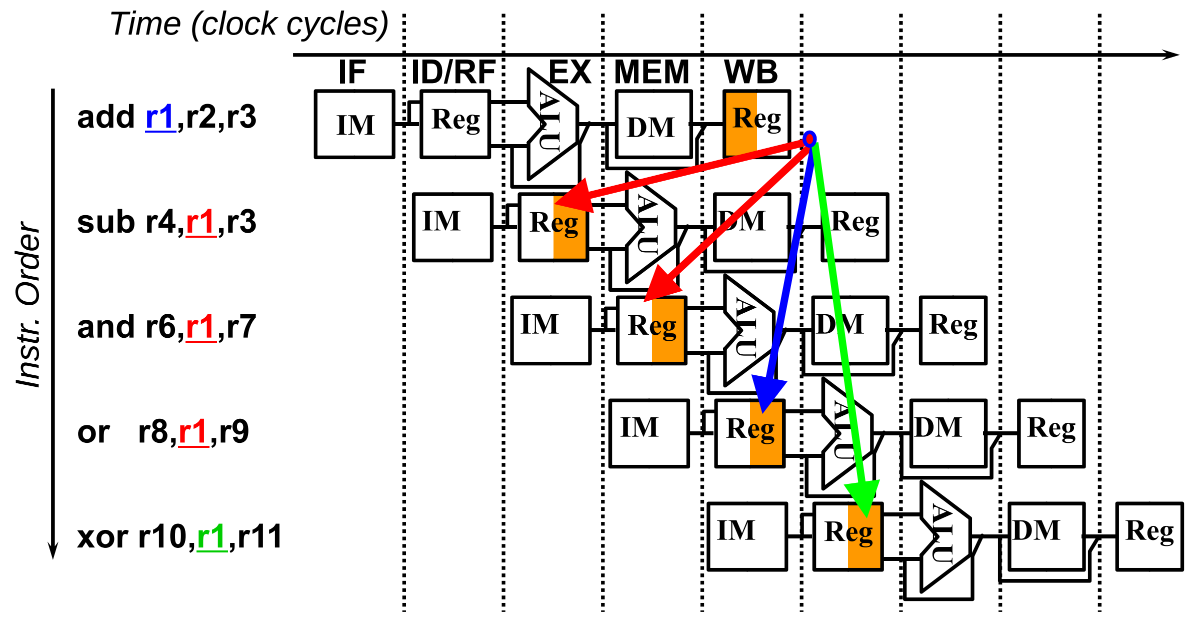
Chapter Four – II (2/5)

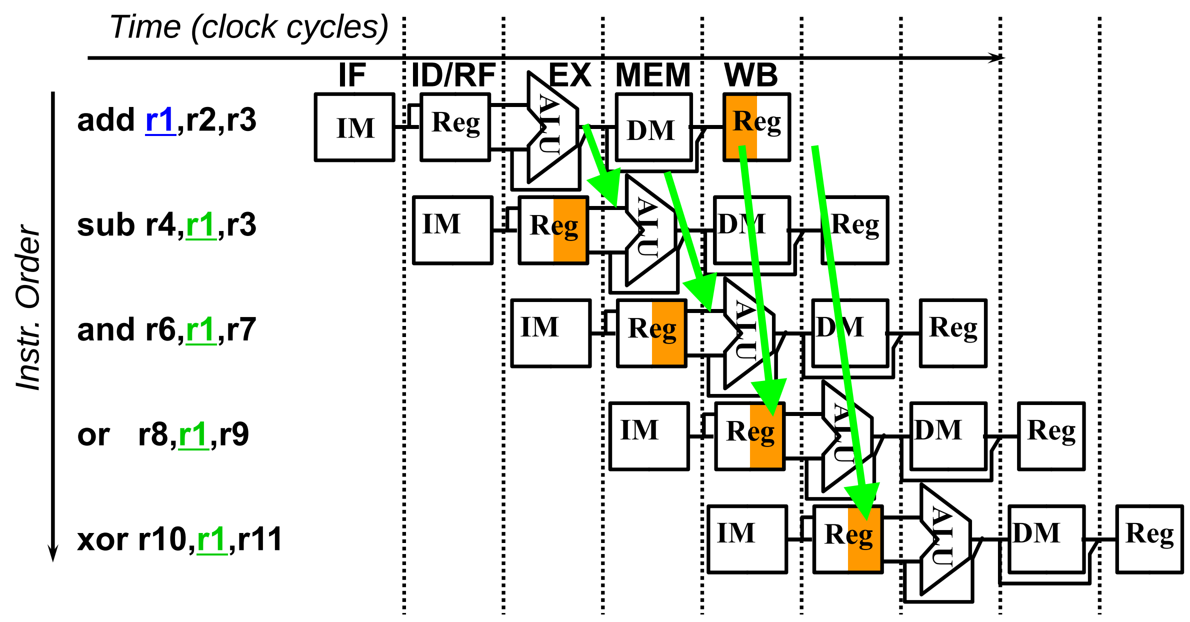
Data Hazard on r1:

- Dependencies backwards in time are hazards
  - "or" is OK if we define read/write properly



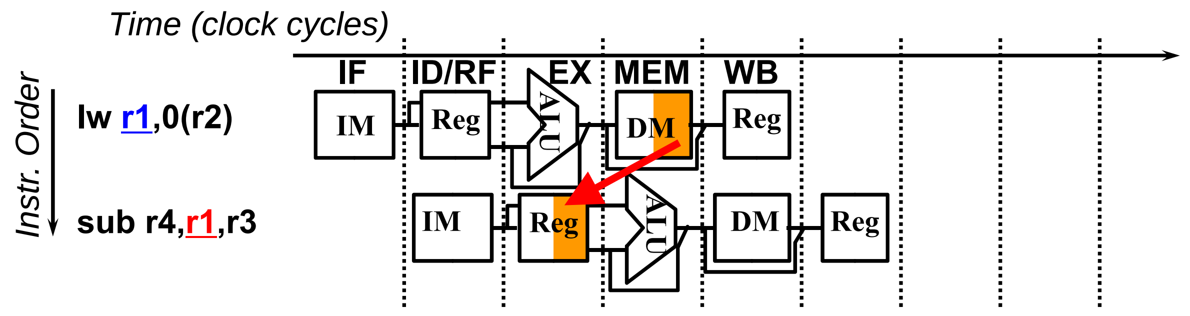
Data Hazard Solution:

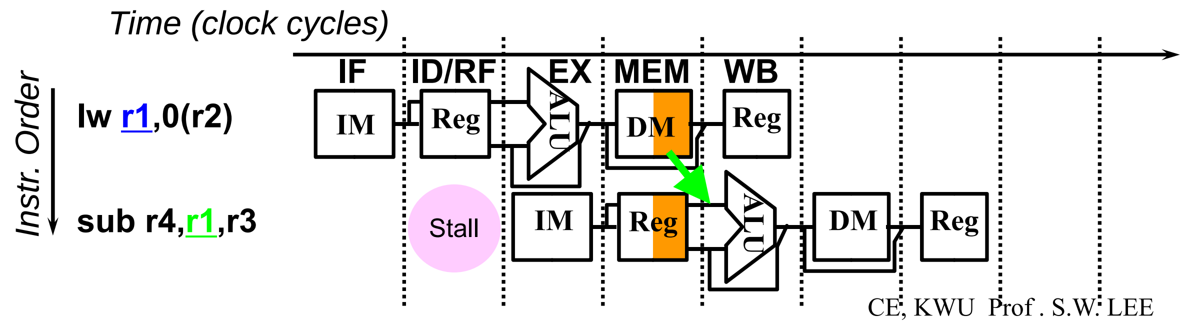
- "Forward" result from one stage to another



Forwarding (or Bypassing): What about Loads

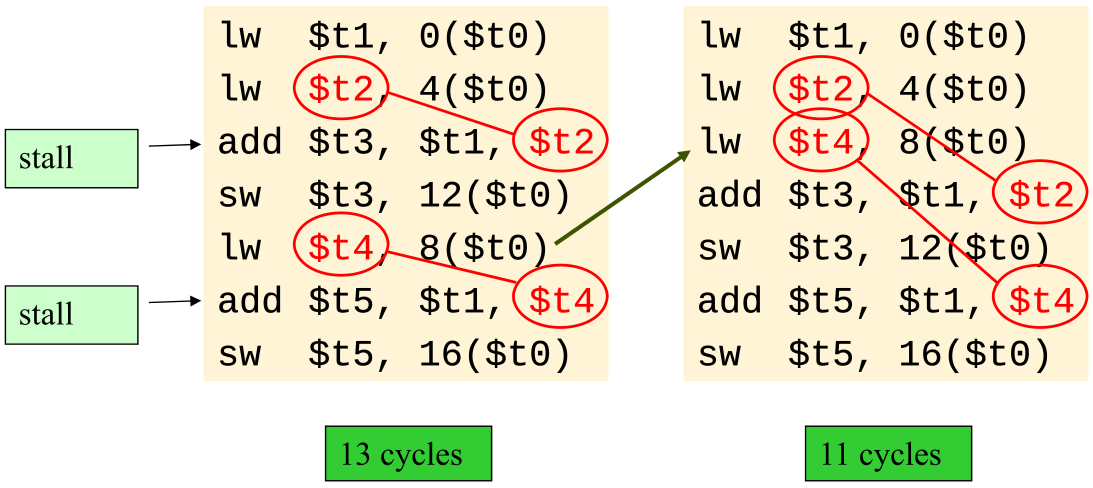
- Dependencies backwards in time are hazards
  - Can't solve with forwarding:
  - Must delay/stall instruction dependent on loads





Code Scheduling to Avoid Stalls

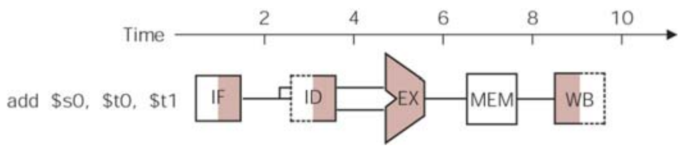
- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;



```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

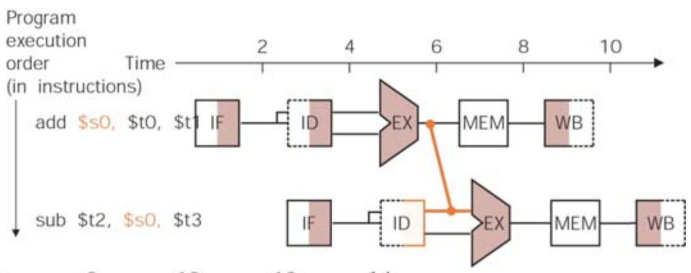
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

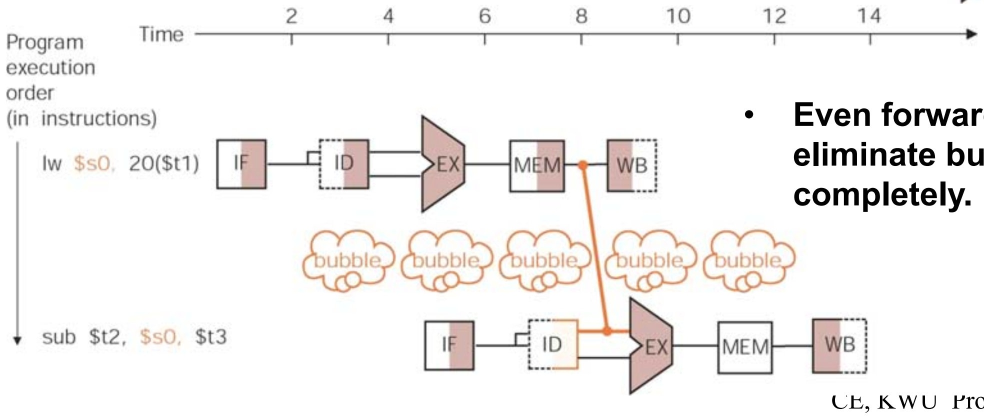
Data Hazards Summary



- Data dependency between two consecutive instructions

- Forwarding or bypassing



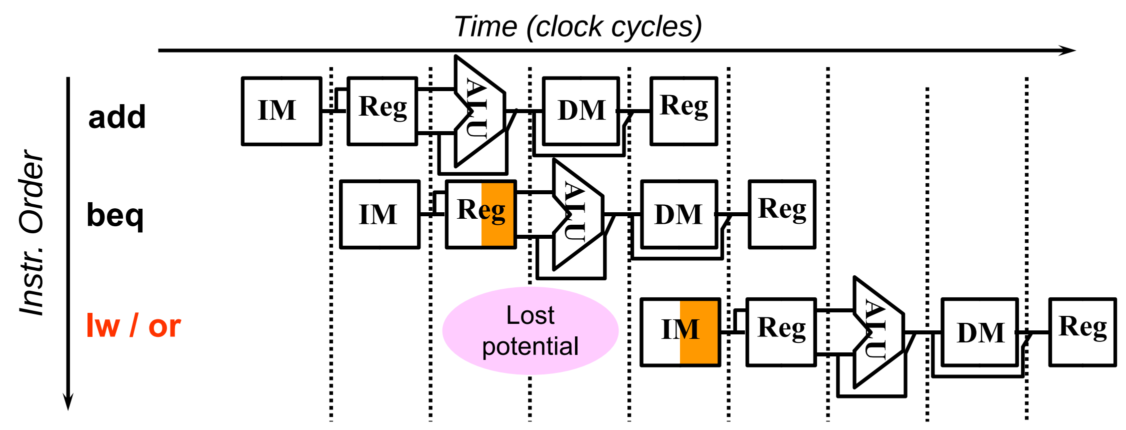


- Even forwarding do not eliminate bubbles completely.

Control Hazards

