NATIONAL CHENGCHI UNIVERSITY

# Computer Architecture and Organization

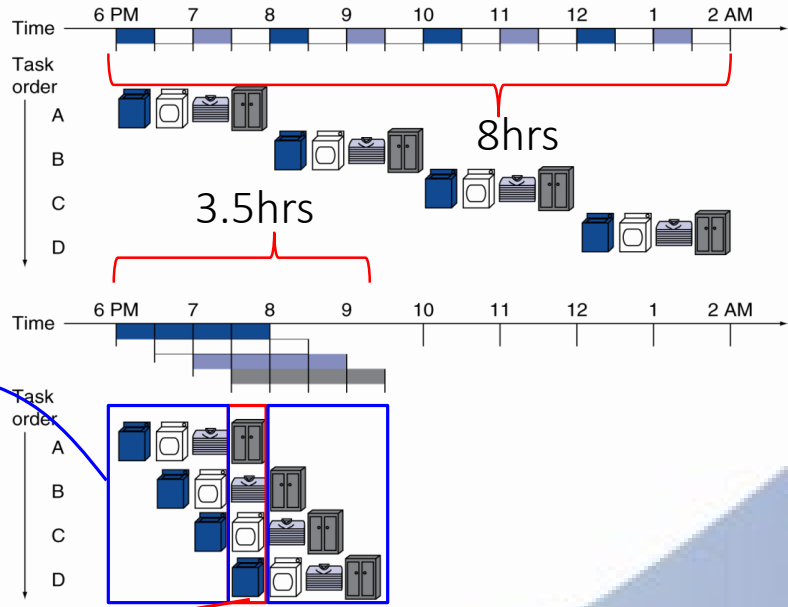
INSTRUCTOR: YAN-TSUNG PENG

DEPT. OF COMPUTER SCIENCE, NCCU

# CPU Design: Pipelining

# Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance (throughput)
  - Not shorten the time needed for completing one load
- Assume each stage takes the same amount of time
- Let each stage take 0.5hr, and 4 stages in total for a task
- If we do four loads separately
  - 4 loads *4 stages * 0.5hr = 8hr
- Using pipelining ($k \geq 3$ loads)
  - (4 loads-3)*0.5hr+(3 stages + 3 stages)*0.5=3.5hr
- Using pipelining for infinite loads:
  - (n loads-3)*0.5hr+(3 stages + 3 stages)*0.5=0.5n+1.5hr




Max speed-up: $\lim_{n\to\infty} \frac{2n}{0.5n+1.5} = 4$

# MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

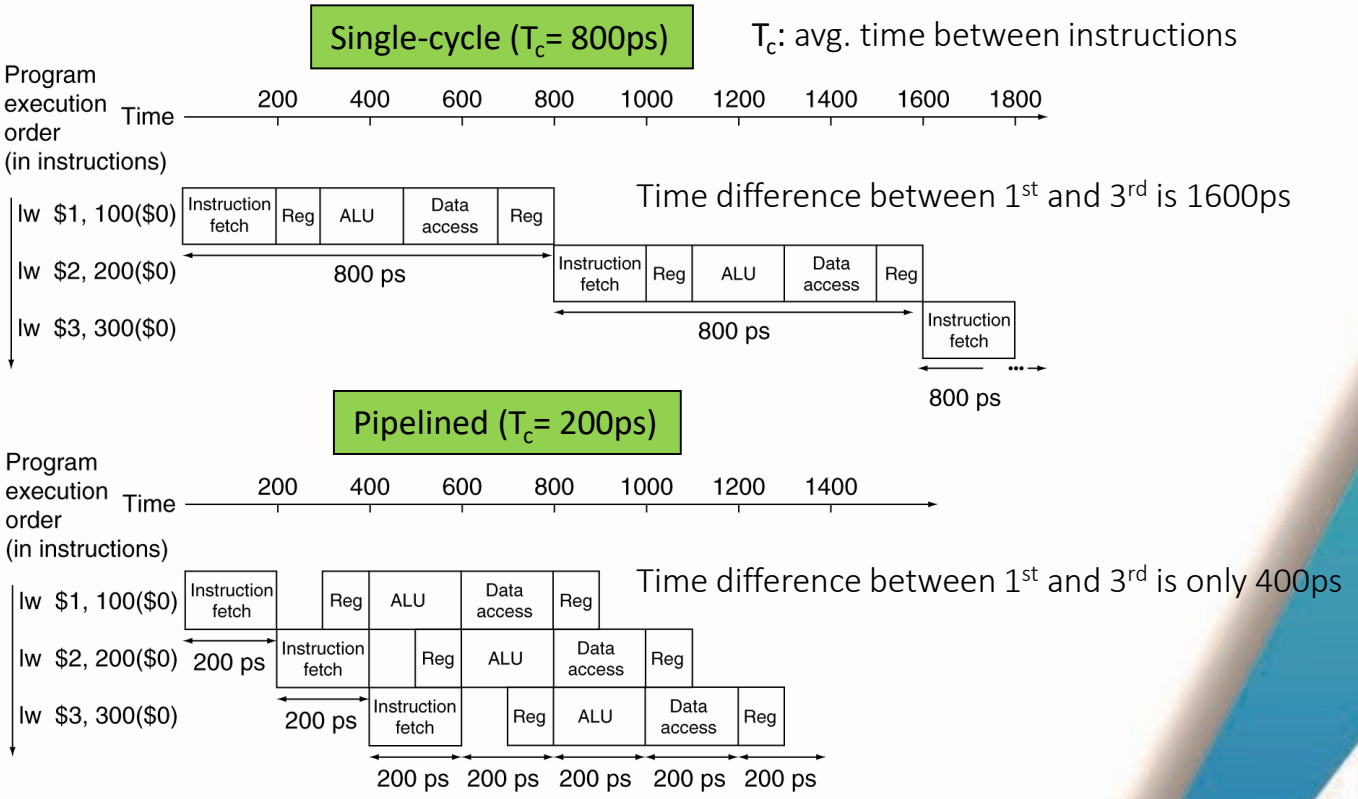
# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath (next page)

| Instr | IF | ID/REG | EX | MEM | WB | Total time |
|---|---|---|---|---|---|---|
| lw | 200ps | 100 ps | 200ps | 200ps | 100 ps | 800ps |
| sw | 200ps | 100 ps | 200ps | 200ps | | 700ps |
| R-format | 200ps | 100 ps | 200ps | | 100 ps | 600ps |
| beq | 200ps | 100 ps | 200ps | | | 500ps |

$1\ \text{ps} = 10^{-12}\ \text{s}$

# Pipeline Performance



# Pipeline Speedup

| Instr | IF | ID/REG | EX | MEM | WB | Total time |
|---|---|---|---|---|---|---|
| lw | 200ps | 100 ps | 200ps | 200ps | 100 ps | 800ps |
| sw | 200ps | 100 ps | 200ps | 200ps | | 700ps |
| R-format | 200ps | 100 ps | 200ps | | 100 ps | 600ps |
| beq | 200ps | 100 ps | 200ps | | | 500ps |

When pipelining, each stage takes 200ps, so in total it's 1000ps.

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$

$$\frac{1000\,ps}{5\,stages} = 200\,ps$$

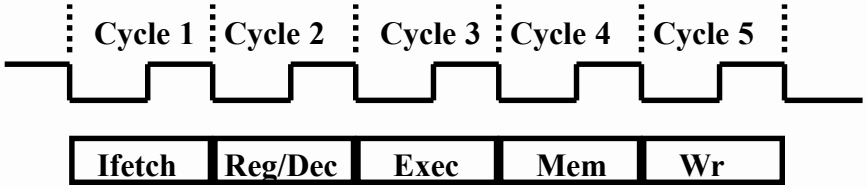
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step (in the ID stage)
  - Load and Store addressing (in two stages)
    - Calculate address in 3rd stage (EX)
    - Access memory in 4th stage (MEM)
  - Alignment of memory operands
    - Memory access takes only one cycle

The fact that the instruction formats of MIPS do not vary much and the length of the instructions is the same makes pipelining easy.

