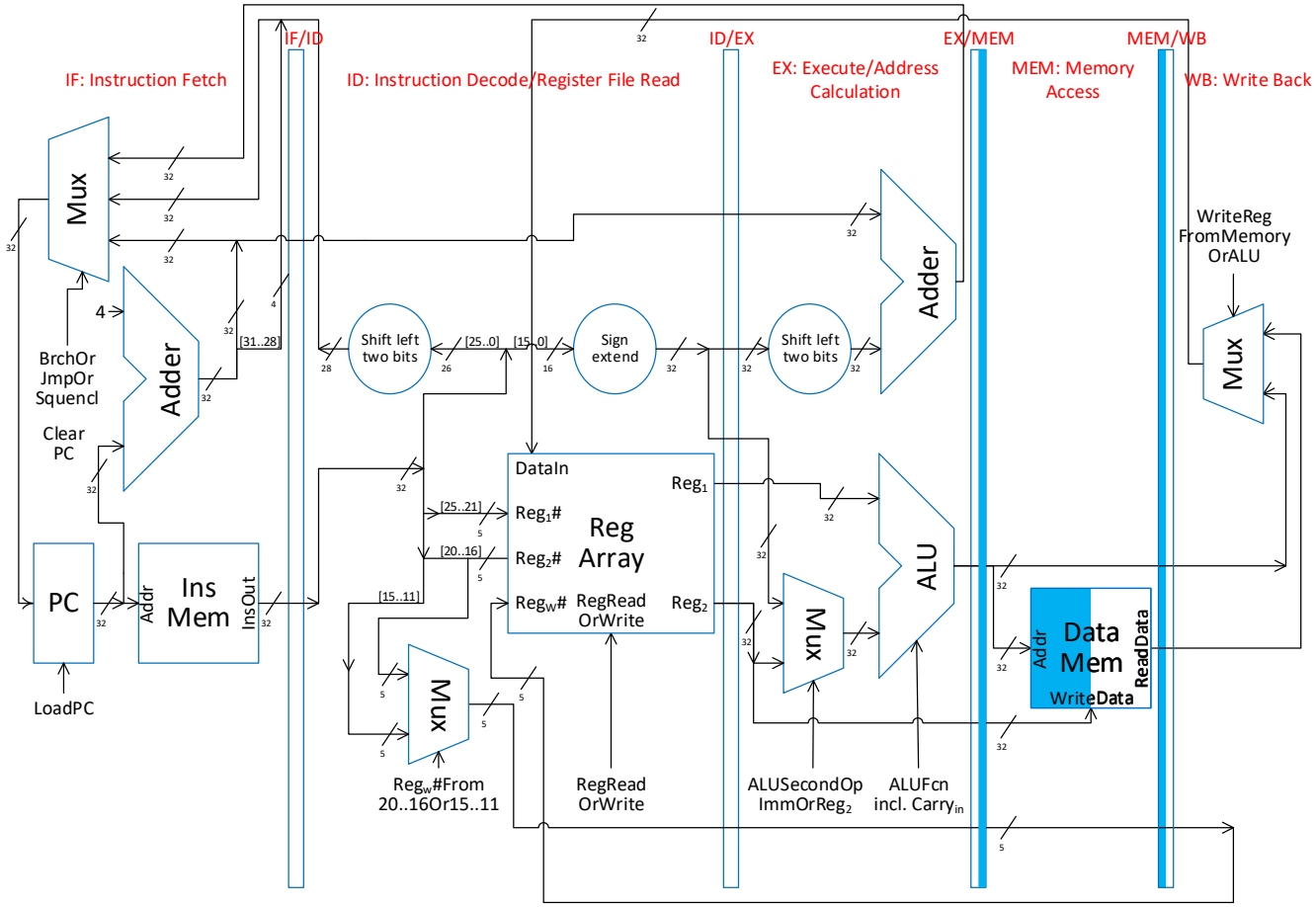
Corrected Pipeline Diagram for lw Instruction in WB Pipeline Stage