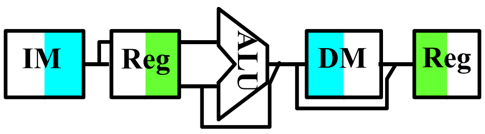
- **Sample code:**

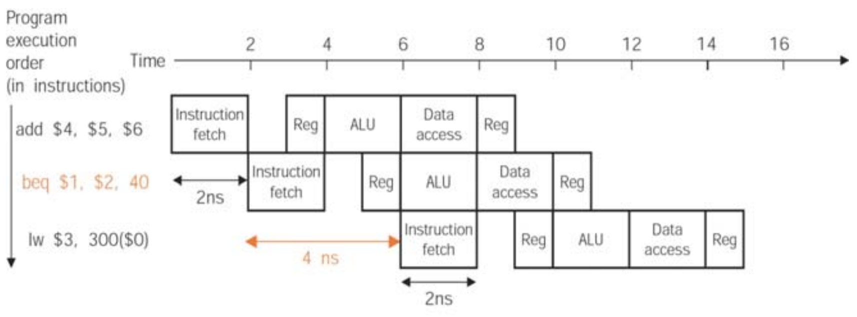
28:	add	$4, $5, $6
32:	beq	$1, $2, 40
36:	lw	$3, 300($0)
40:	or	$7, $8, $9



- **Solutions:**
  1. **Stall**
  2. **Prediction**
  3. **Delayed branch**

Control Hazards Solutions: Stall

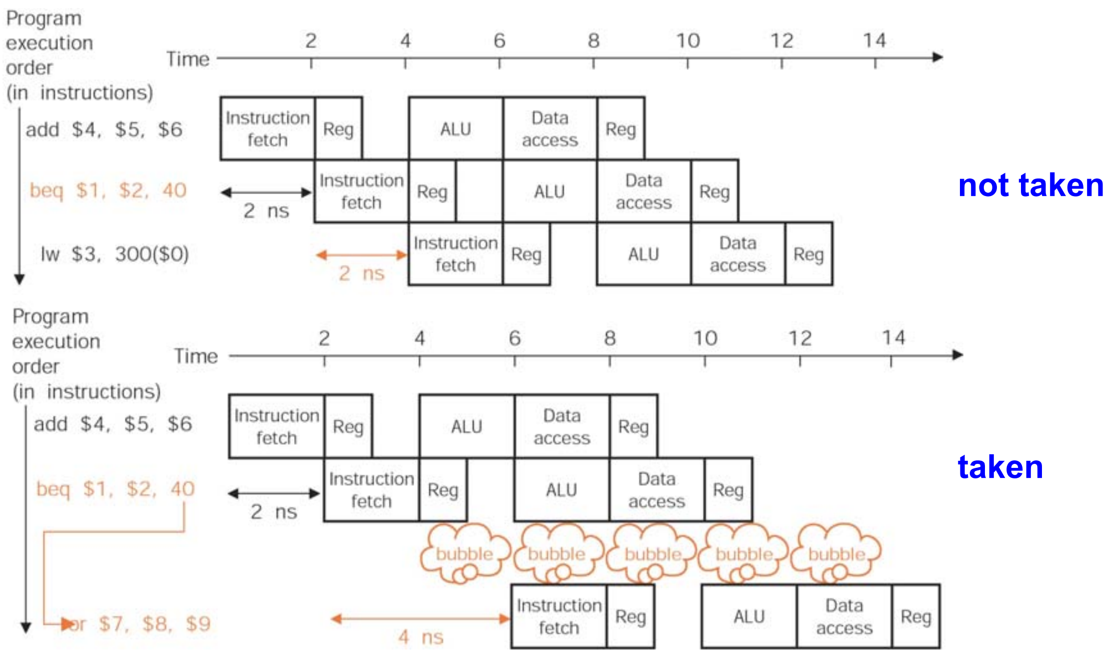




- **Stall: wait until decision is clear**
  - **Its possible to move up decision to 2nd stage (ID) by adding extra hardware to check registers as being read, to calculate the branch address, and to update the PC**
  - **Impact: 2 clock cycles per branch instruction ➔ slow**

Reduce Control Hazards by Prediction

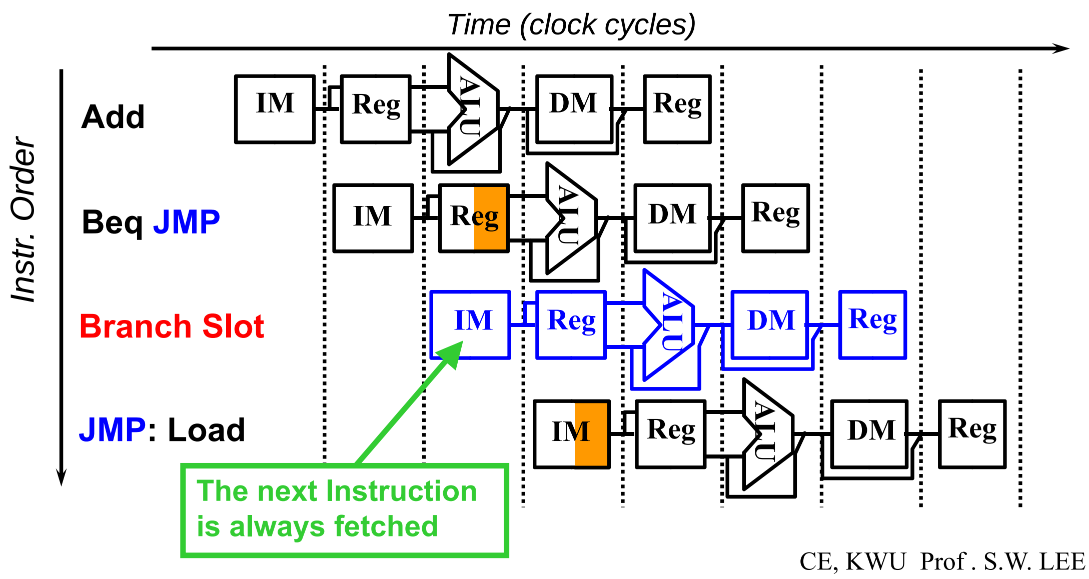
- **Predict: guess one direction then back up if wrong → predict not taken**



Impact: 1 clock cycle per branch inst. if right, 2 if wrong (right 50% of time) → More dynamic scheme: history of 1 branch (right 90%)

Delayed Branch for Solving Control Hazards

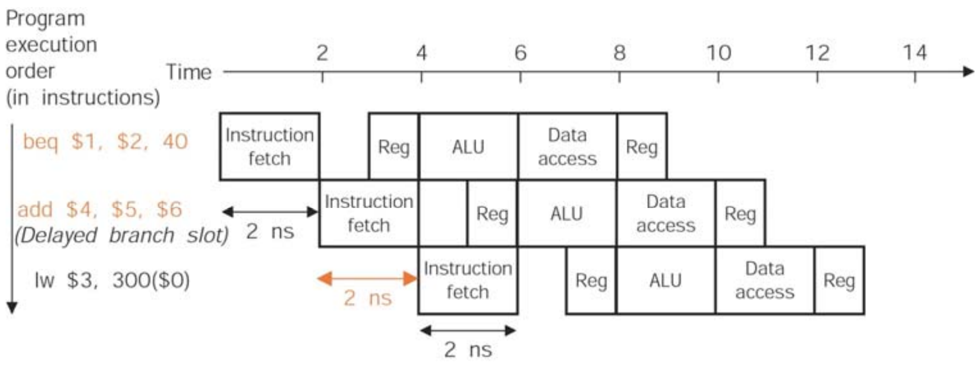
- **Redefine branch behavior "delayed branch"**
  - **branch takes place after the next instruction**
- **Impact: 1 clock cycles per branch instruction if we can find an instruction to put in "slot"**