# Pipeline Stages




IF: Instruction Fetch

Fetch the instruction from the Instruction Memory

ID: Instruction Decode

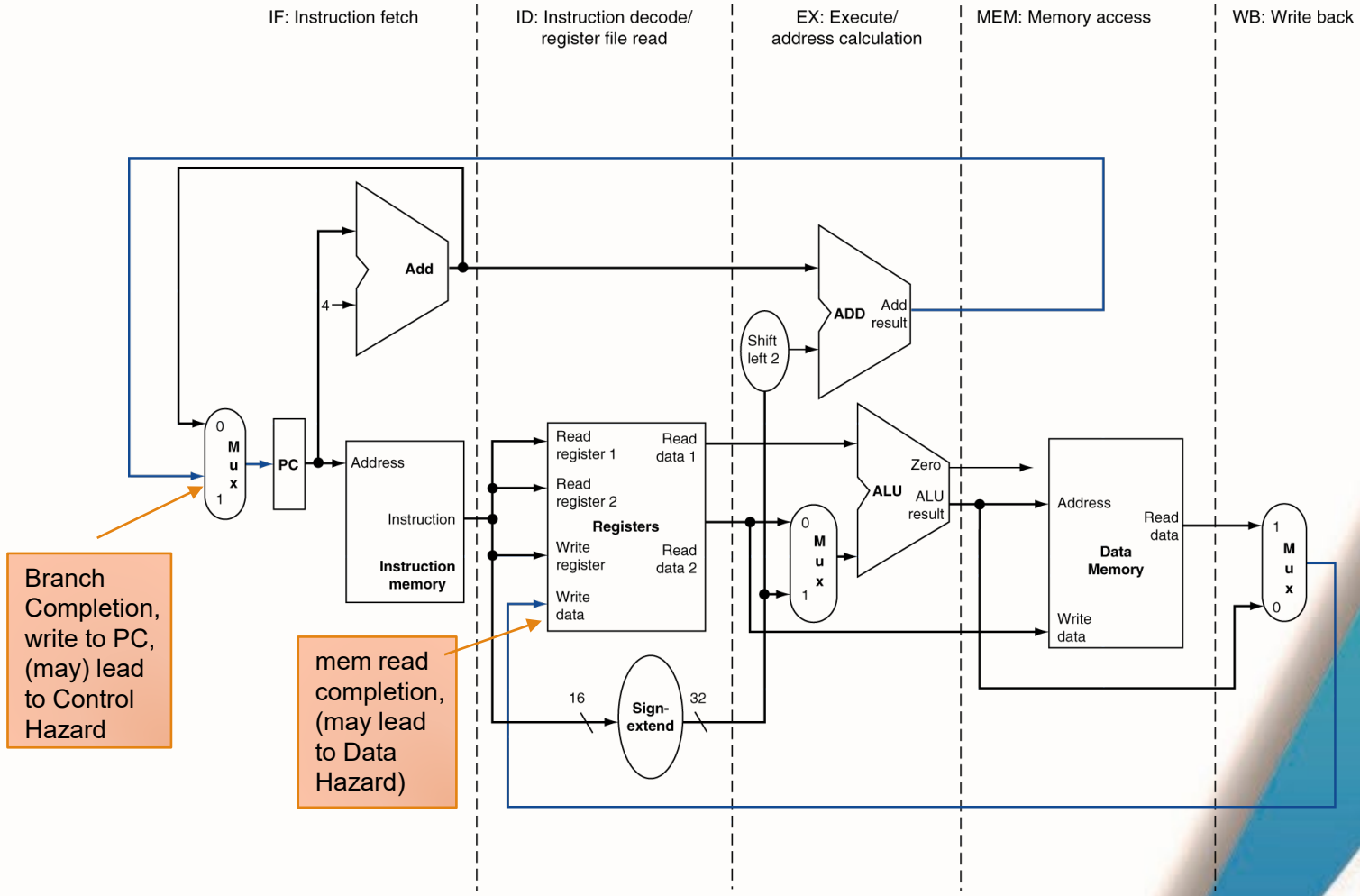
Registers fetch and instruction decode

EX: Calculate the memory address

MEM: Read the data from the Data Memory

WB: Write the data back to the register file

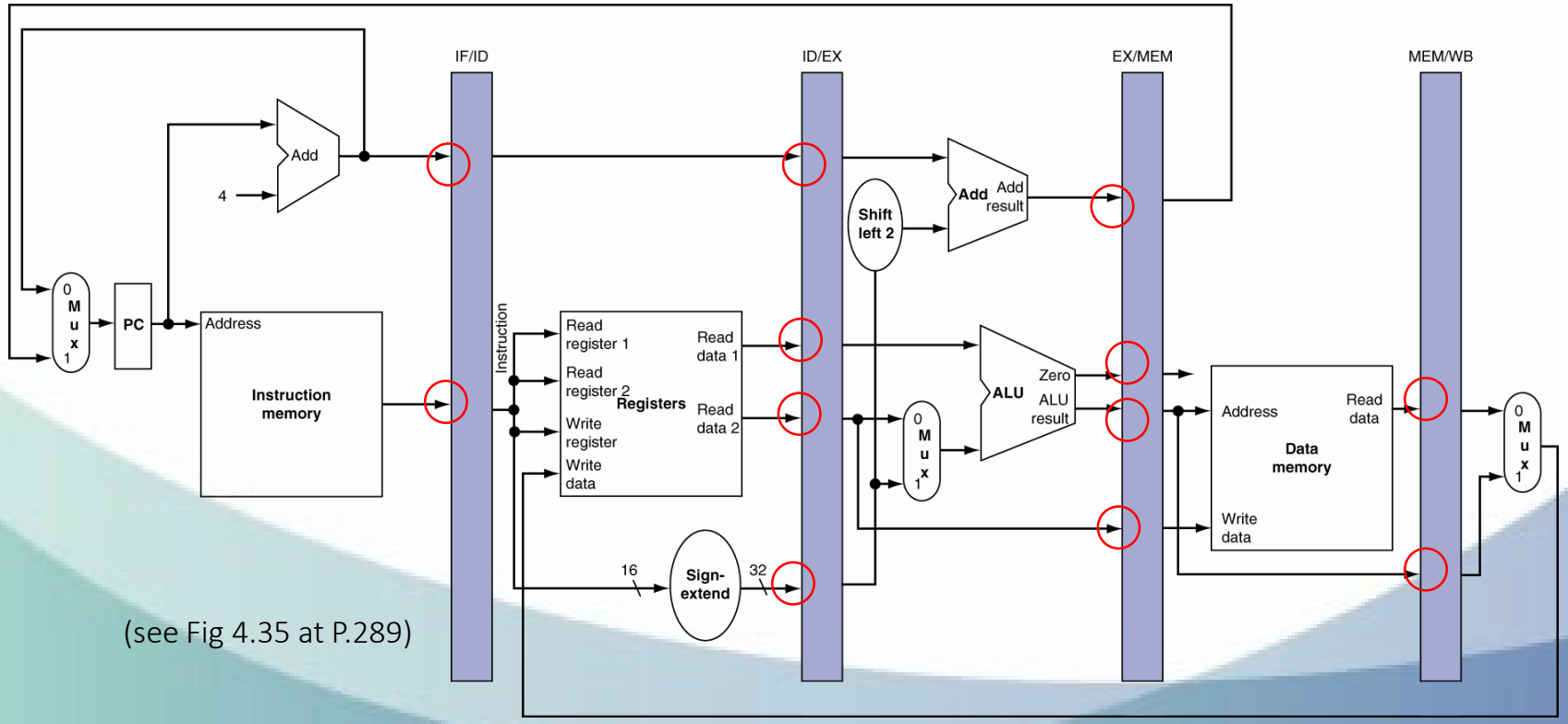
# MIPS Pipelined Datapath - What do we need to split an instruction into stages?



# Pipelined Version of Datapath

- Need registers between stages
  - To hold information produced in previous cycle (including data and control)

Pipelined registers:
IF/ID, ID/EX, EX/MEM, MEM/WB




(see Fig 4.35 at P.289)

# Hazards

- Situations that prevent starting the next instruction in the next cycle
- Three Hazards: Structure, data, and control hazards
- Structure hazards
  - A required resource is busy
  - Competing the same hardware resource in one cycle
- Data hazard
  - Need to wait for data outputted from previous instruction
- Control hazard
  - Taken branch decision from the previous instruction needs to void the current instruction

# Example - lw

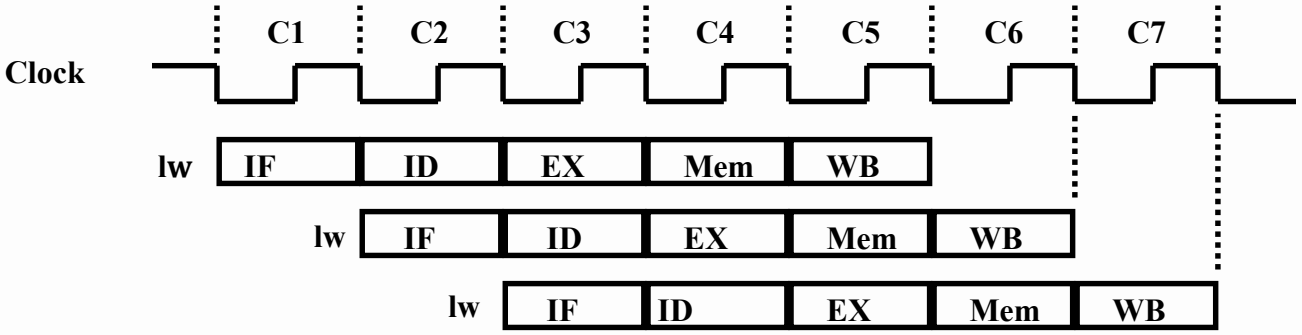
- lw $t0, 1200($t1)

R8=01000 R9=01001
rt rs

$$1200_{ten} = 0000010010110000_{two}$$

| 100011 | 01001 | 01000 | 0000 0100 0110 0000 |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |
| op | rs | rt | address |

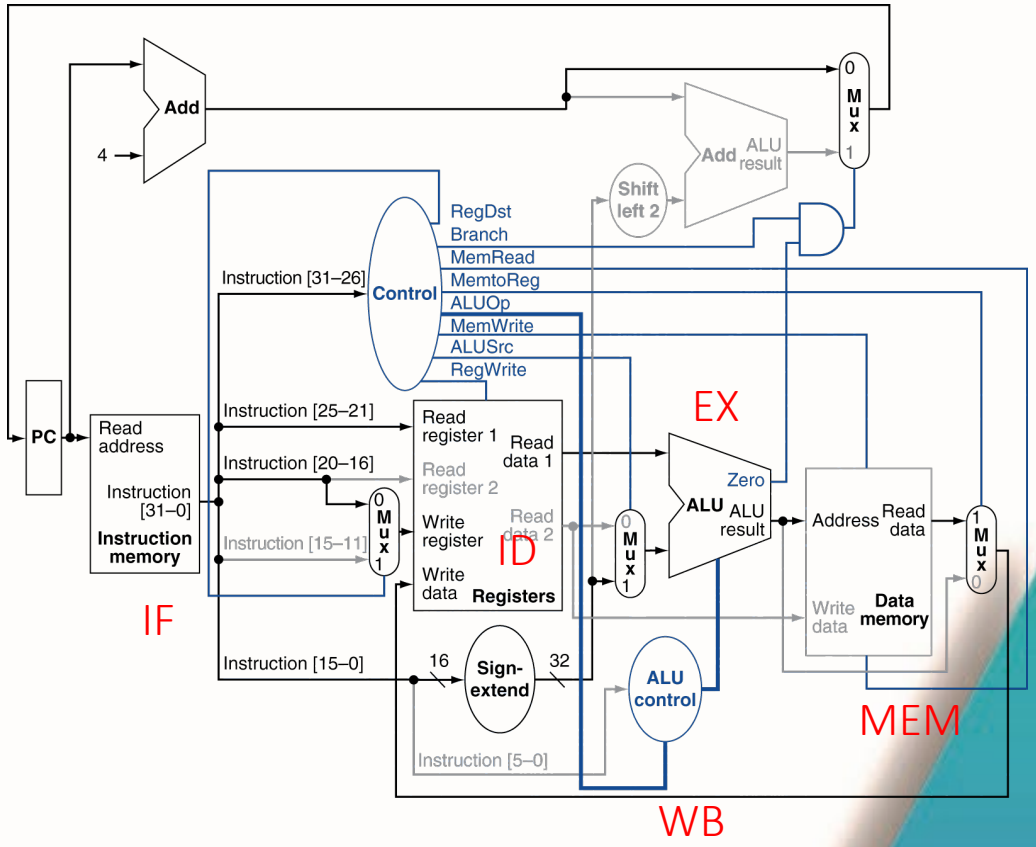
# Example - lw




There are 5 functional units in the pipeline datapath:

- IF stage: Instruction Memory
- ID stage: Register File's Read ports
- Ex stage: ALU execution
- MEM stage: Data Memory
- WB stage: Register File's Write port

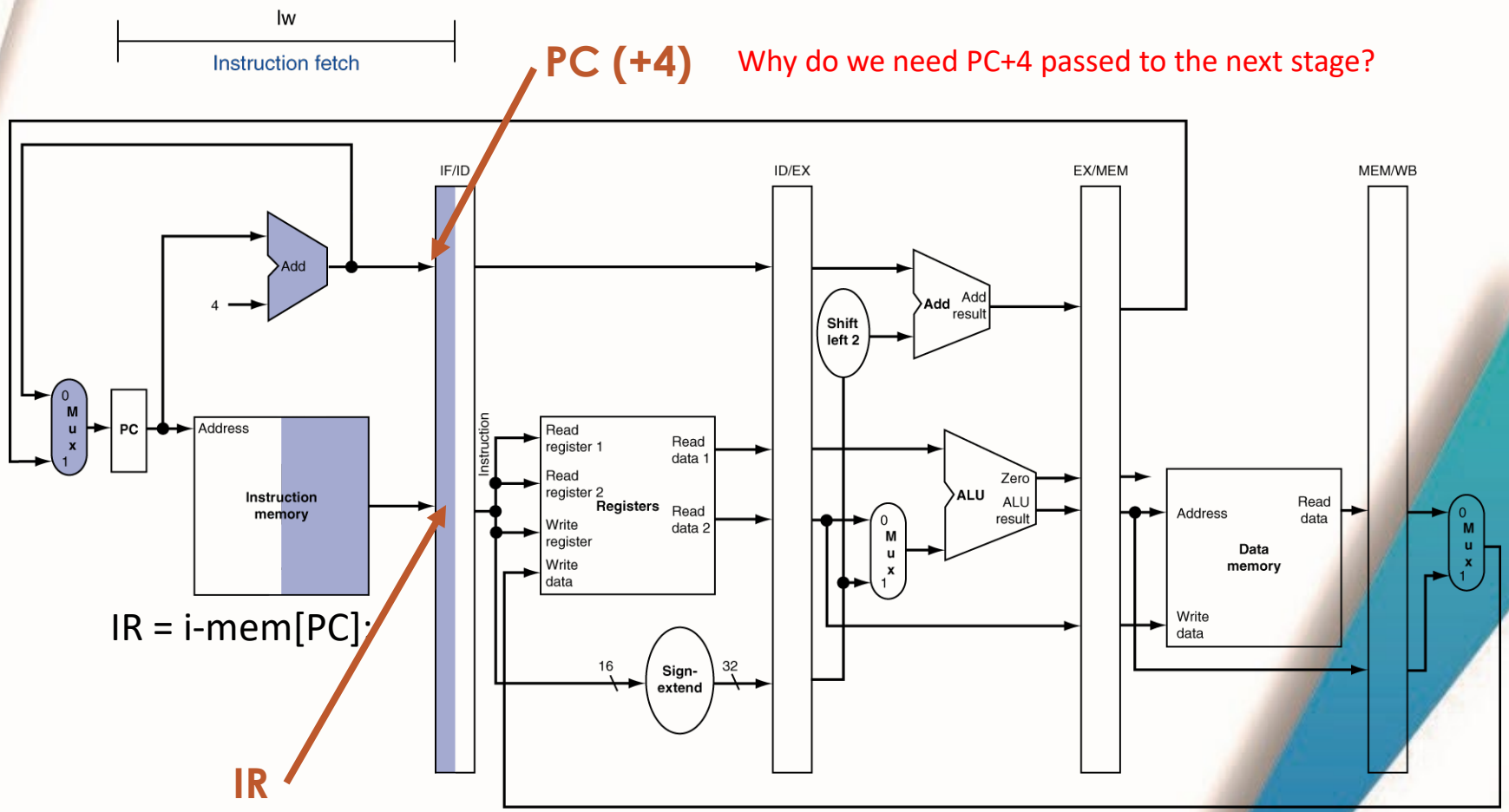
lw $t0, 1200($t1)