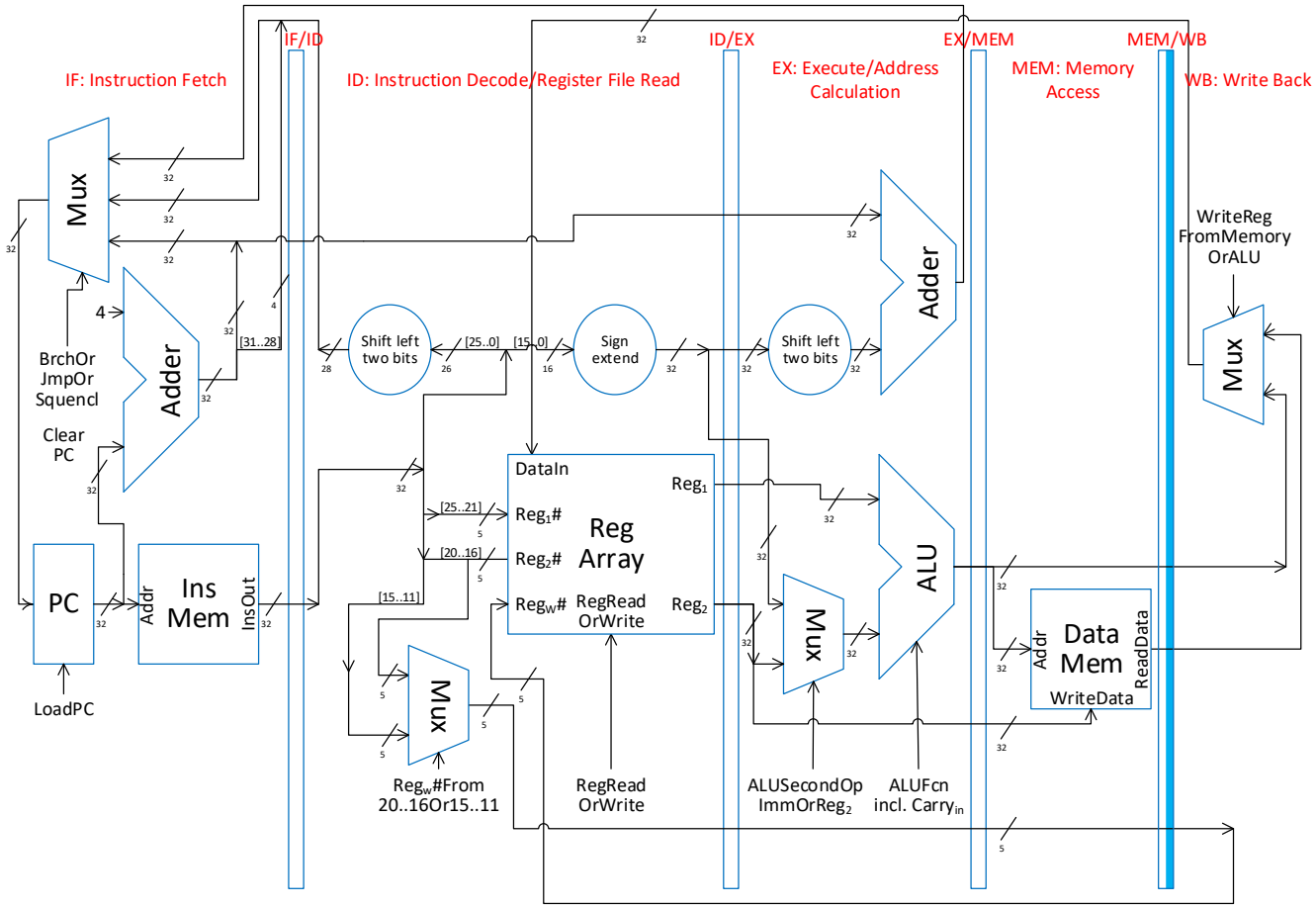
Executing an sw Instruction



sw Instruction in MEM Pipeline Stage



sw Instruction in WB Pipeline Stage