- **As launch more instructions per clock cycle => less useful**



Delayed Branch for Solving Control Hazards

- **Delayed branch slot**
  - **Make the execution of the delayed branch slot harmless**

```
28:   add    $4, $5, $6           ➔     28:   beq    $1, $2, 40
32:   beq    $1, $2, 40                 32:   add    $4, $5, $6
36:   lw     $3, 300($0)                36:   lw     $3, 300($0)
40:   or     $7, $8, $9                 40:   or     $7, $8, $9
```



Pipeline Summary

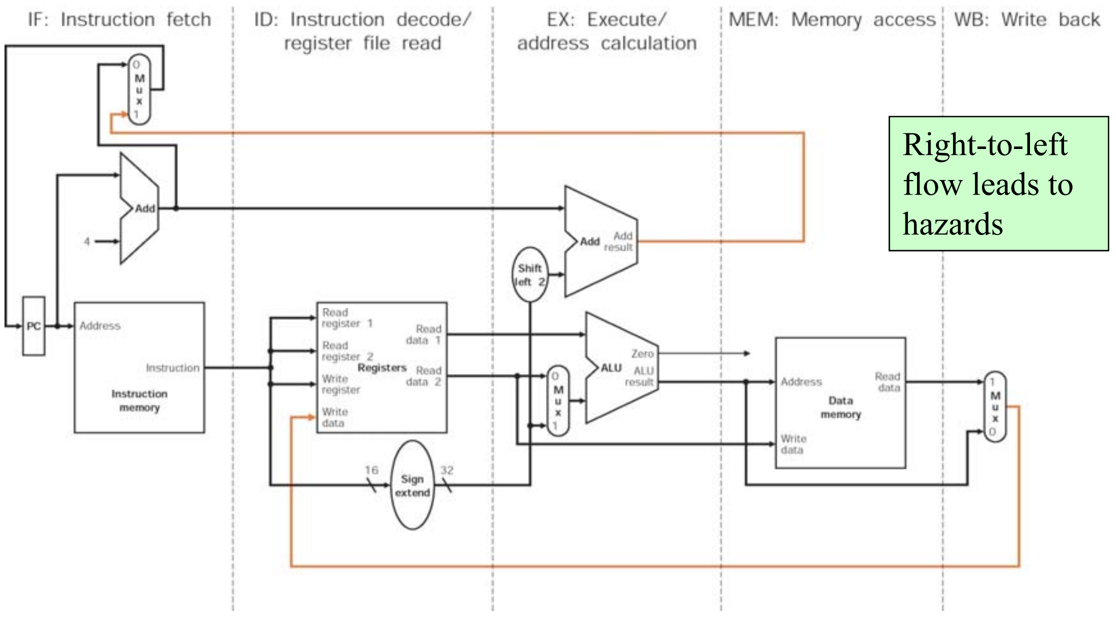
The BIG Picture

- **Pipelining improves performance by increasing instruction throughput**
  - **Executes multiple instructions in parallel**
  - **Each instruction has the same latency**
- **Subject to hazards**
  - **Structure, data, control**
- **Instruction set design affects complexity of pipeline implementation**

Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- Examine associated resources with states
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

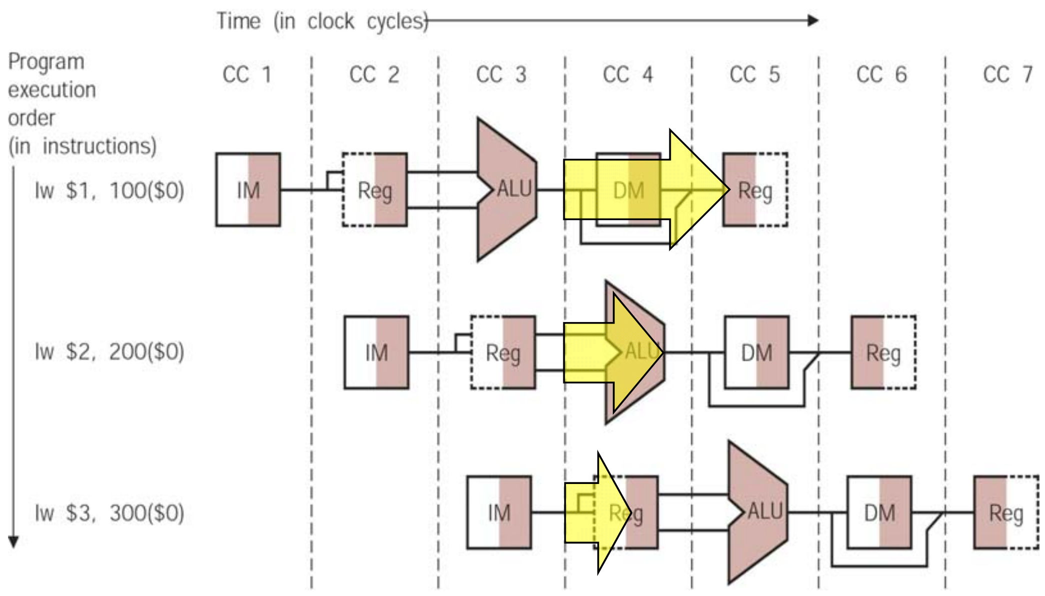
Single-Cycle Datapath



Right-to-left flow leads to hazards

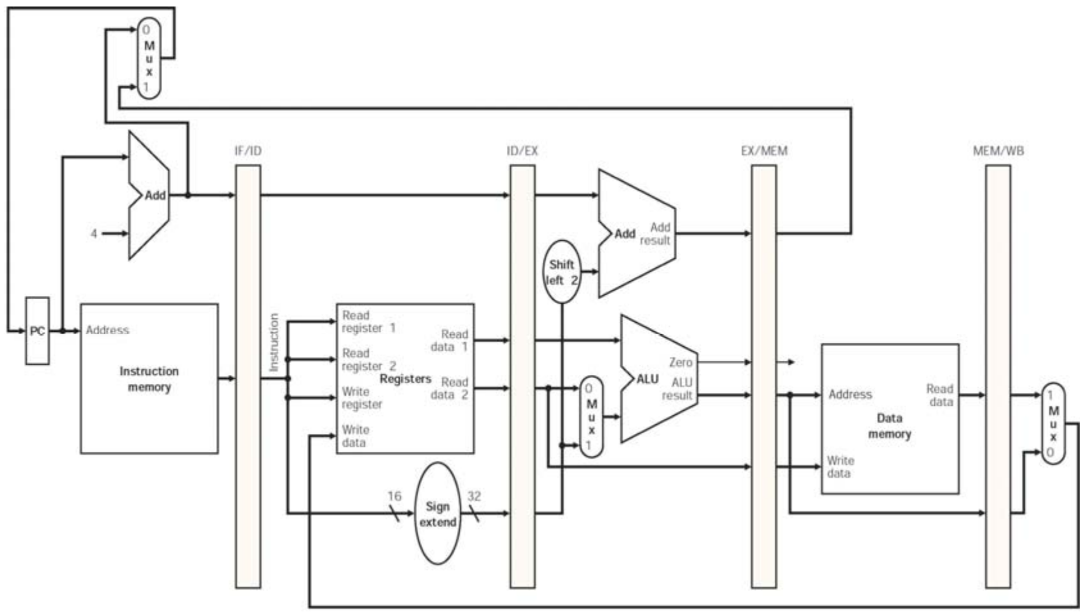