- IF: Fetch the instruction from the Instruction Memory
- ID: Registers fetch and instruction decode
- EX: Calculate the memory address
- MEM: Read the data from the Data Memory
- WB: Write the data back to the register file

# IF Stage - lw

*Fetch the instruction from the Instruction Memory*

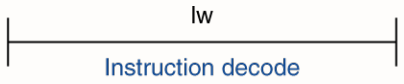



lw $t0, 1200($t1)

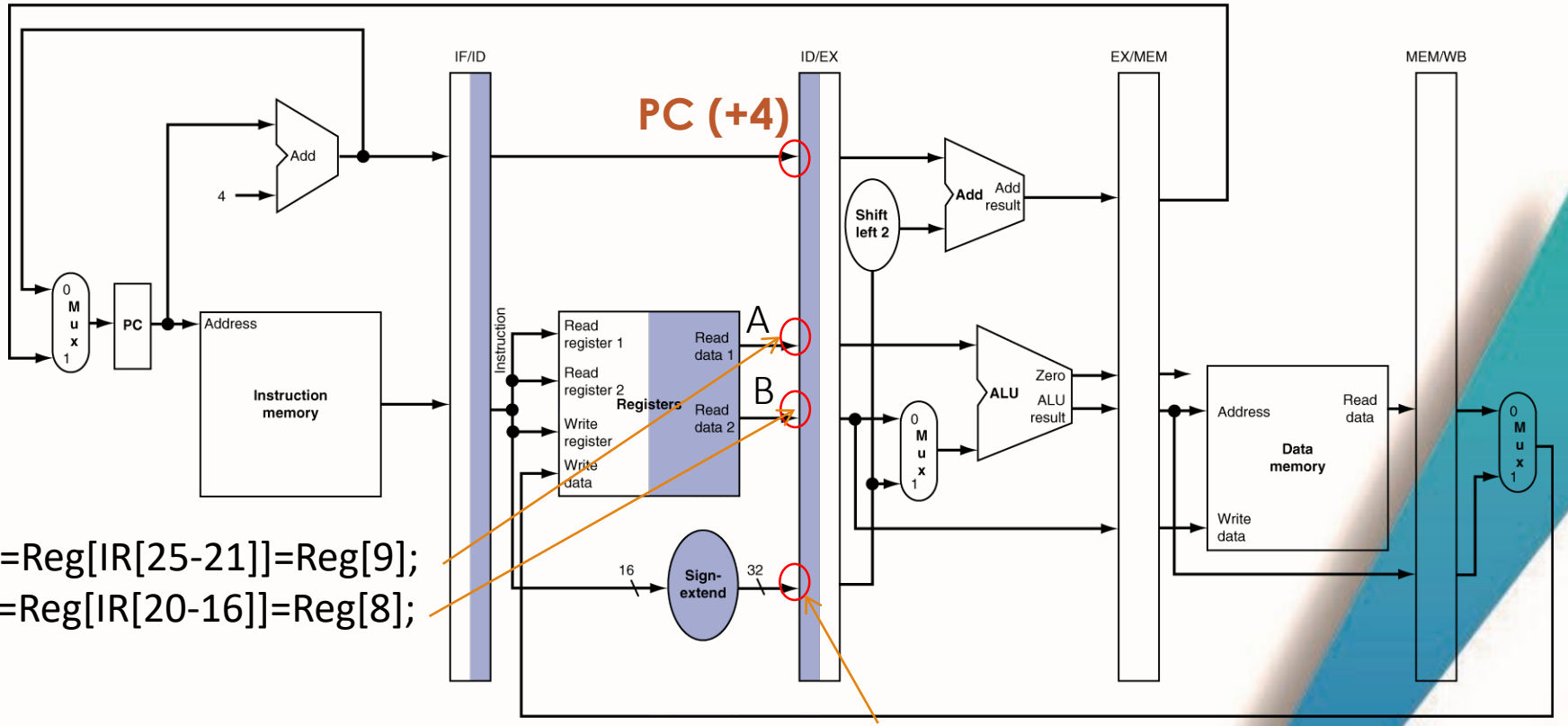
| 100011 | 01001 | 01000 | 0000 0100 0110 0000 |
|---|---|---|---|

# ID Stage - lw

*Fetch registers and decode the instruction*




lw $t0, 1200($t1)
rt rs




A = rs=Reg[IR[25-21]]=Reg[9];
B = rt=Reg[IR[20-16]]=Reg[8];

0000 0000 0000 0000 0000 0100 0110 0000

sign-extension

| 100011 | 01001 | 01000 | 0000 0100 0110 0000 |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |
| op | rs | rt | address |

# EX Stage - lw

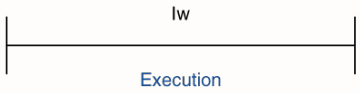
*Compute the memory address ($t1+1200)*

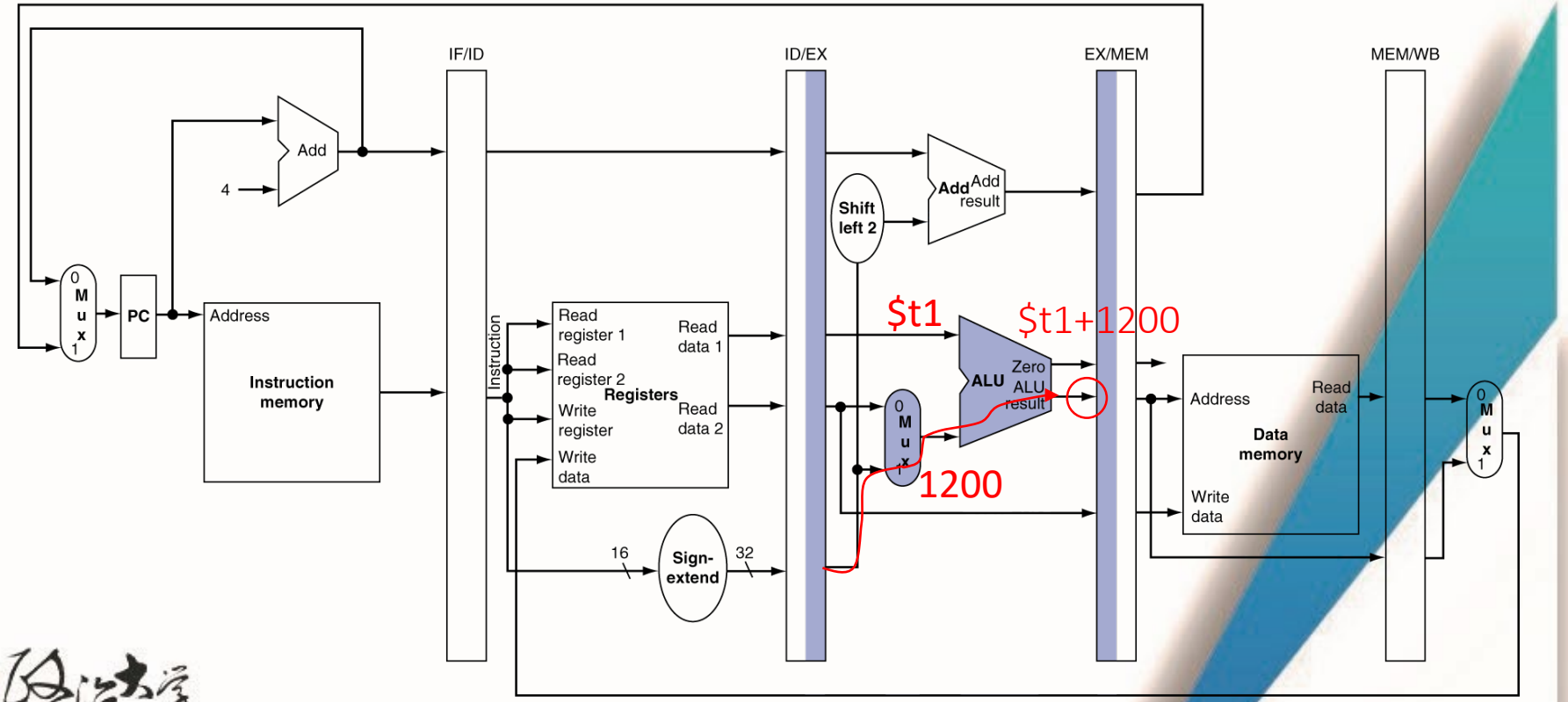
$t1 1200

ALUout = A + sign-ext(IR[15-0])

Calculate Base + Offset

lw $t0, 1200($t1)