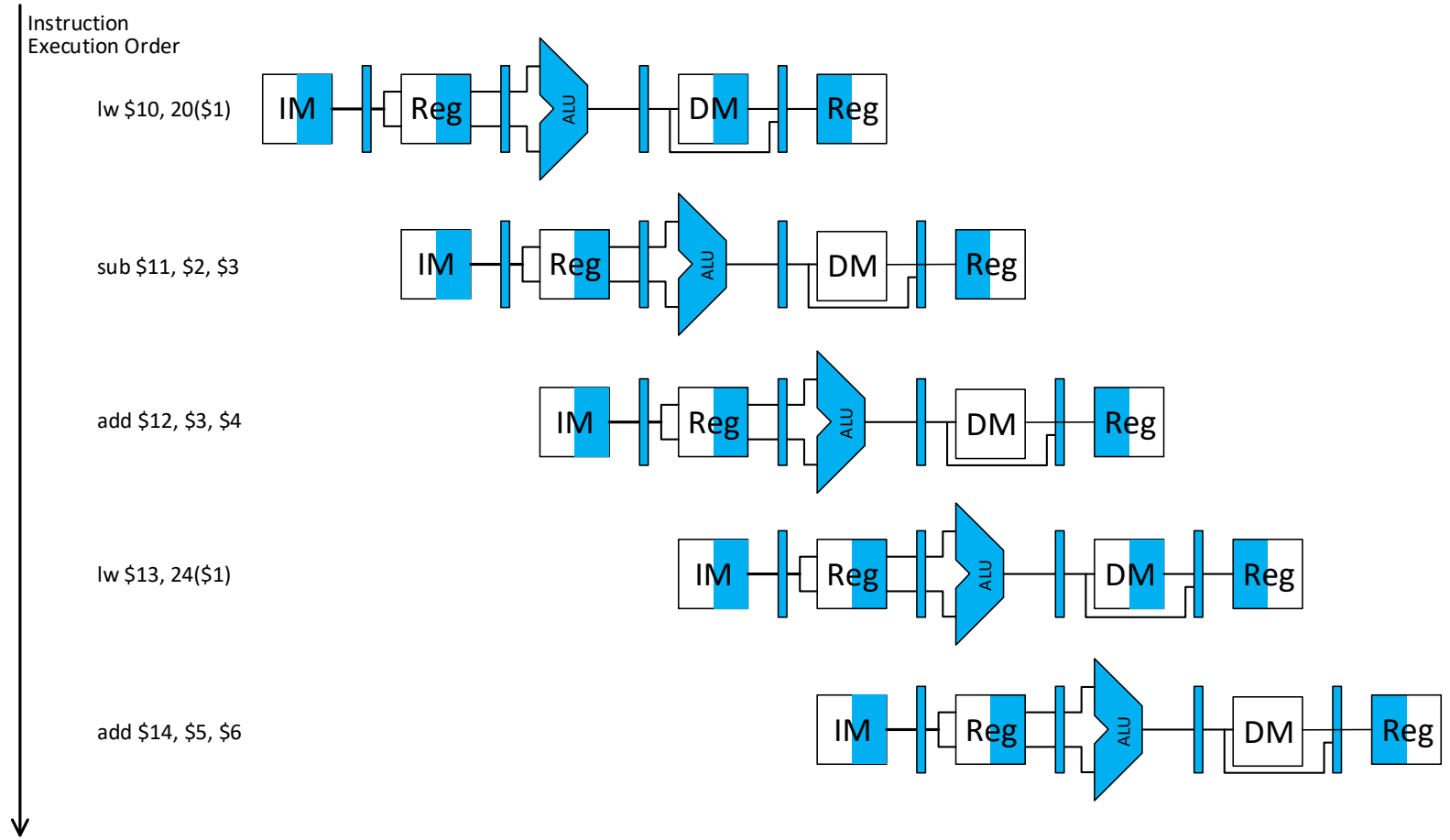


Sequence of Instructions

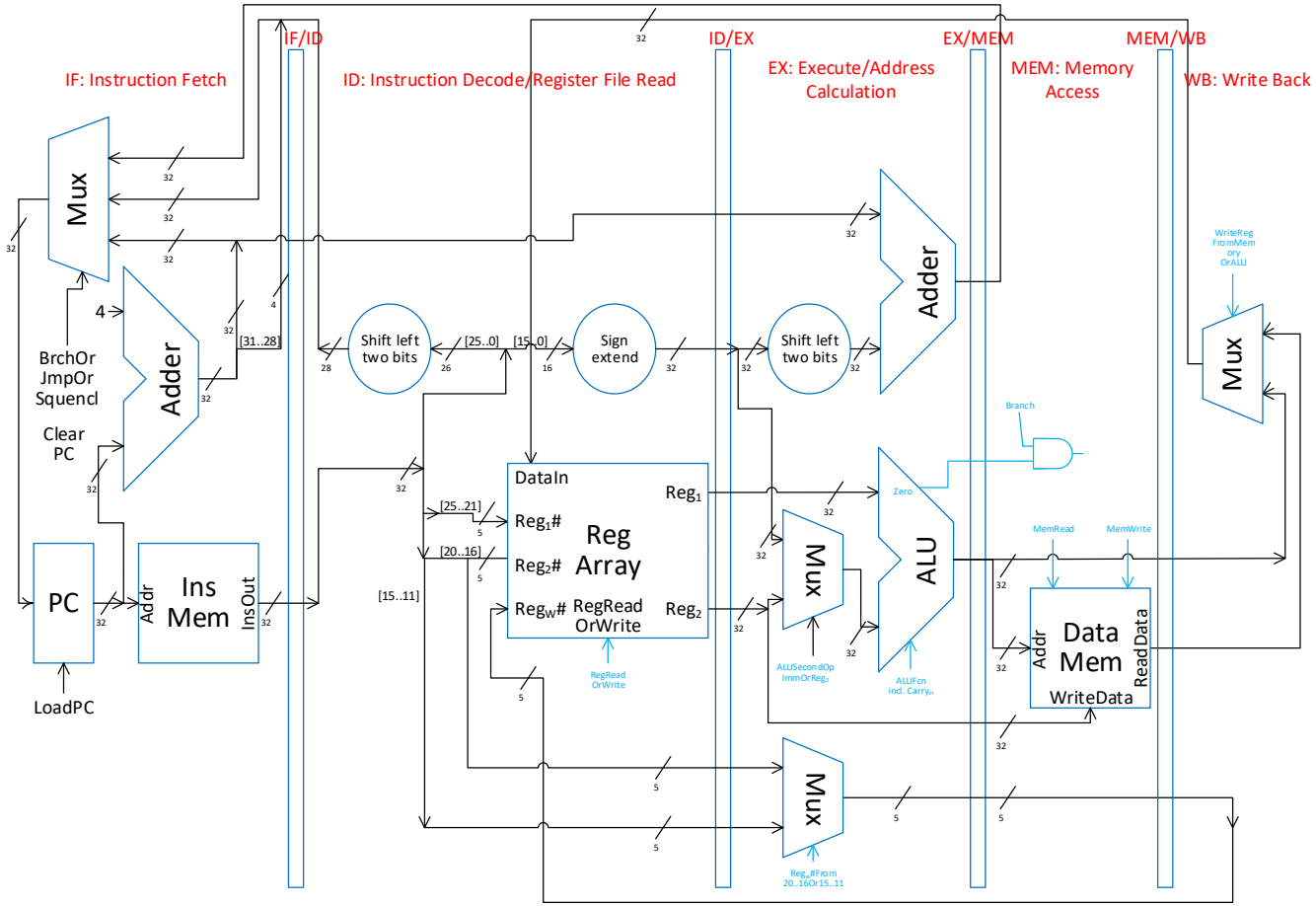
- Imaging the pipelined execution of a sequence of five instructions:

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

Multi-clock-cycle Pipeline Diagram of Above Instructions with Pipeline Registers



Identification of Control Lines

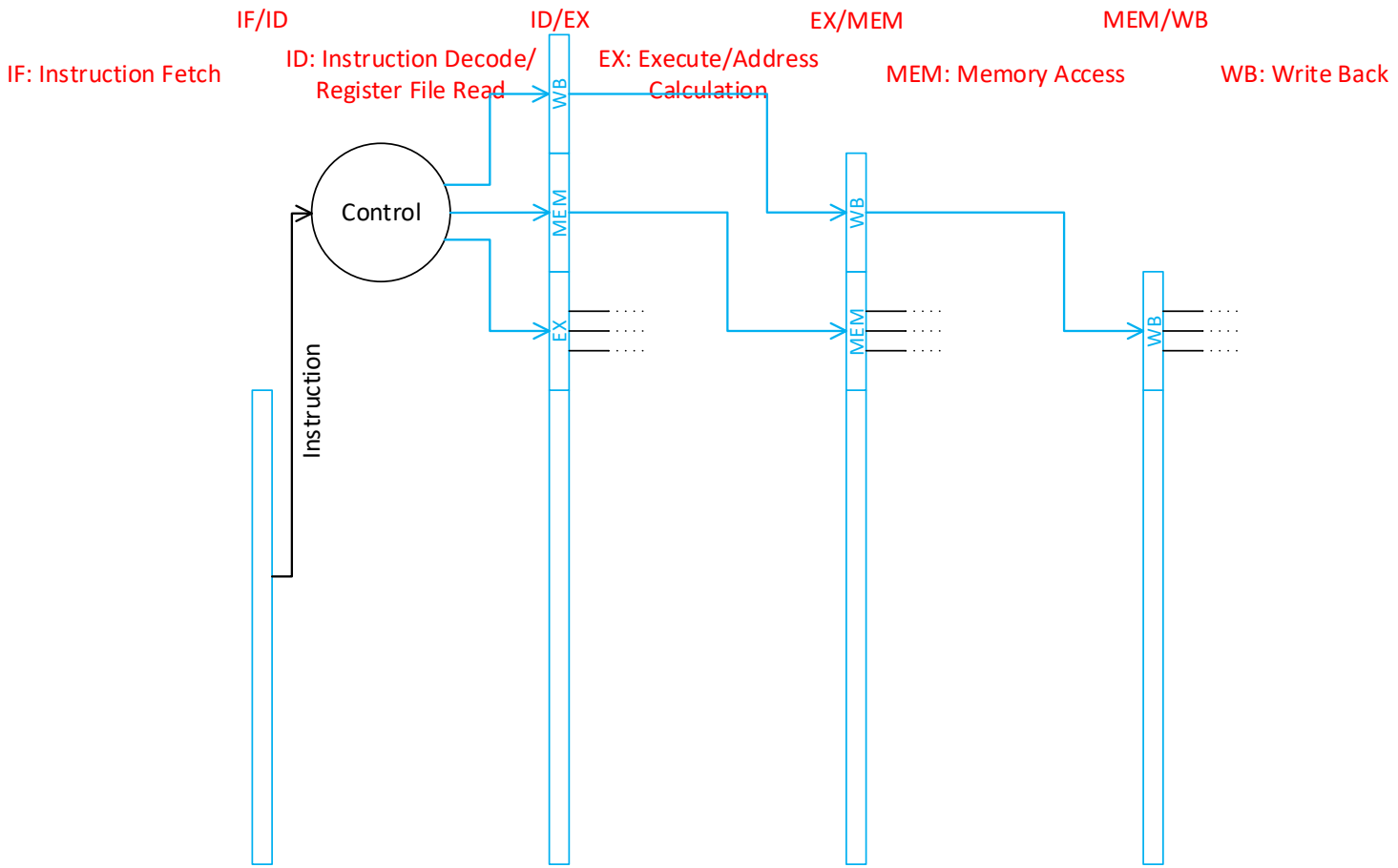


Problem with Control Lines

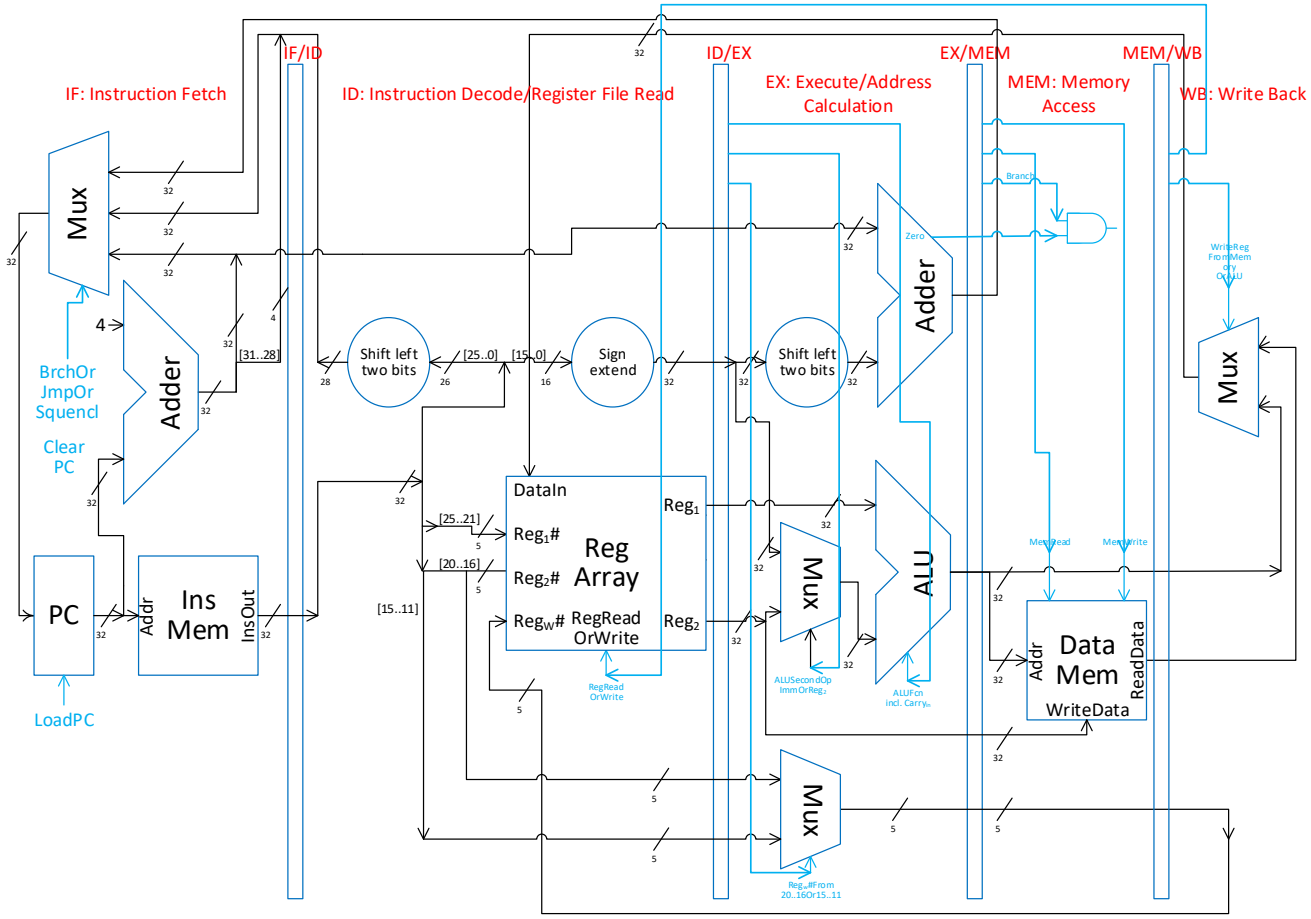
- Each instruction at each stage of the pipeline needs to have its specific control lines asserted
- Difficult problem to manage the state of each instruction in each stage

- However, what if we stage the control lines with the pipelined data!