- Five stages: IF, ID, EX, MEM, WB
- Register Access
  - Write during the first half of the cycle
  - Read during the second half of the cycle

Instruction Execution on Single-Cycle Datapath



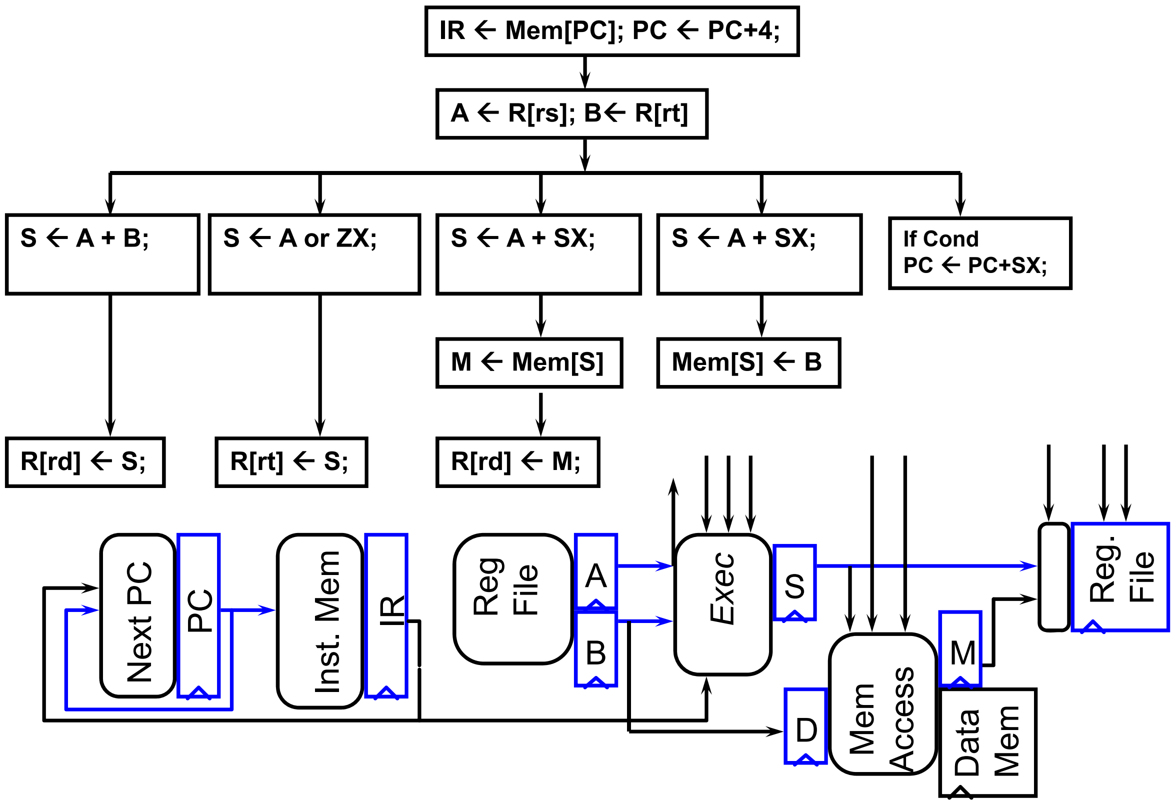
- ***What do we need to add to actually split the datapath into stages?***

Pipelined Datapath

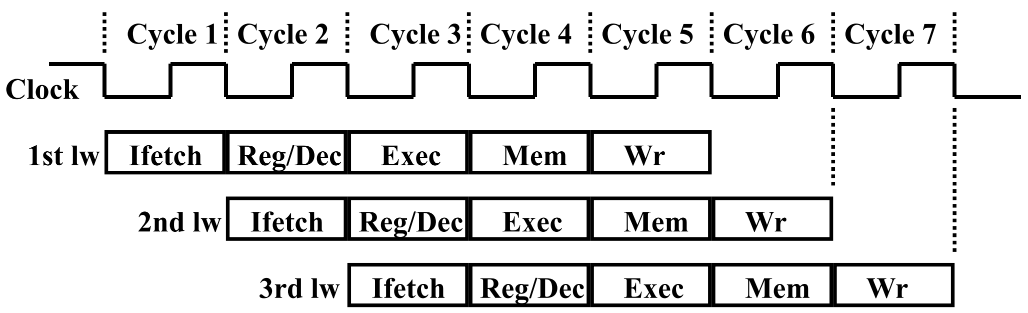


- **Registers separate stages in pipeline**
- **No pipeline register at the end of the WB stage**

Control and Datapath

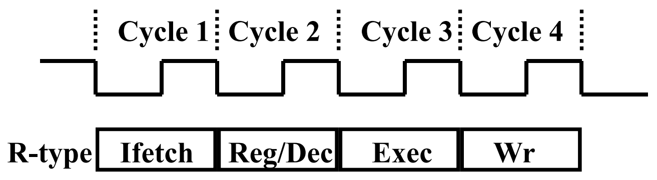


Pipelining the Load Instruction



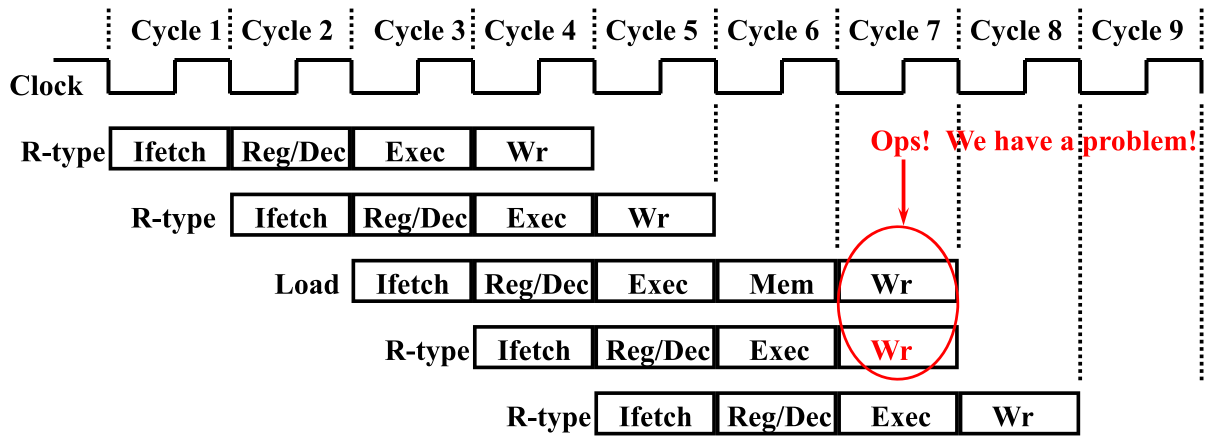
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
  - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
  - ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

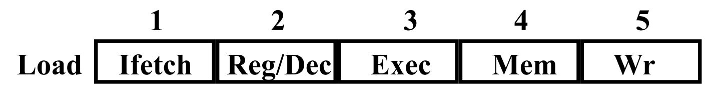
Pipelining the R-type and Load Instruction



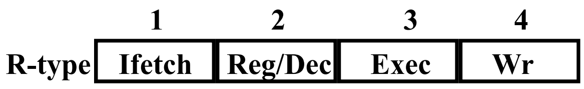
- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
- Load uses Register File's Write Port during its 5th stage

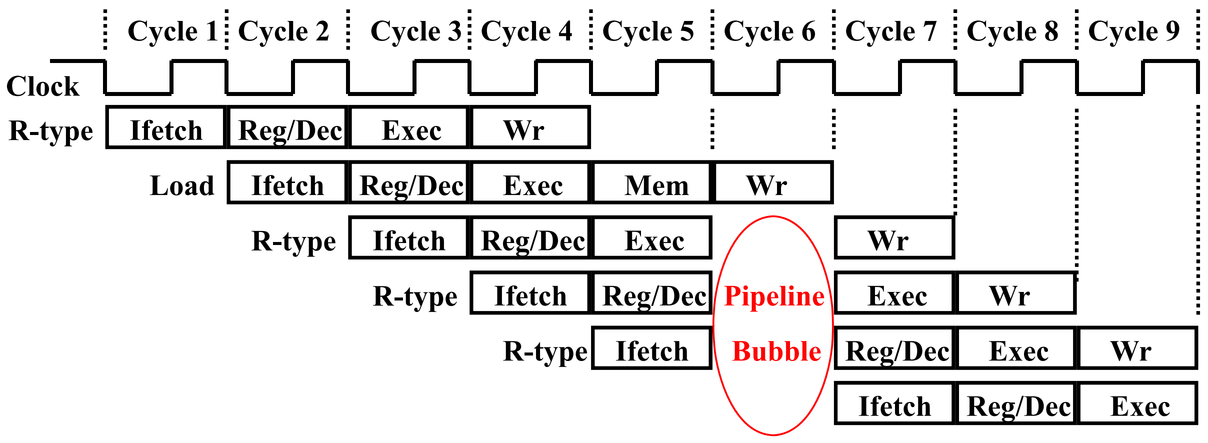


- R-type uses Register File's Write Port during its 4th stage



- 2 ways to solve this pipeline hazard.