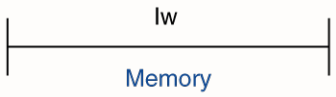


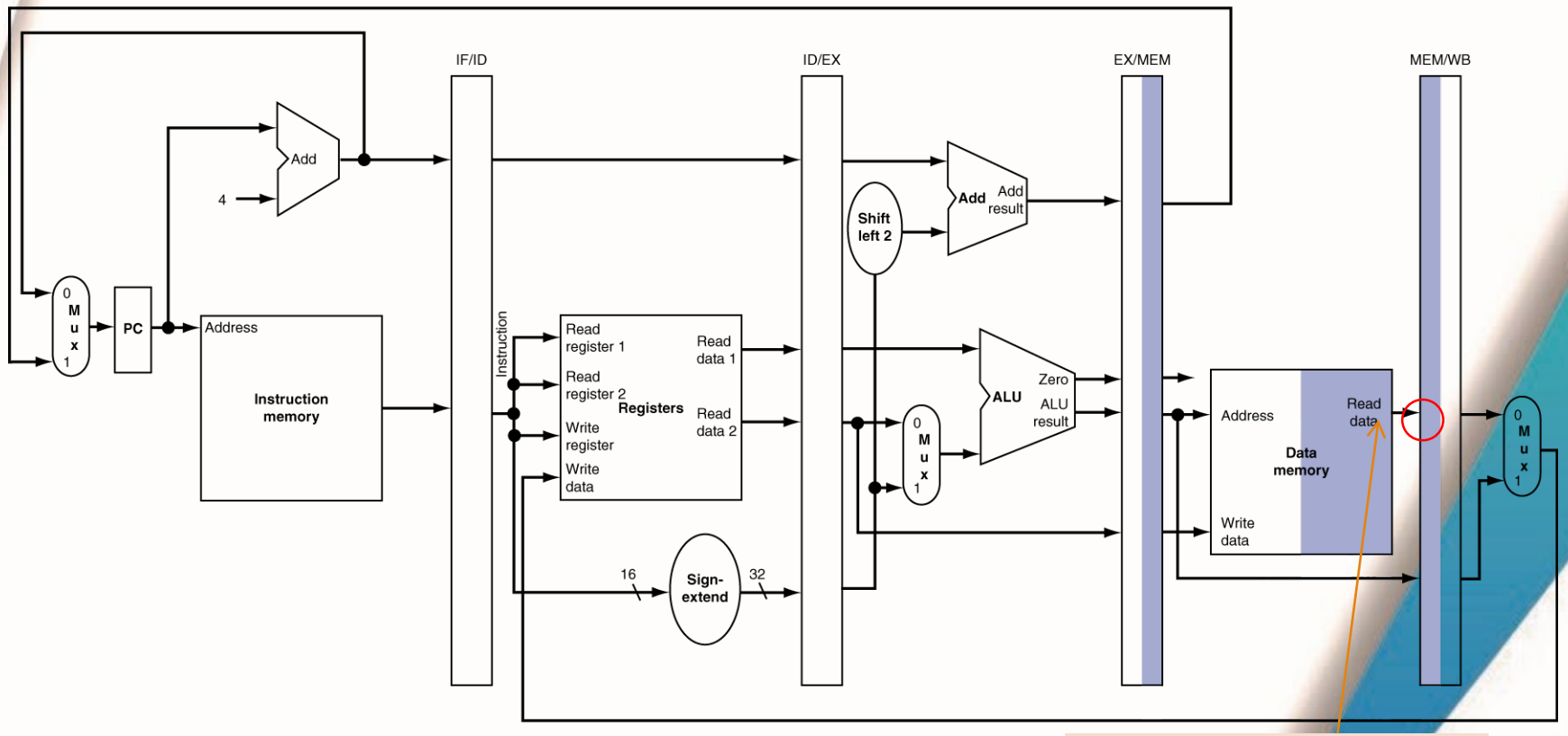



# MEM Stage - lw

*Read data from the Data Memory*

lw $t0, 1200($t1)

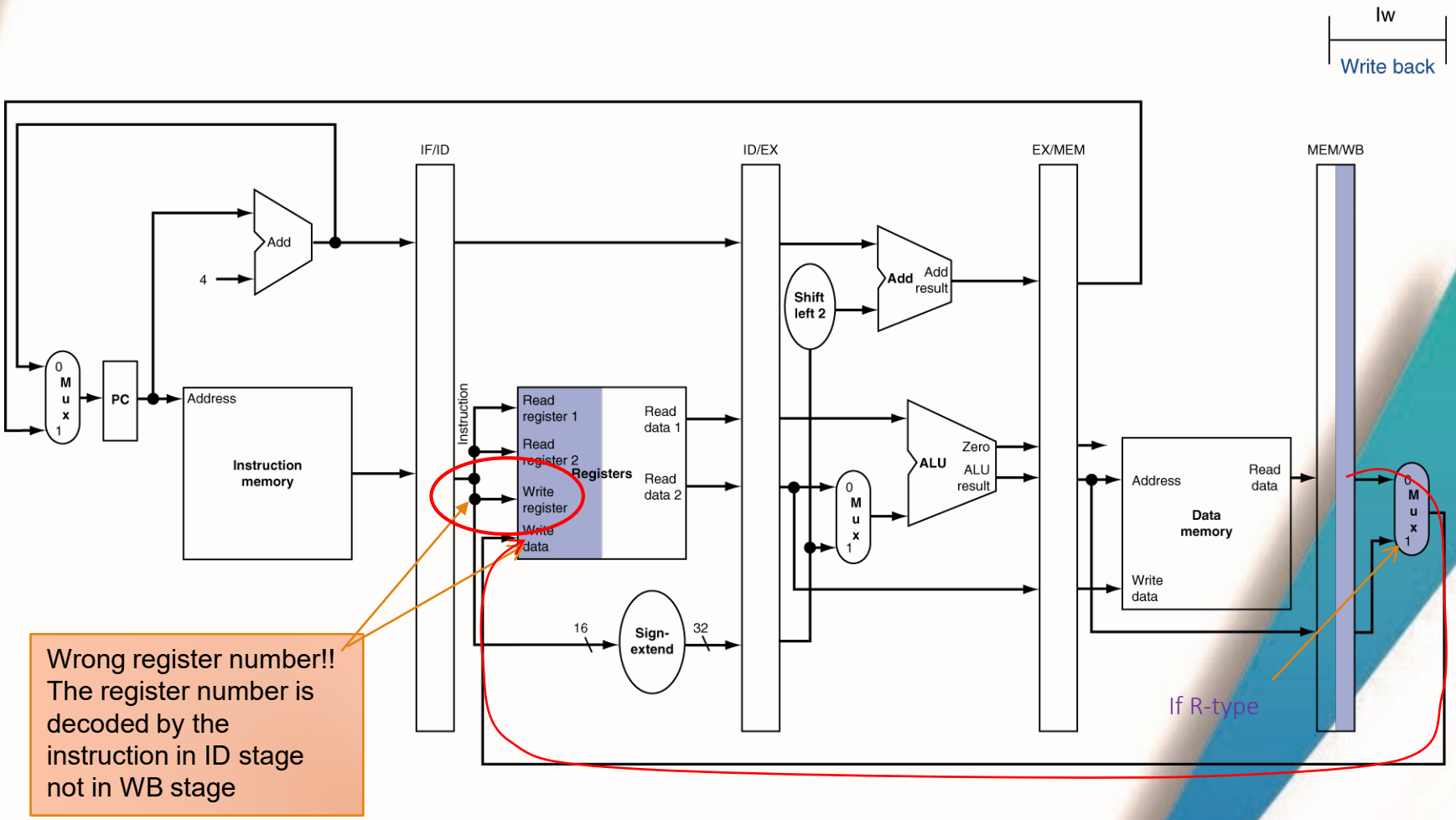



Read MEM[$t1+1200]

# WB Stage - lw

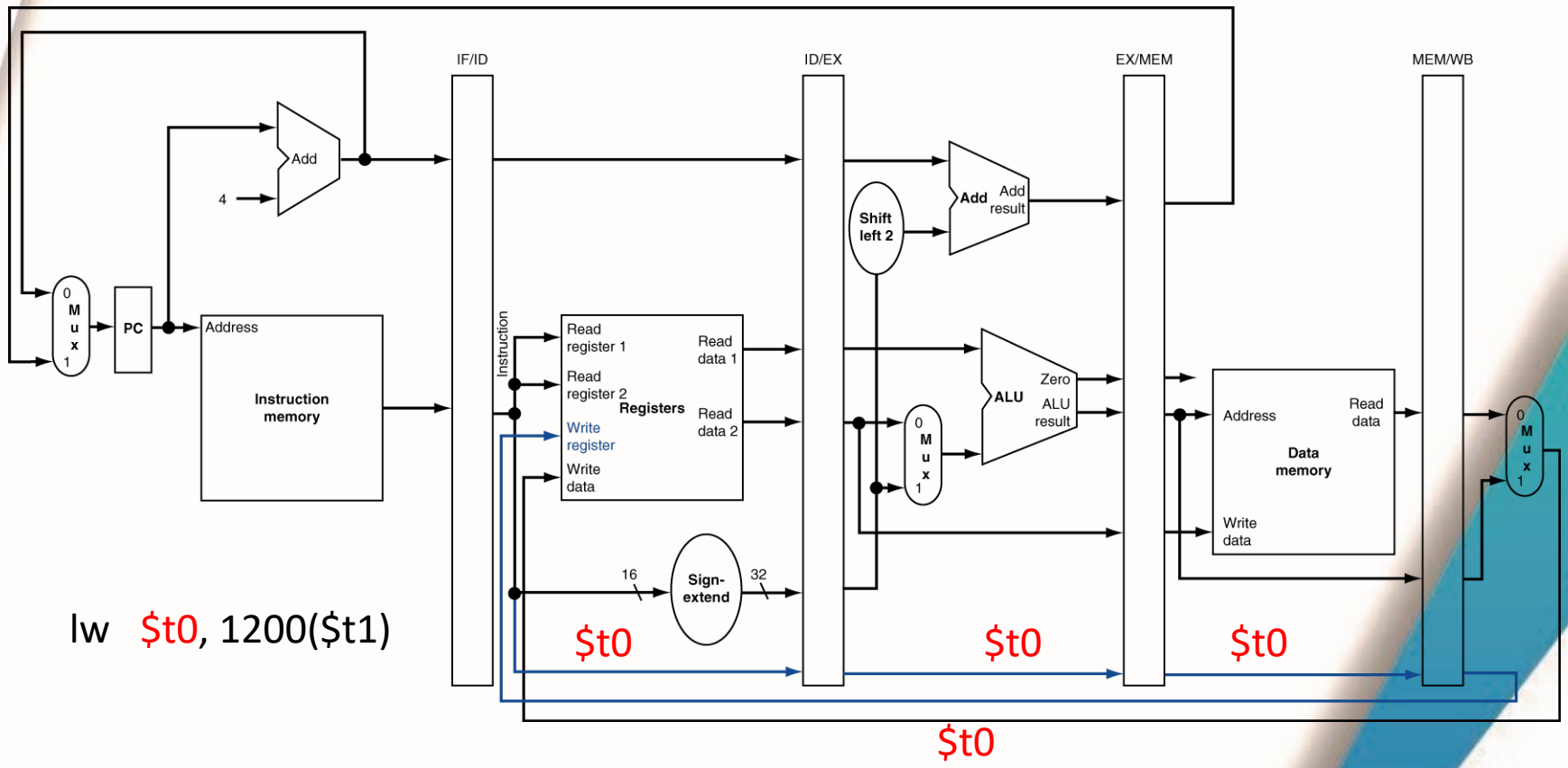
*Write the data back to the register*




Wrong register number!!
The register number is decoded by the instruction in ID stage not in WB stage

# WB Stage - lw

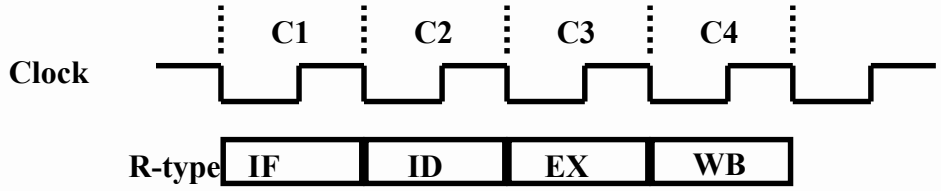
## Corrected Datapath for Load




lw $t0, 1200($t1)

Write register number shall be decided by the instruction in WB stage.