Pipelined Control Summary



Pipelined Control Detail

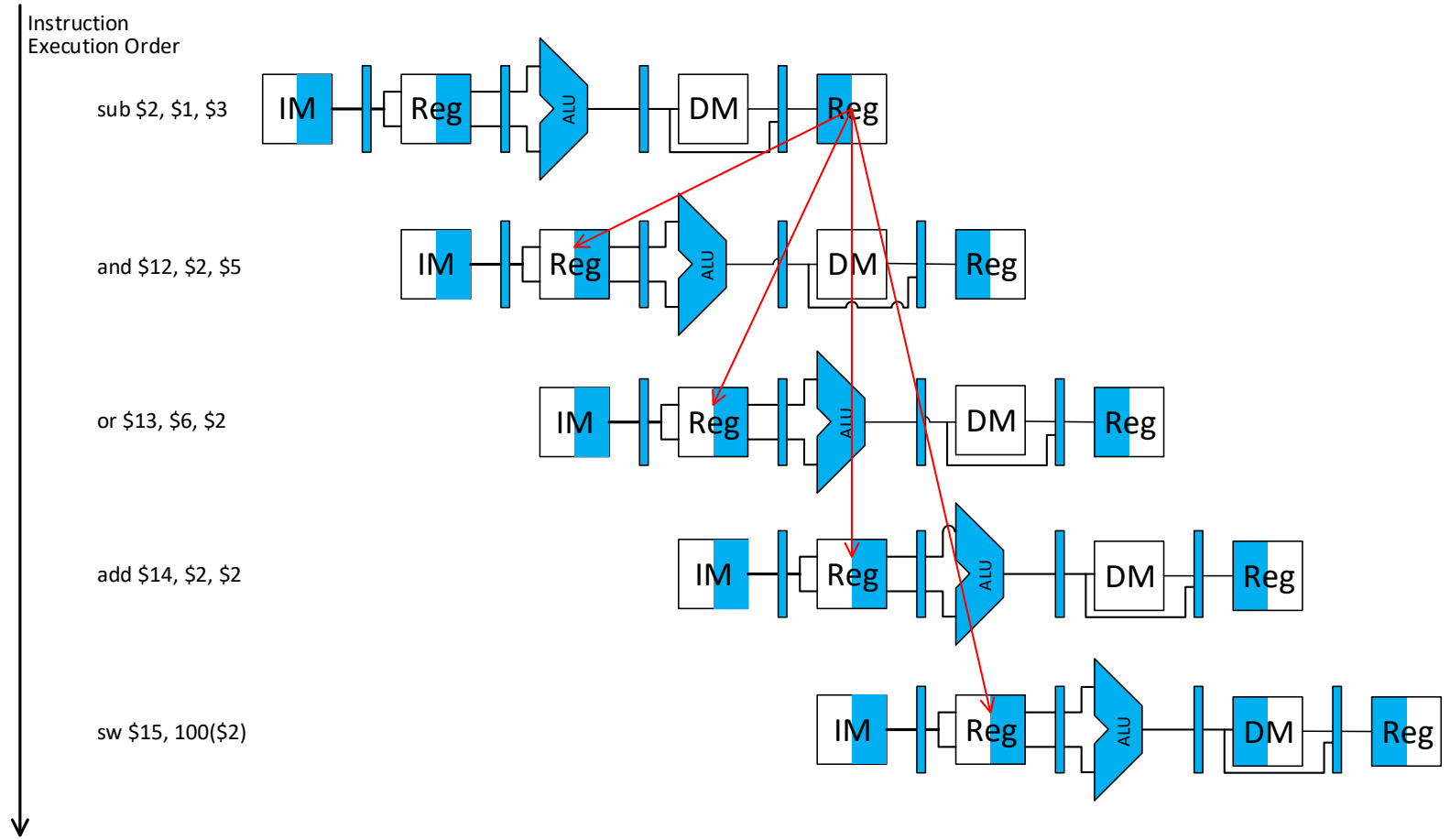


Sequence of Instructions with Dependencies

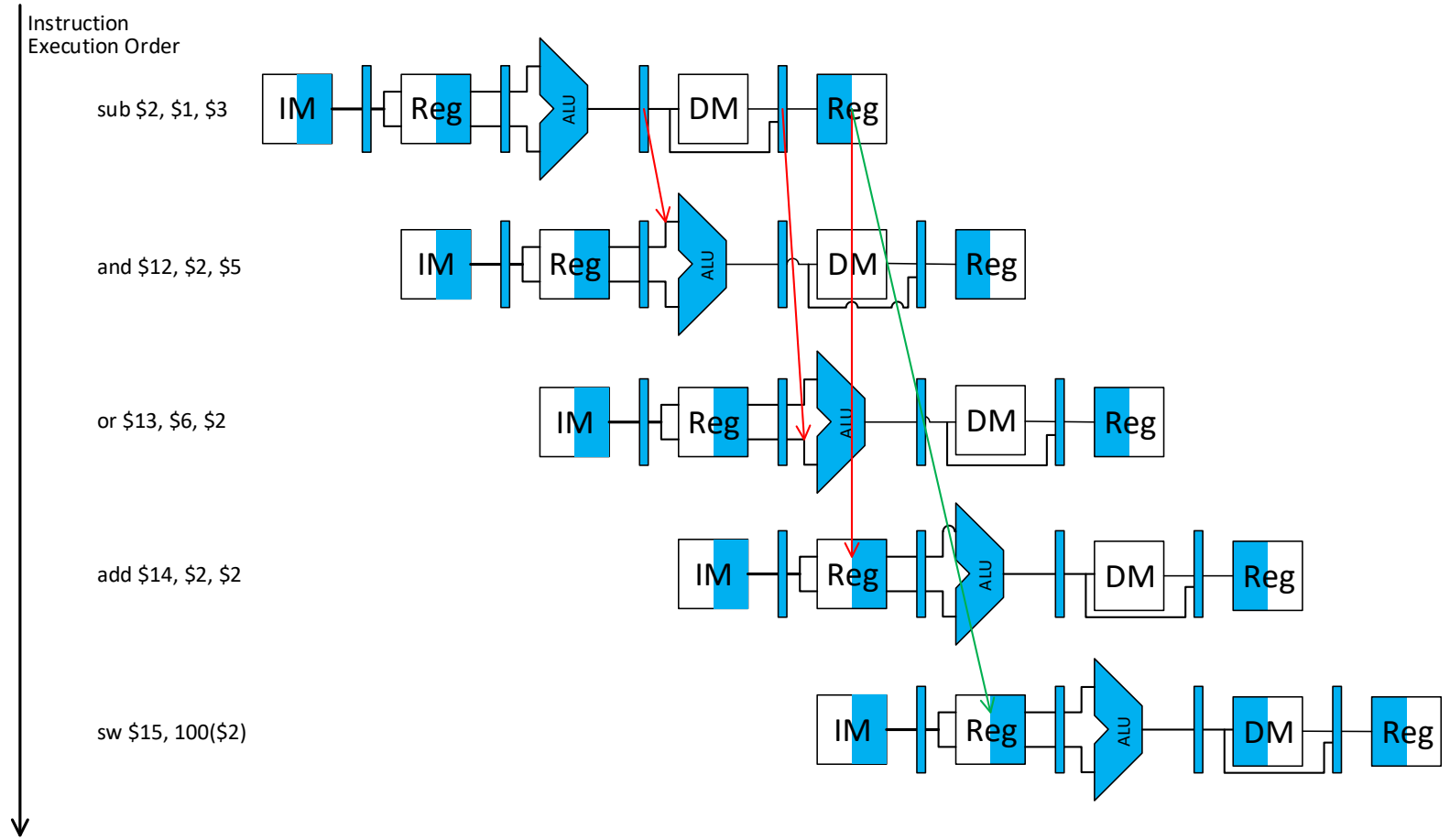
- Pipelined execution of a sequence of five instructions with dependencies:

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Multi-clock-cycle Pipeline Diagram of Above Instructions with Dependencies



Multi-clock-cycle Pipeline Diagram of Above Instructions with Dependencies & Forwarding



Forwarding

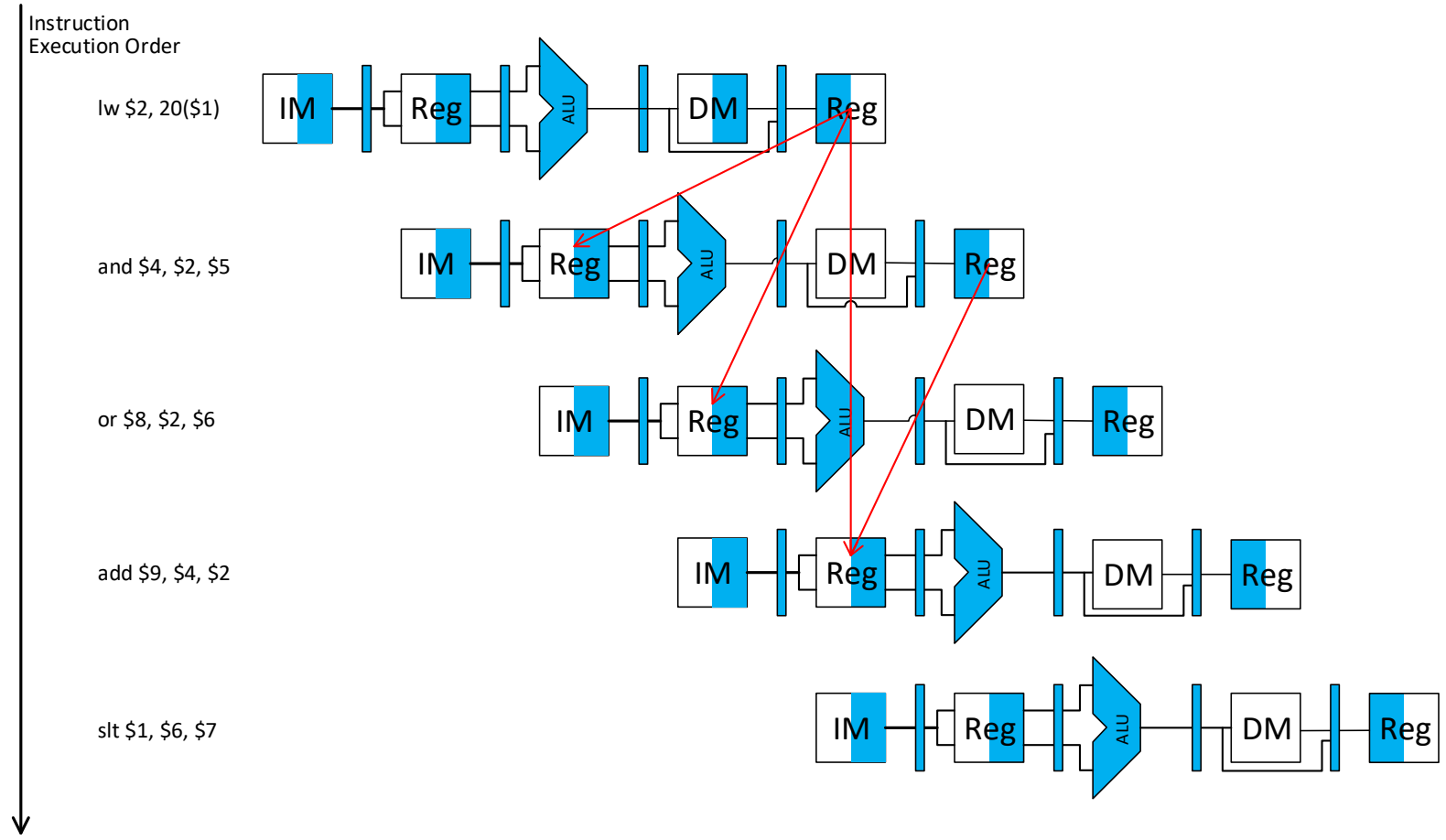
- It is the responsibility of the Forwarding Unit to determine how to forward available values to where they are used