Solution 1: Insert "Bubble" into the Pipeline

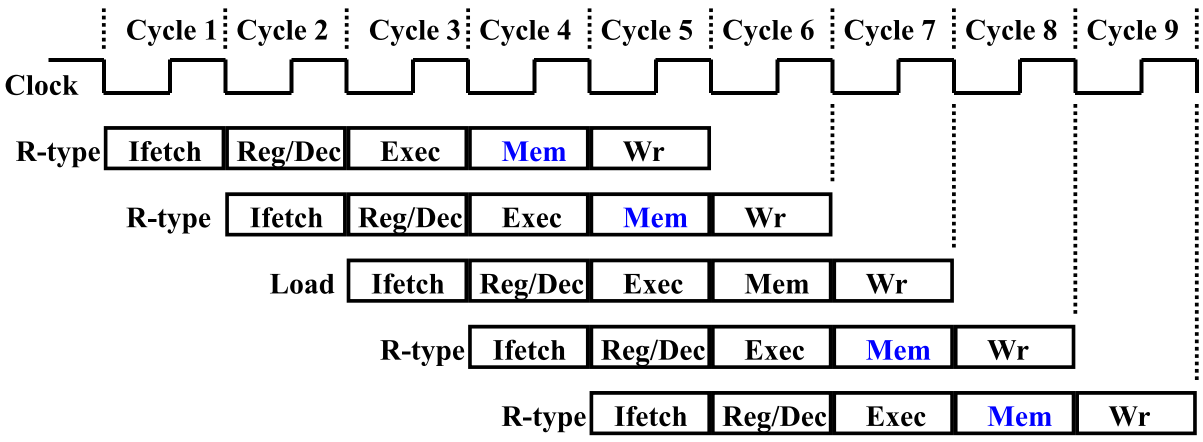


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

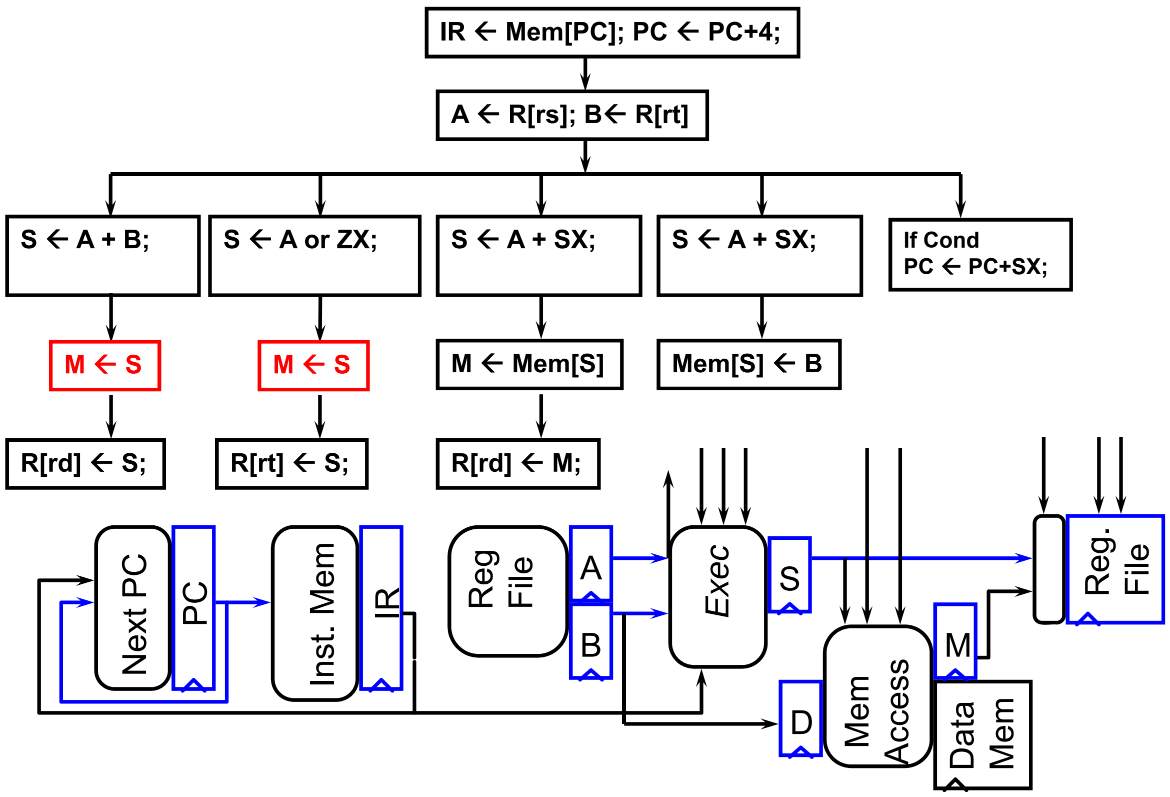
- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NO-OP stage: nothing is being done.