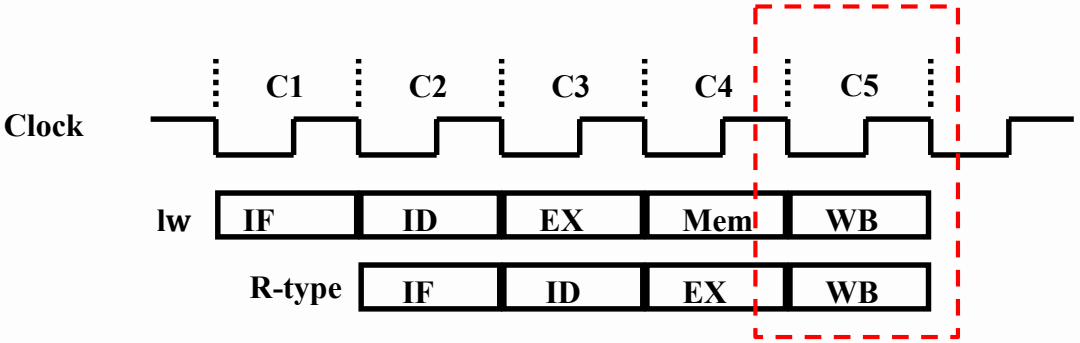
# R-type Instructions




There are four stages for a R-type instruction

- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

# Pipelining R-type and load

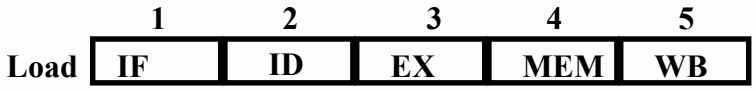



- If R-type takes four cycles and lw taks five cycles, we then have a *structural hazard*:
  - lw and R-type instruction both write results into the register file on the same stage but the register only has one write port.

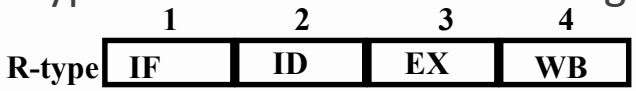
Ref: Prof. T.-T. Huang, NTHU, CS4100 course slides, Pipelining

# Observation

- Structural hazard – two or more instructions try to access the same hardware resource
  - Load writes results into the register on its 5th stage




  - R-type writes results into the register on its 4th stage

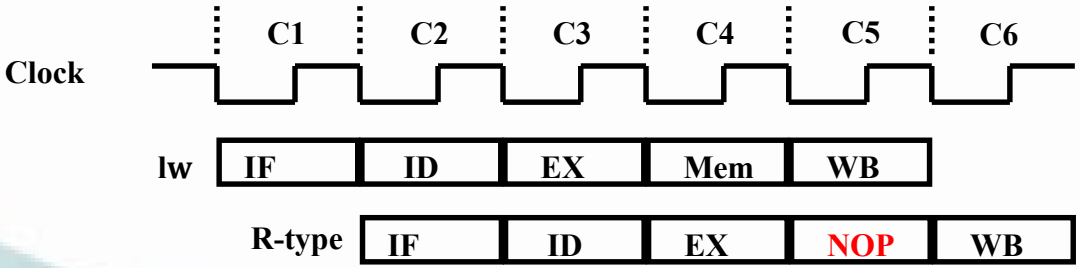



This can be resolved by inserting bubbles or making all the instructions having the same length

# Make Instructions having the Same Length

Make R-type instructions to run five stages

- R-type WB at Stage 5
- MEM stage == NOP


Ref: Prof. T.-T. Huang, NTHU, CS4100 course slides, Pipelining

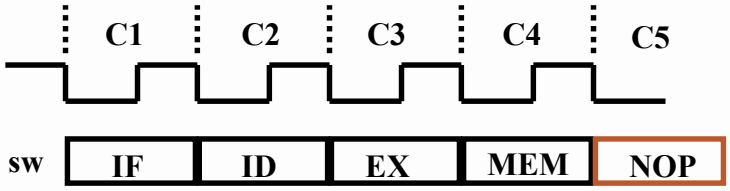
# Example – sw

There are four stages for sw

- IF: fetch the instruction from the Instruction Memory
- ID: fetch registers and decode the instruction
- EX: compute the target memory address
- MEM: write the reg data into the Data Memory

Make it having five stages

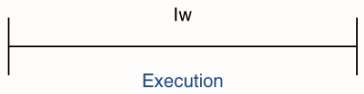
- WB == NOP

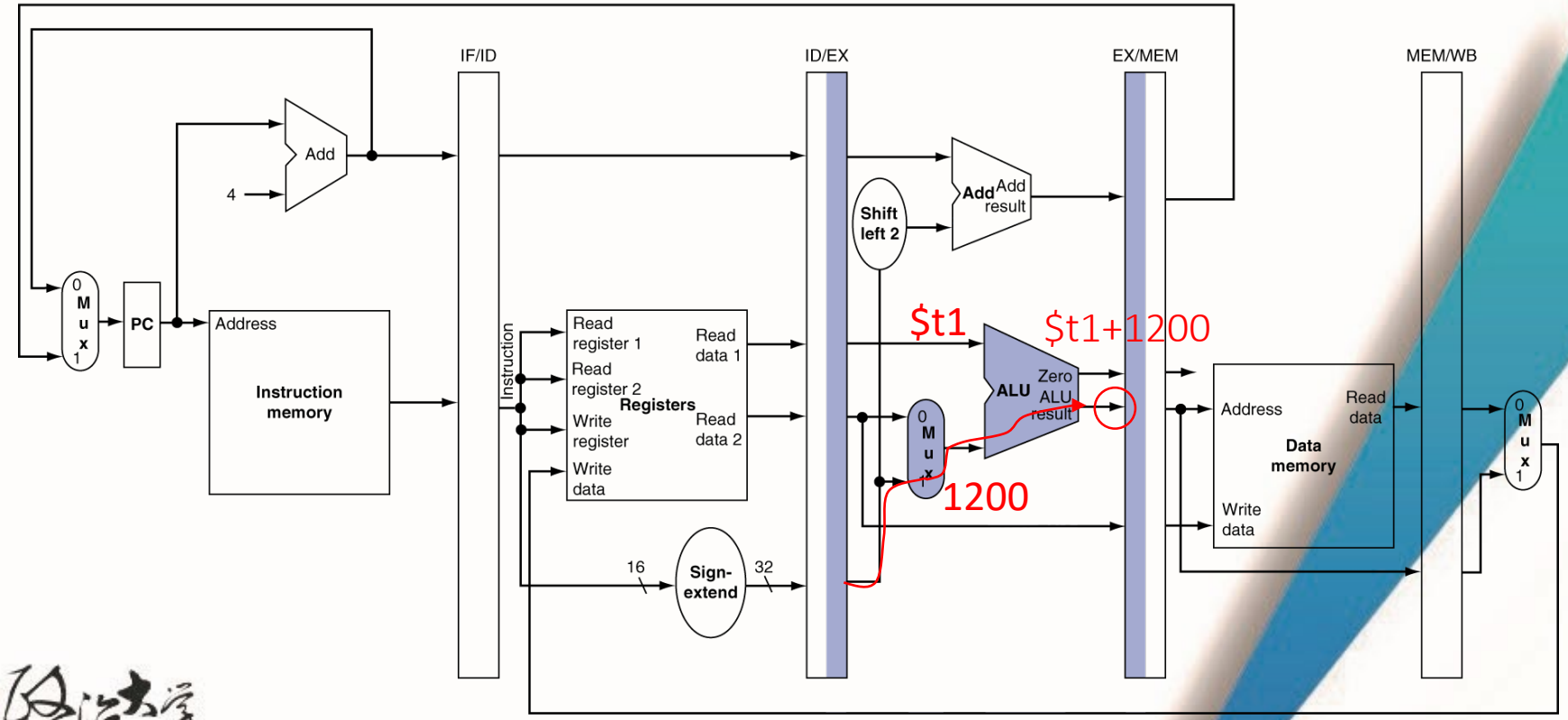


# EX Stage - sw

*Compute the memory address ($t1+1200)*

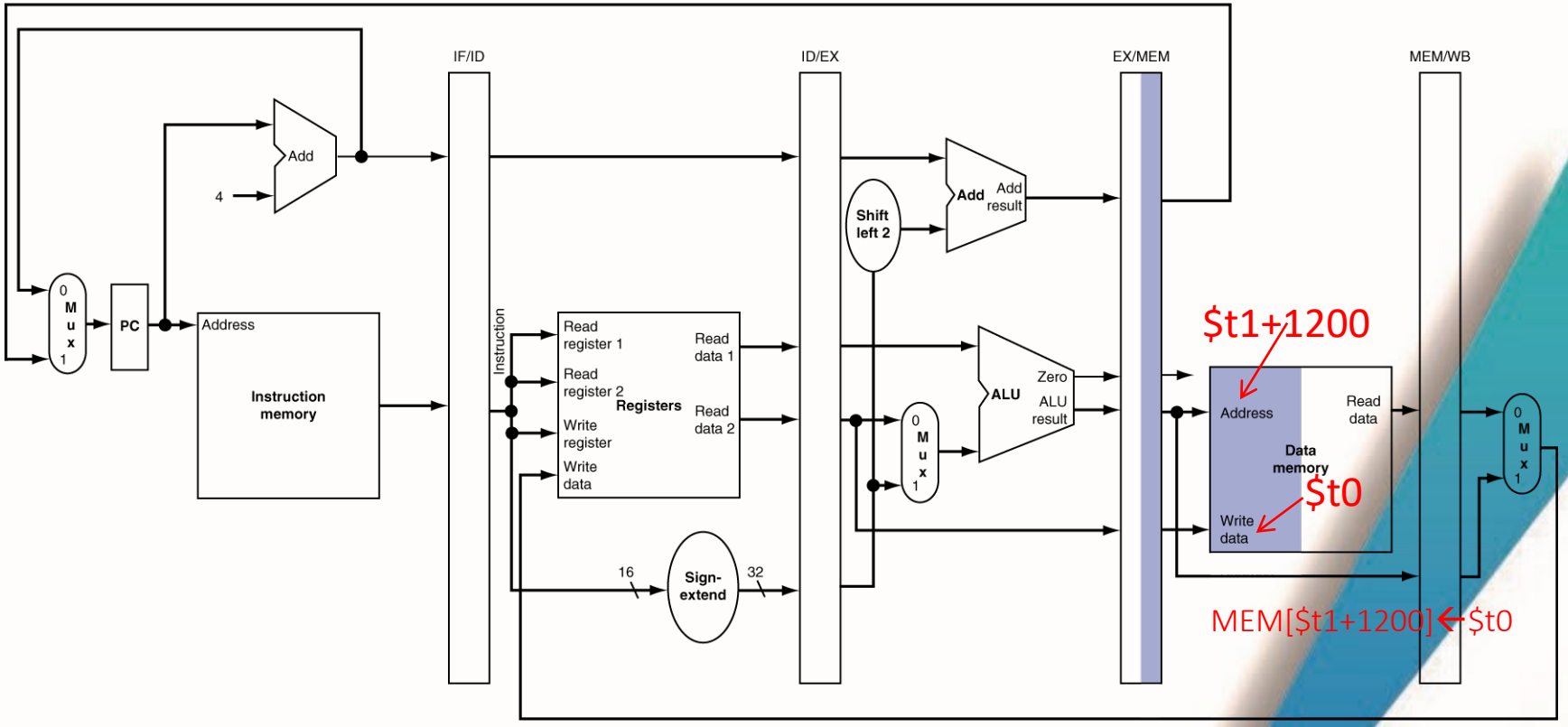
sw $t0, 1200($t1)