Sequence of Instructions with Hazards & Stalls

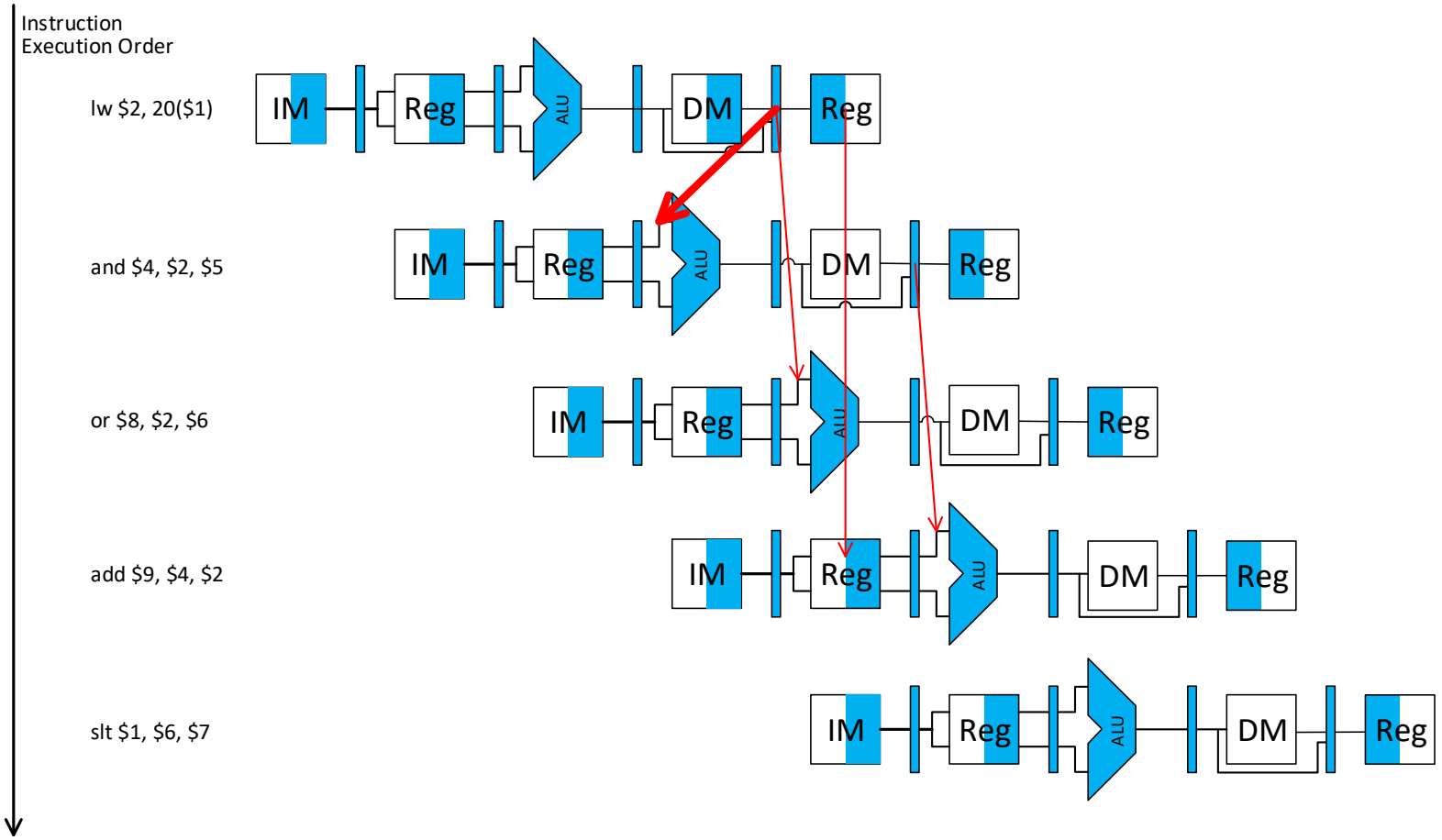
- Pipelined execution of a sequence of five instructions with dependencies:

```
lw    $2, 20($1)
and   $4, $2, $5
or    $8, $2, $6
add   $9, $4, $2
slt   $1, $6, $7
```

Multi-clock-cycle Pipeline Diagram of Above Instructions with Hazards & Stalls



Multi-clock-cycle Pipeline Diagram of Above Instructions with Hazards & Stalls & Forwarding



Forwarding Does Not Solve All Our Problems

- We really need the value of **$2** from
  lw $2, 20($1)
  in
  and $4, $2, $5
  before it is available

- This can't be corrected by using forwarding
- We need to delay the execution of **and $4, $2, $5** for one clock cycle until **$2** is available

Hazard Detection Unit

- It is the responsibility of the Hazard Detection Unit to determine if a stall is required

- The Hazard Detection Unit inserts bubbles
- A bubble causes the CPU to perform no operation
  - We refer to this as a nop (pronounced: no op)

Multi-clock-cycle Pipeline Diagram of Above Instructions with Hazards & Stalls Inserted