	1	2	3	4	5
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr

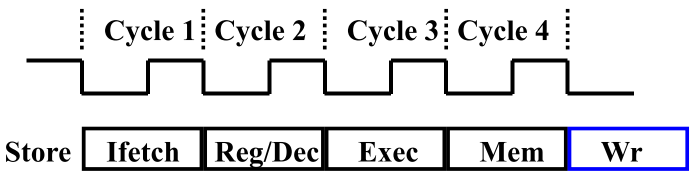


Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Clock
R-type Ifetch Reg/Dec Exec Mem Wr
R-type Ifetch Reg/Dec Exec Mem Wr
Load Ifetch Reg/Dec Exec Mem Wr
R-type Ifetch Reg/Dec Exec Mem Wr
R-type Ifetch Reg/Dec Exec Mem Wr

Modified Control & Datapath

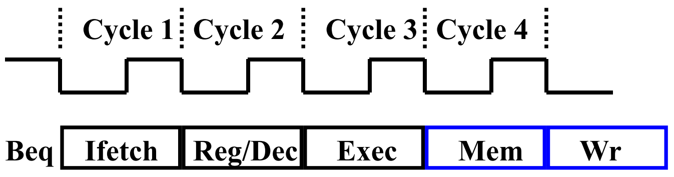


The Four Stages of Store



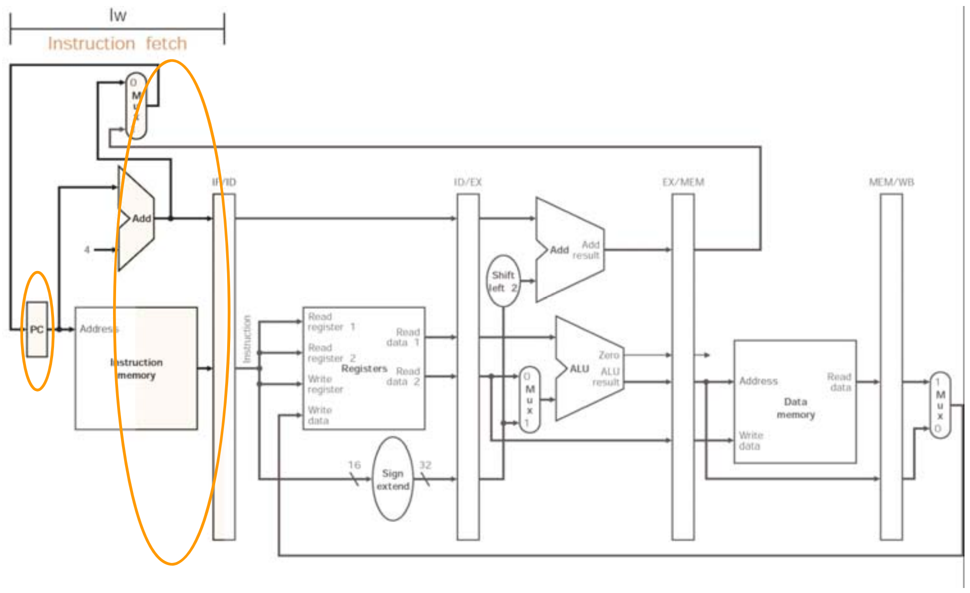
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
  - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Three Stages of Beq



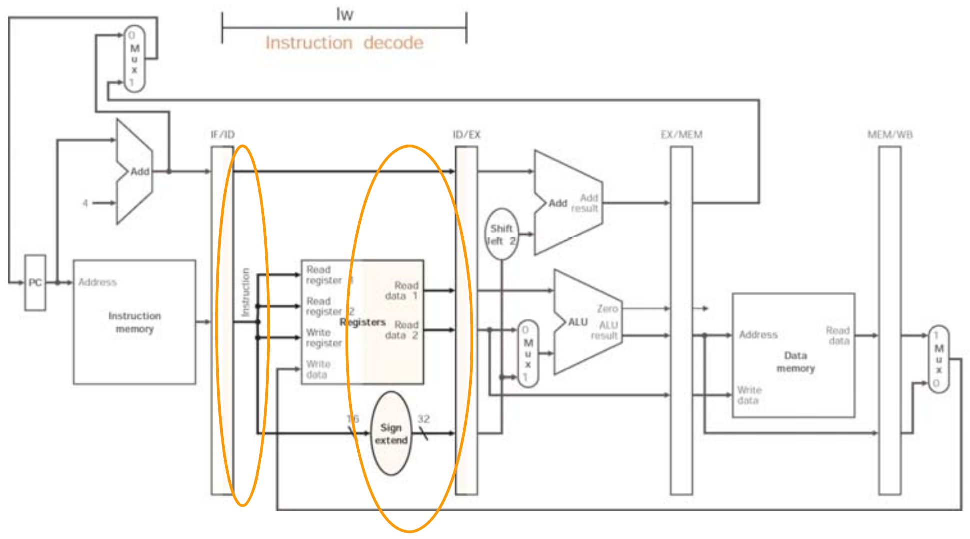
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

IF Stage of Pipeline



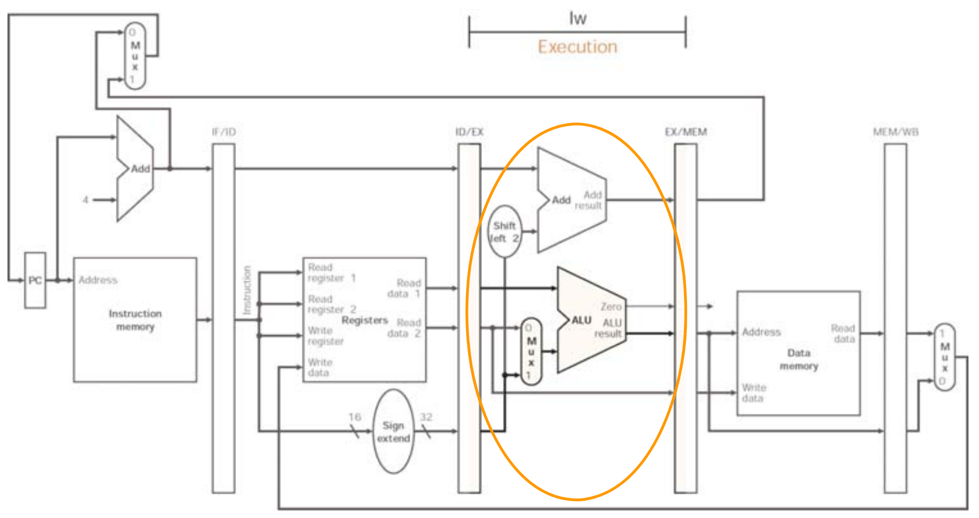
- IF/ID[31:0] = PC_{IF} ← PC + 4
- IF/ID[63:32] = IR ← IMem[PC]

ID stage of Pipeline



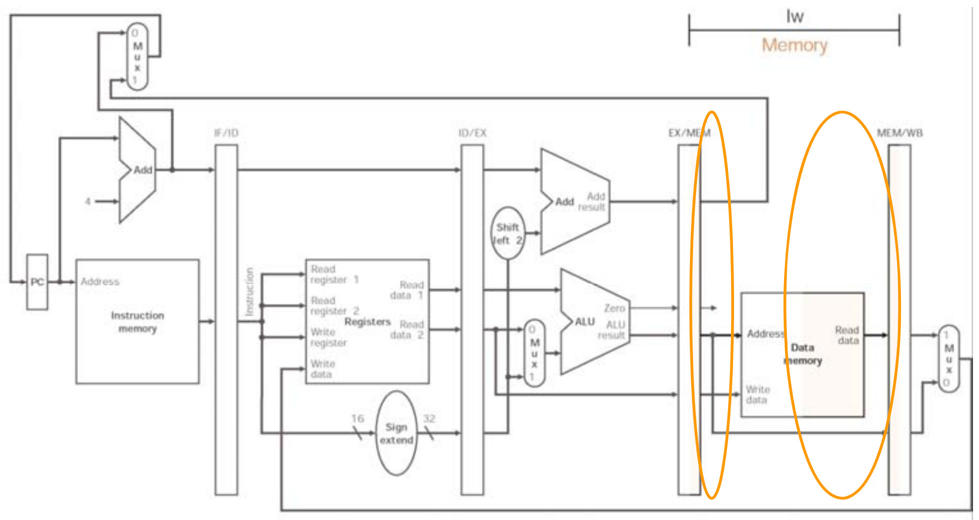
- ID/EX[31:0] = PC_{ID} ← IF/ID[31:0] = PC_{IF}