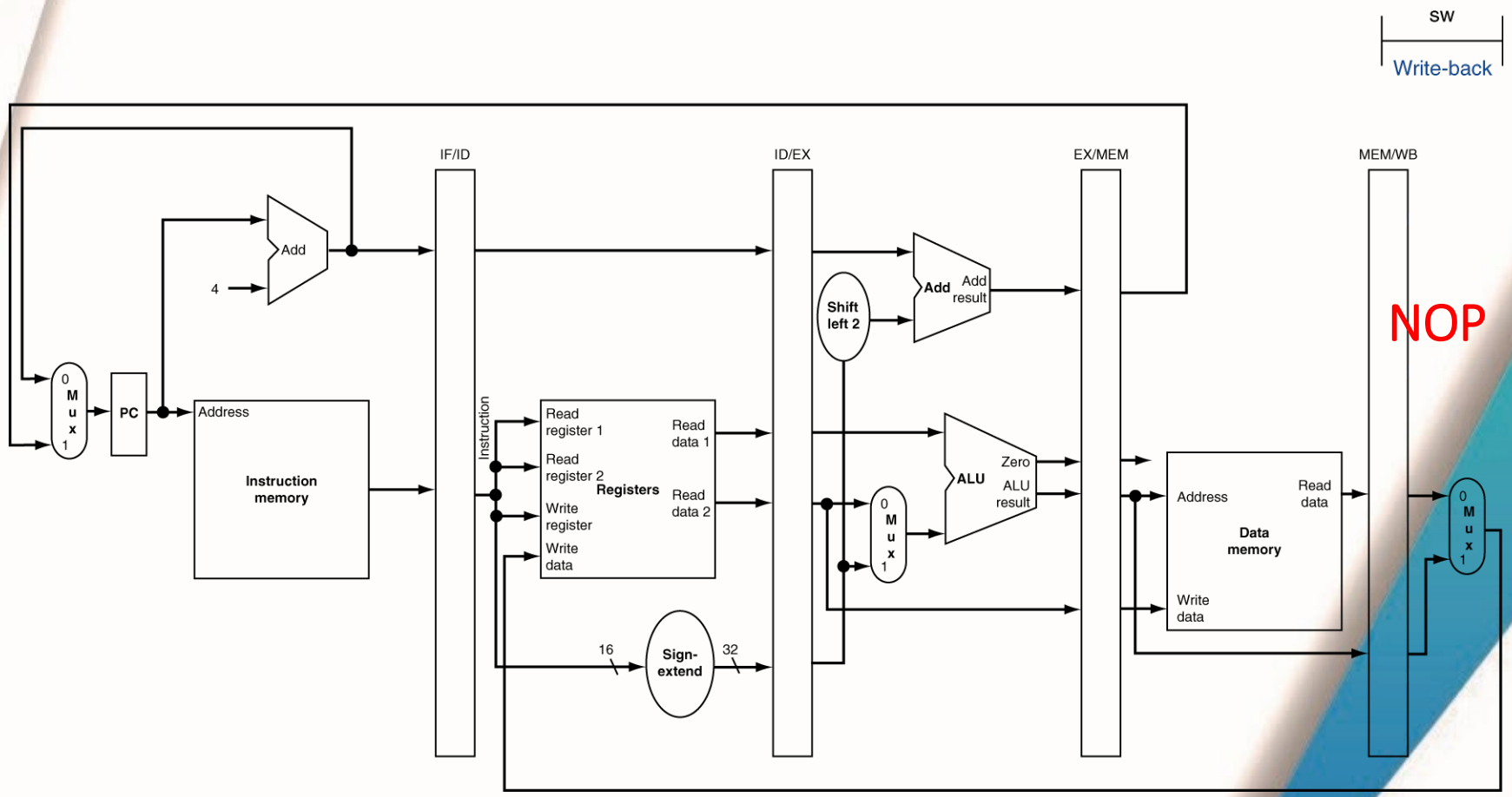



# MEM Stage - sw

sw $t0, 1200($t1)

# WB Stage - sw

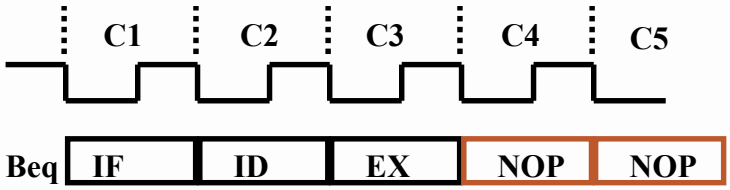

SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
NOP
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

# Example - beq

- IF: fetch the instruction from the Instruction Memory
- ID: fetch registers and decode the instruction
- EX:
  - Compares the two register operands with ALU
  - Compute the target address
  - If branch is taken, write the target address into PC; otherwise, write PC+4

Two extra stages:

- MEM: NOP
- WB: NOP



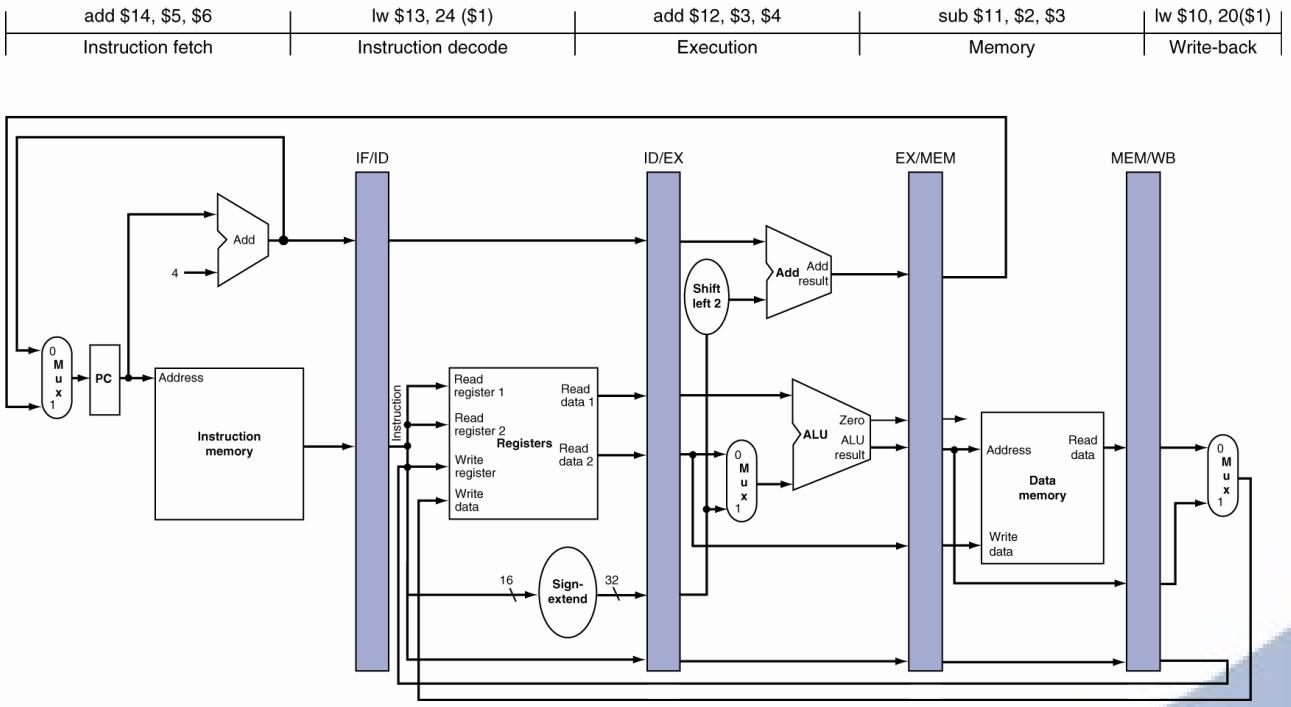

It's possible to calculate the branch address, and update the PC during ID stage with extra hardware added to test registers.

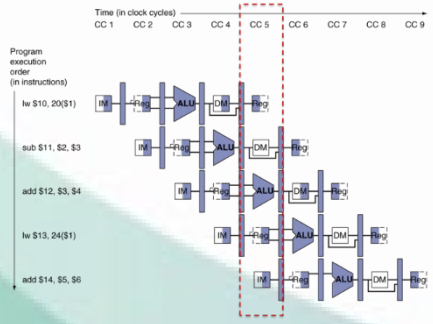
# Pipeline Diagrams

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - "multi-clock-cycle" diagram
    - Graph of operation over time

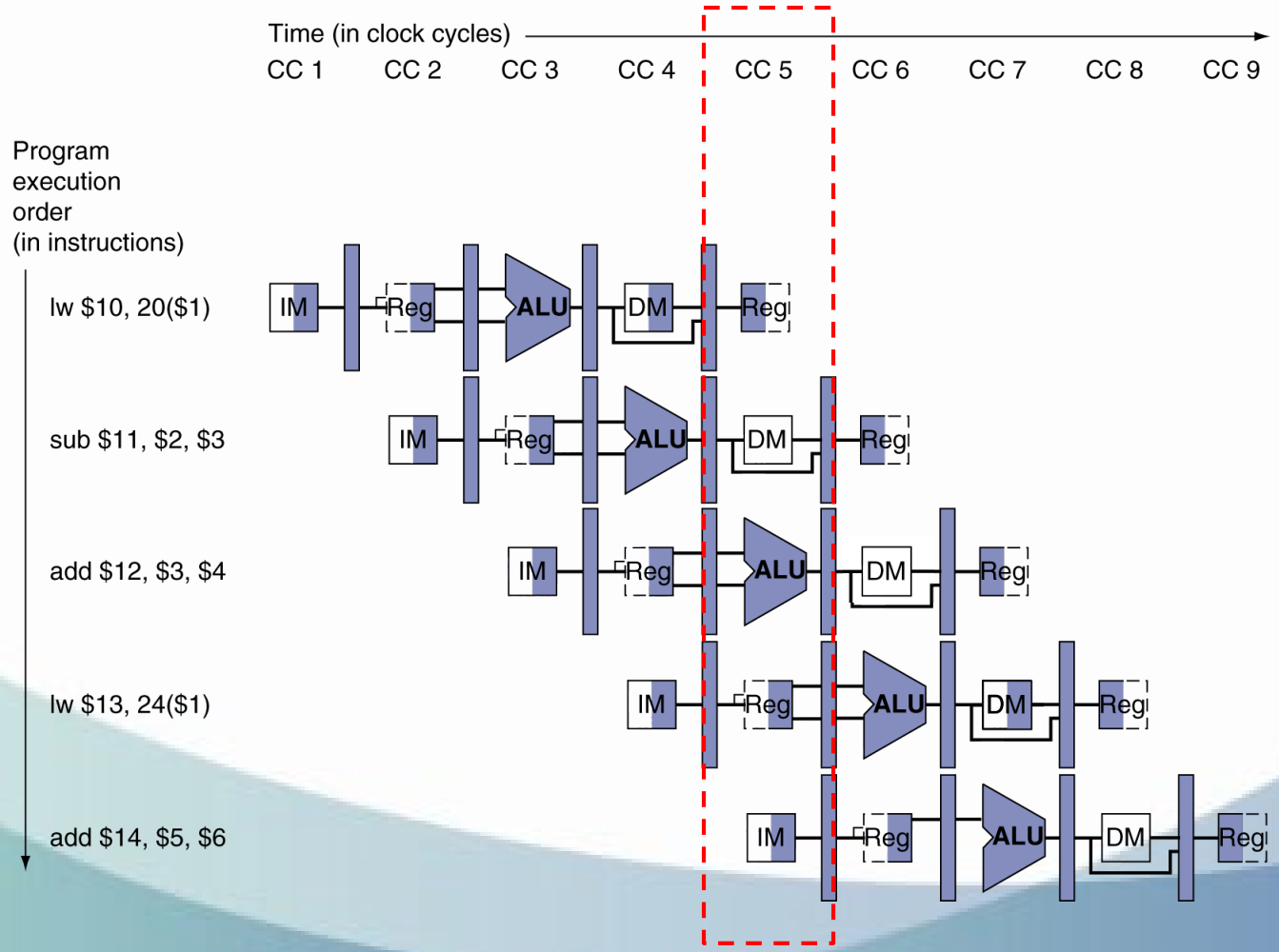
# Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

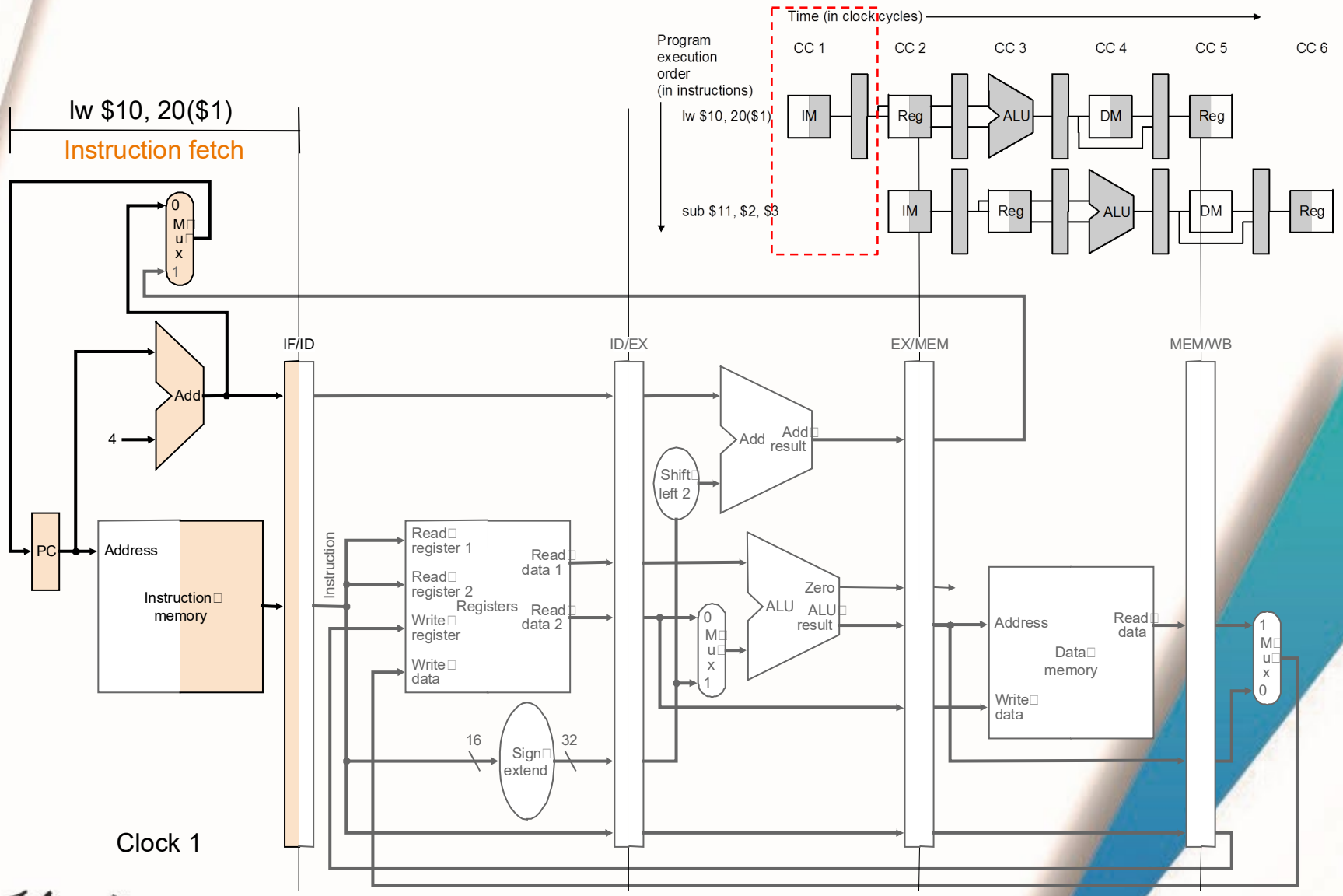


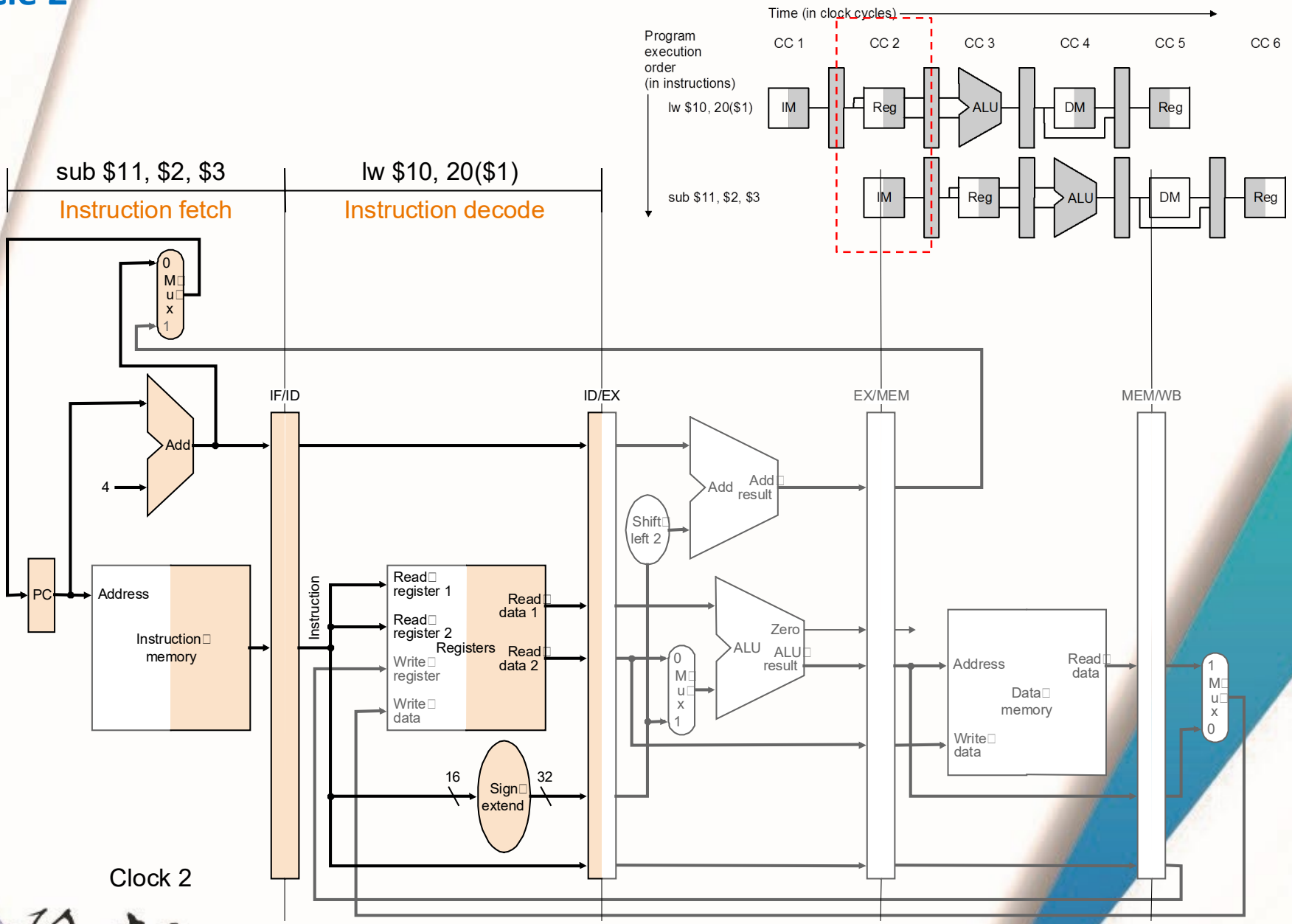
# Multi-Cycle Pipeline Diagram



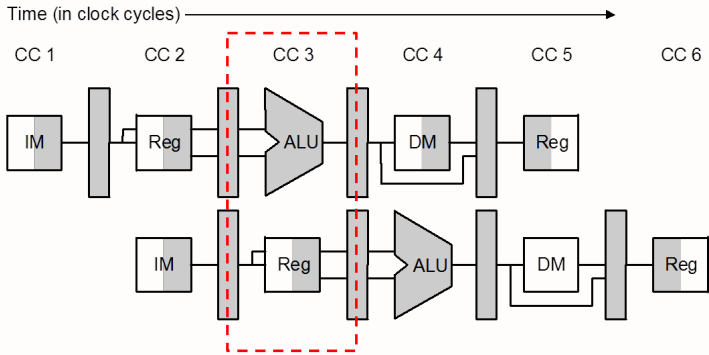
# Example – Cycle 1

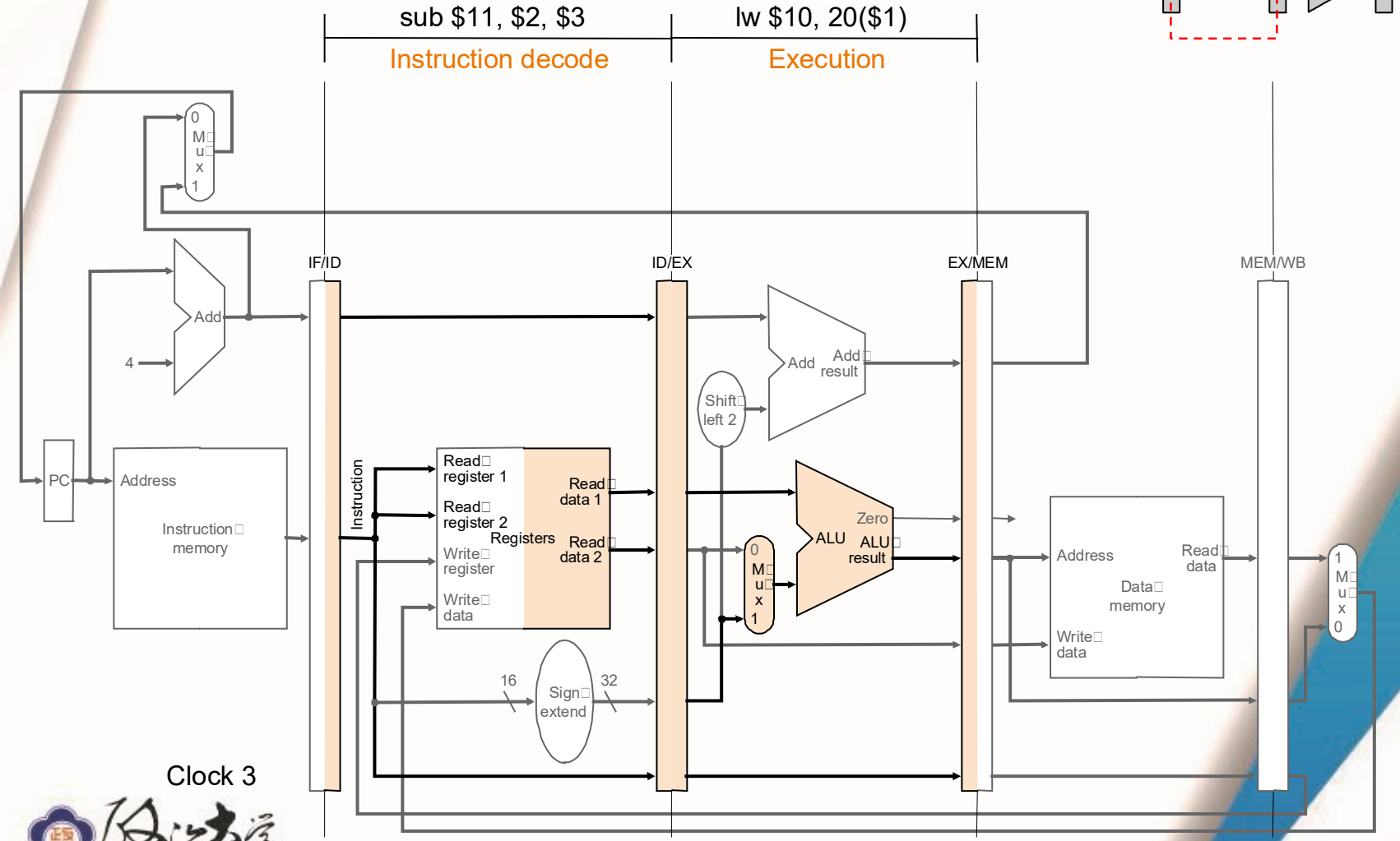


# Cycle 2


Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
lw $10, 20($1)
sub $11, $2, $3
IM
Reg
ALU
DM
Reg
sub $11, $2, $3
lw $10, 20($1)
Instruction fetch
Instruction decode
IF/ID
ID/EX
EX/MEM
MEM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Shift
left 2
Add
Add
result
ALU
Zero
ALU
result
16
Sign
extend
32
Address
Data
memory
Read
data
Write
data
1
M
u
x
0
Clock 2