- ID/EX[63:32] = A_{ID} ← Reg[IR[25:21]]
- ID/EX[95:64] = B_{ID} ← Reg[IR[20:16]]
- ID/EX[127:96] = IMM ← SignExt(IR[15:0])

EXE stage of Pipeline for Load



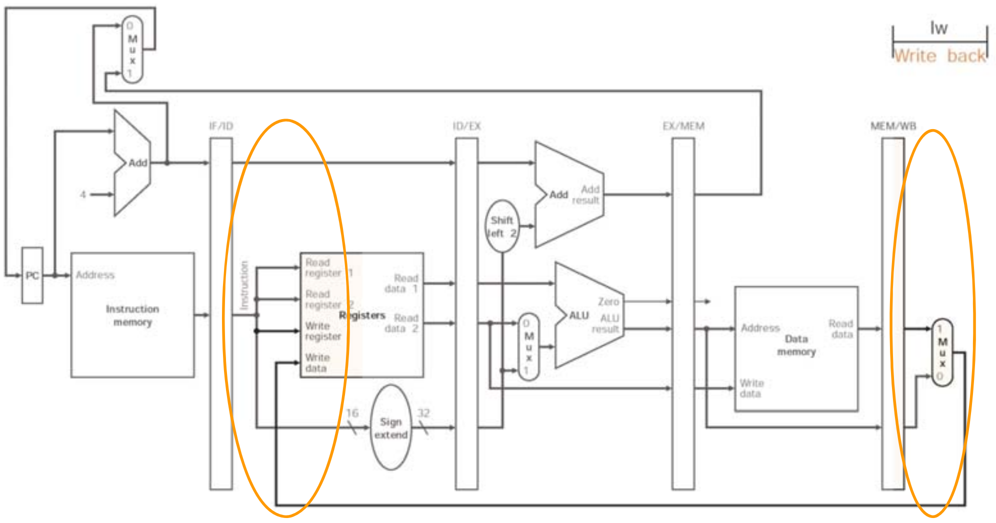
- $EX/MEM[31:0] = PC_{BR} \leftarrow PC_{ID} + IMM<<2$
- $EX/MEM[32] = Zero \leftarrow A_{ID} == B_{ID}$
- $EX/MEM[64:33] = ALUO_{EX} \leftarrow (A_{ID}\ op\ B_{ID})_{R\text{-}TYPE}$ or $(A_{ID}\ op\ IMM)_{I\text{-}TYPE}$
- $EX/MEM[96:64] = B_{EX} \leftarrow B_{ID}$

MEM stage of Pipeline for Load



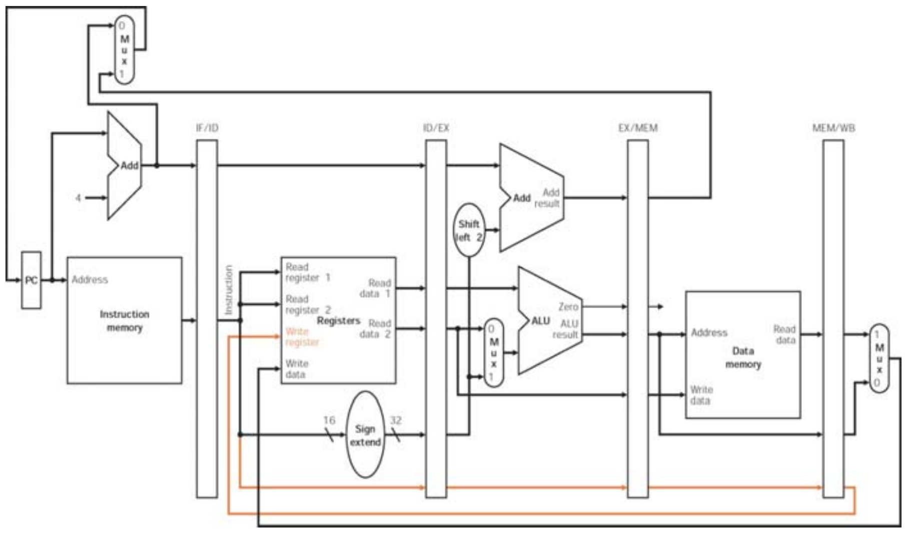
- $MEM/WB[31:0] = D_{MEM} \leftarrow DMem[ALUO_{EX}]$
- $MEM/WB[63:32] = ALUO_{MEM} \leftarrow ALUO_{EX}$

WB stage of Pipeline for Load



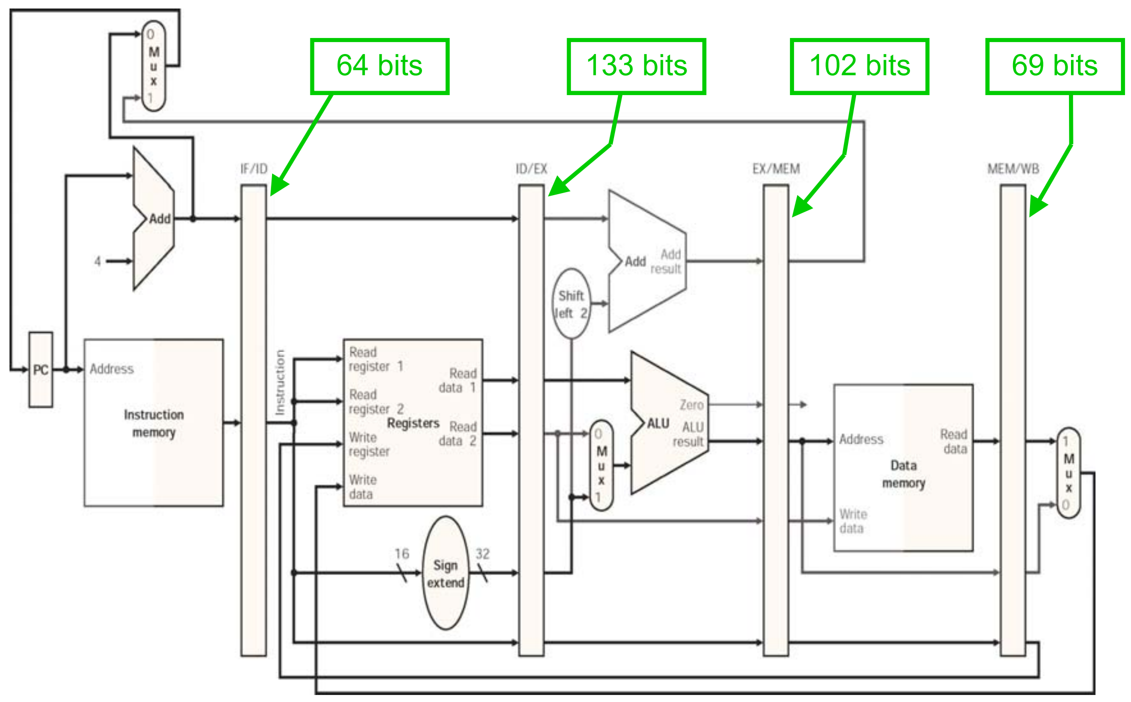
- Reg[?] = $(ALUO_{MEM})_{Arithmetic}$ or $(D_{MEM})_{Load}$

Corrected Pipelined Datapath

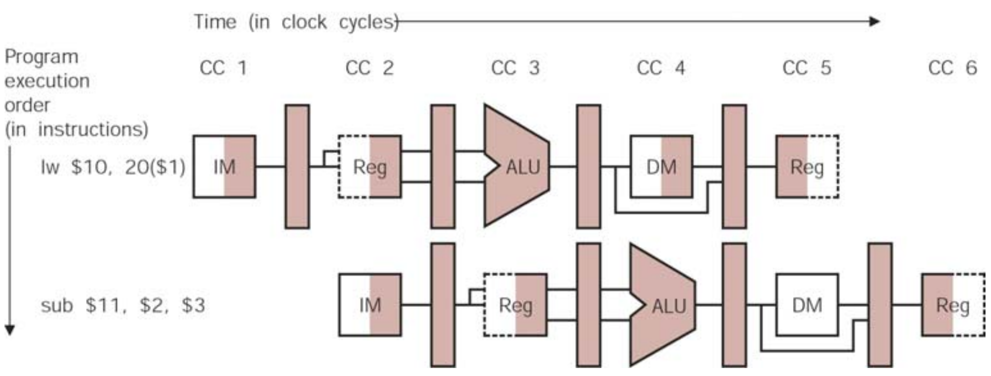


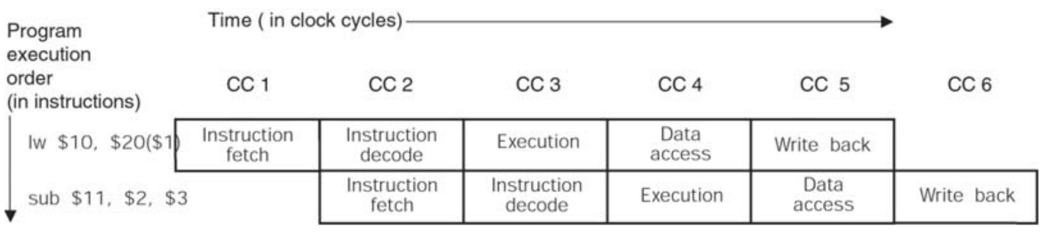
- The **write register number** at the WB stage should be passed from the ID stage until it reaches the MEM/WB pipeline register.
  - ID/EX[132:128] = RD_{ID} ← IR[19:15]
  - EX/MEM[101:97] = RD_{EX} ← RD_{ID}
  - MEM/WB[68:64] = RD_{MEM} ← RD_{EX}

Datapath used in all five stages of a load

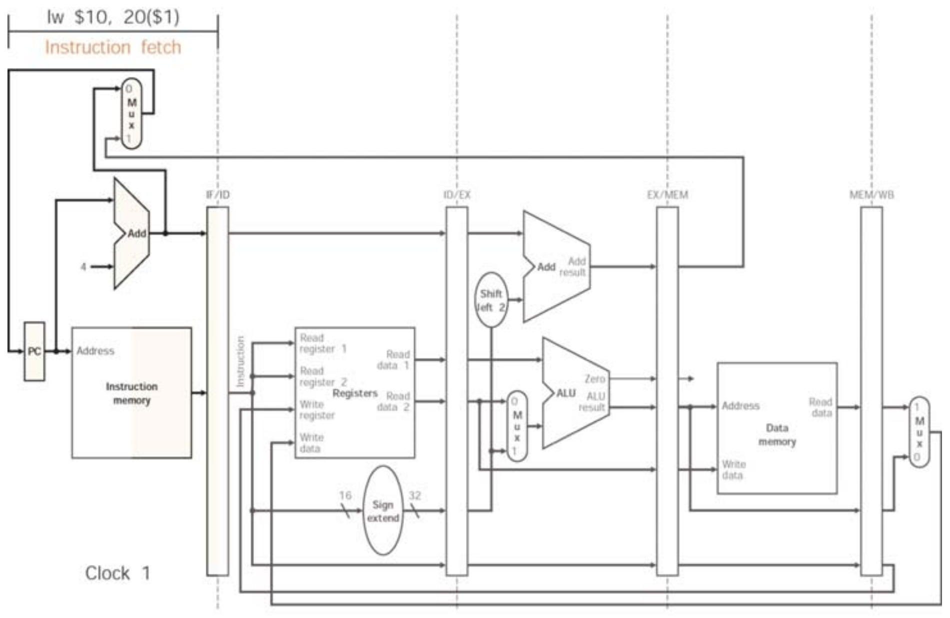


Graphically Representing Pipelines

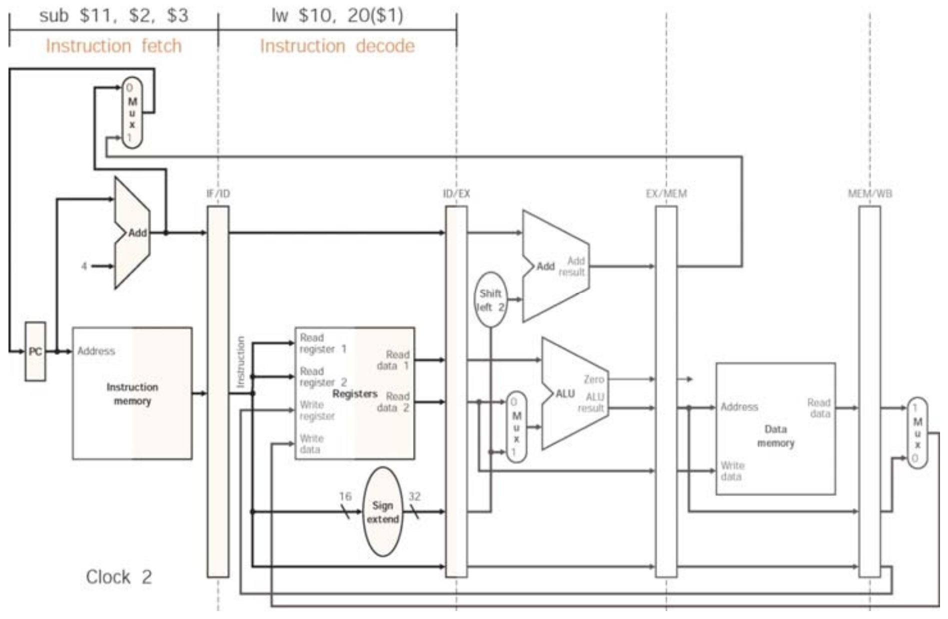




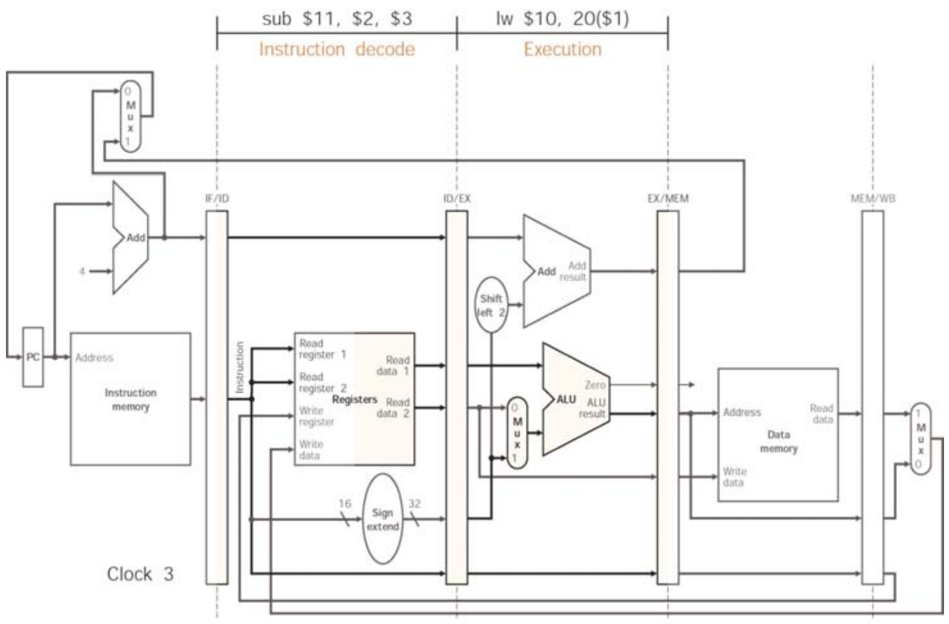
Execution of two instructions (Clock 1)



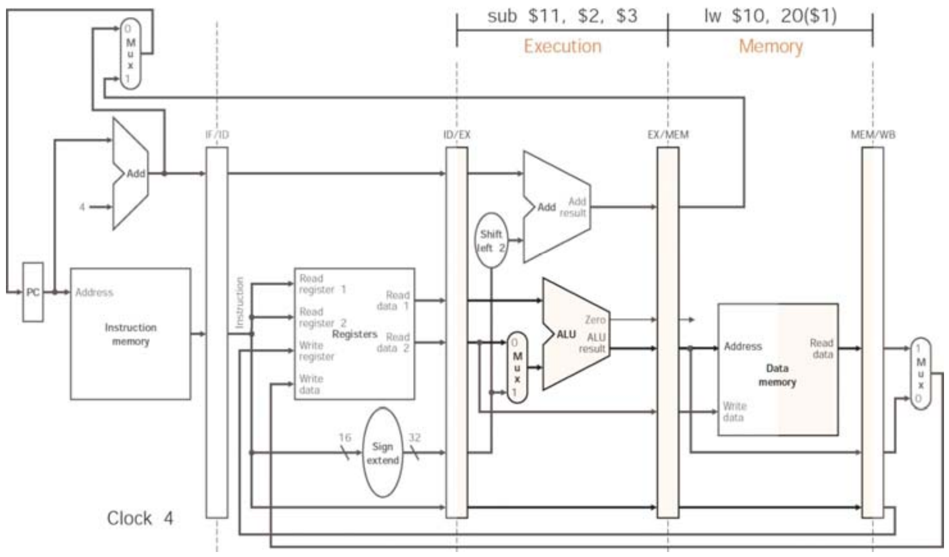
Execution of two instructions (Clock 2)



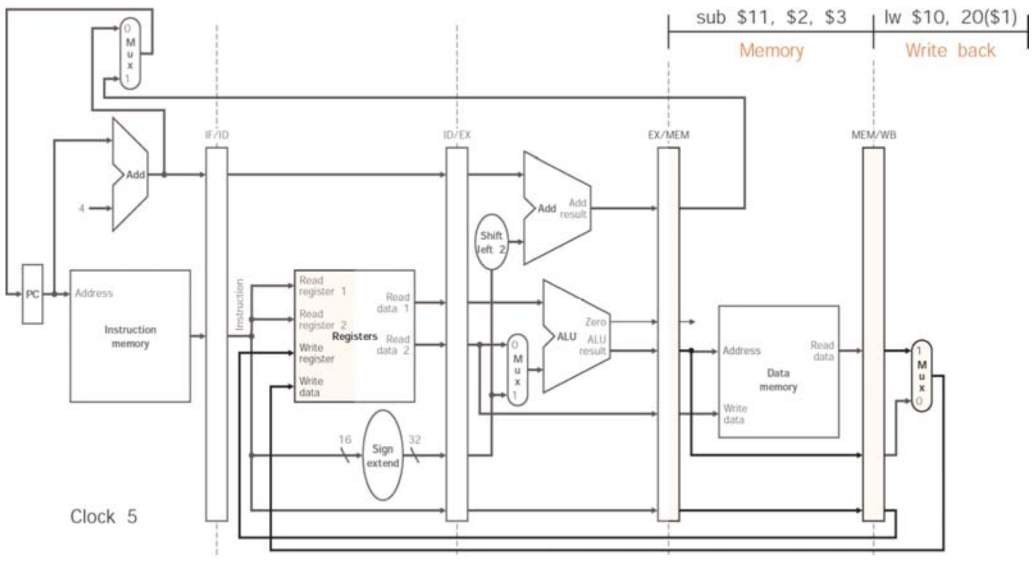
Execution of two instructions (Clock 3)



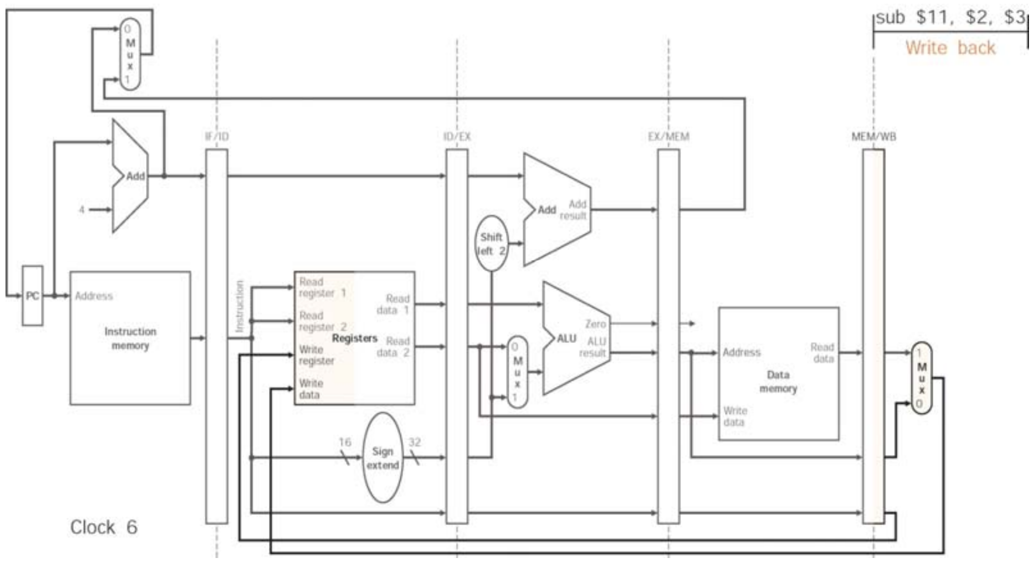
Execution of two instructions (Clock 4)



Execution of two instructions (Clock 5)



Execution of two instructions (Clock 6)


sub $11, $2, $3
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Mux
Address
Data memory
Write data
Read data
Mux
Sign extend
16
32
Clock 6