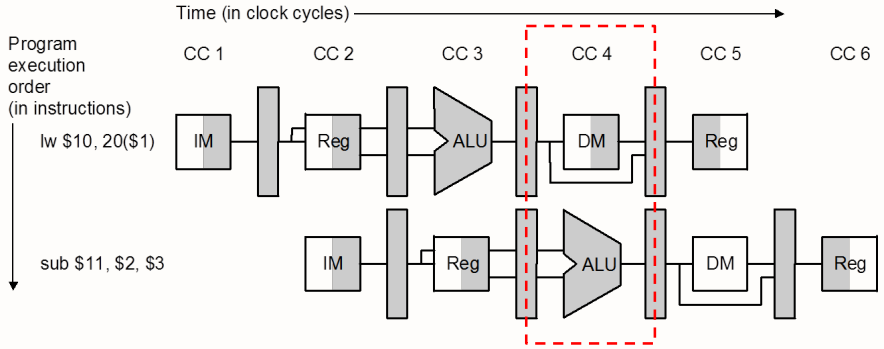
# Cycle 3

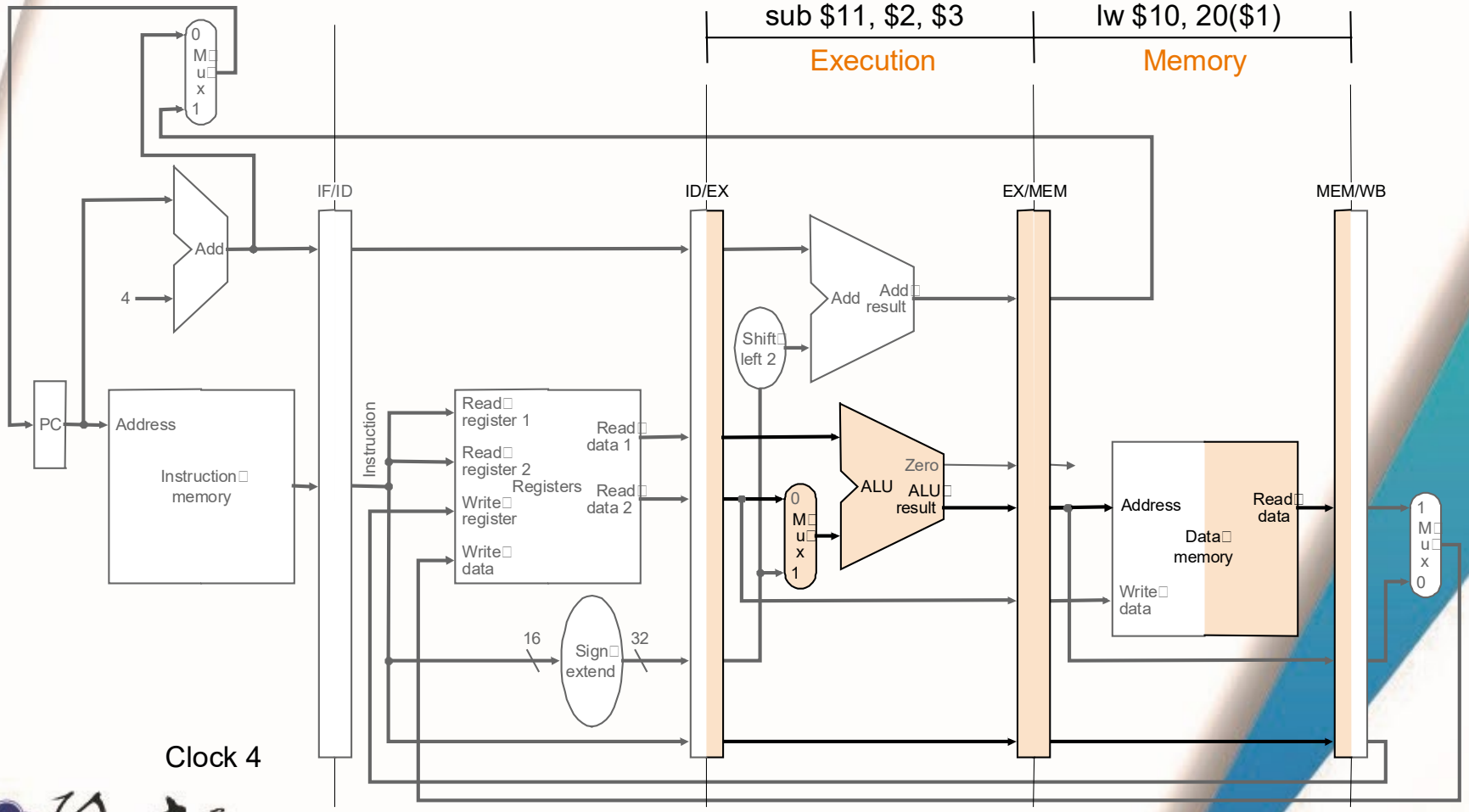



Clock 3

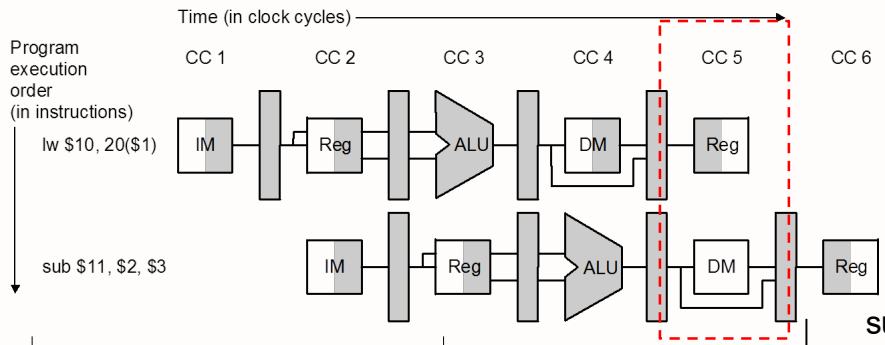
# Cycle 4

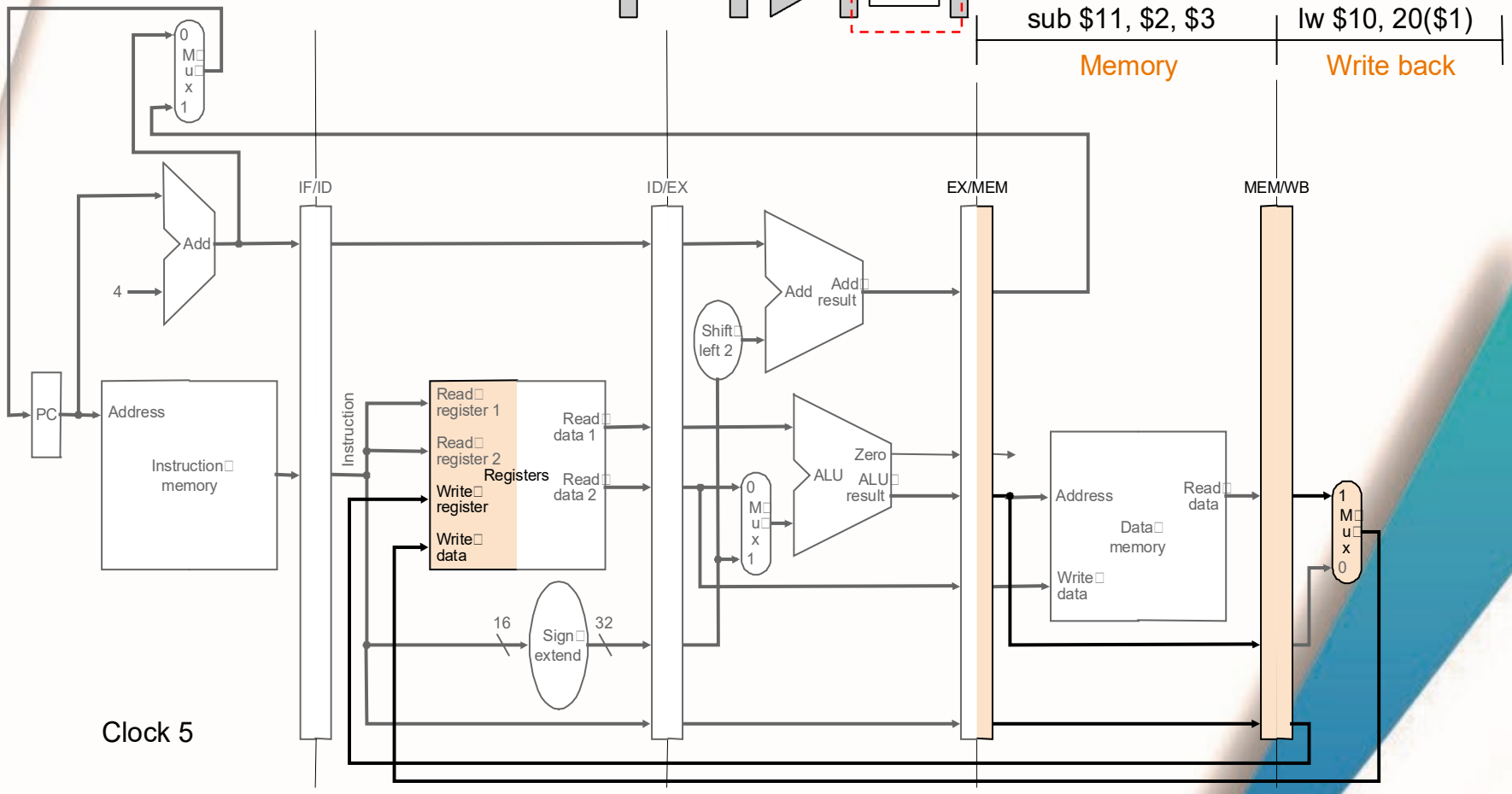



Clock 4

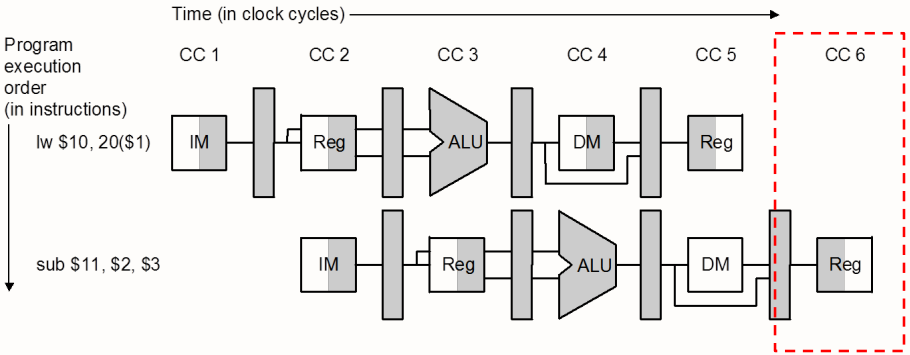
# Cycle 5

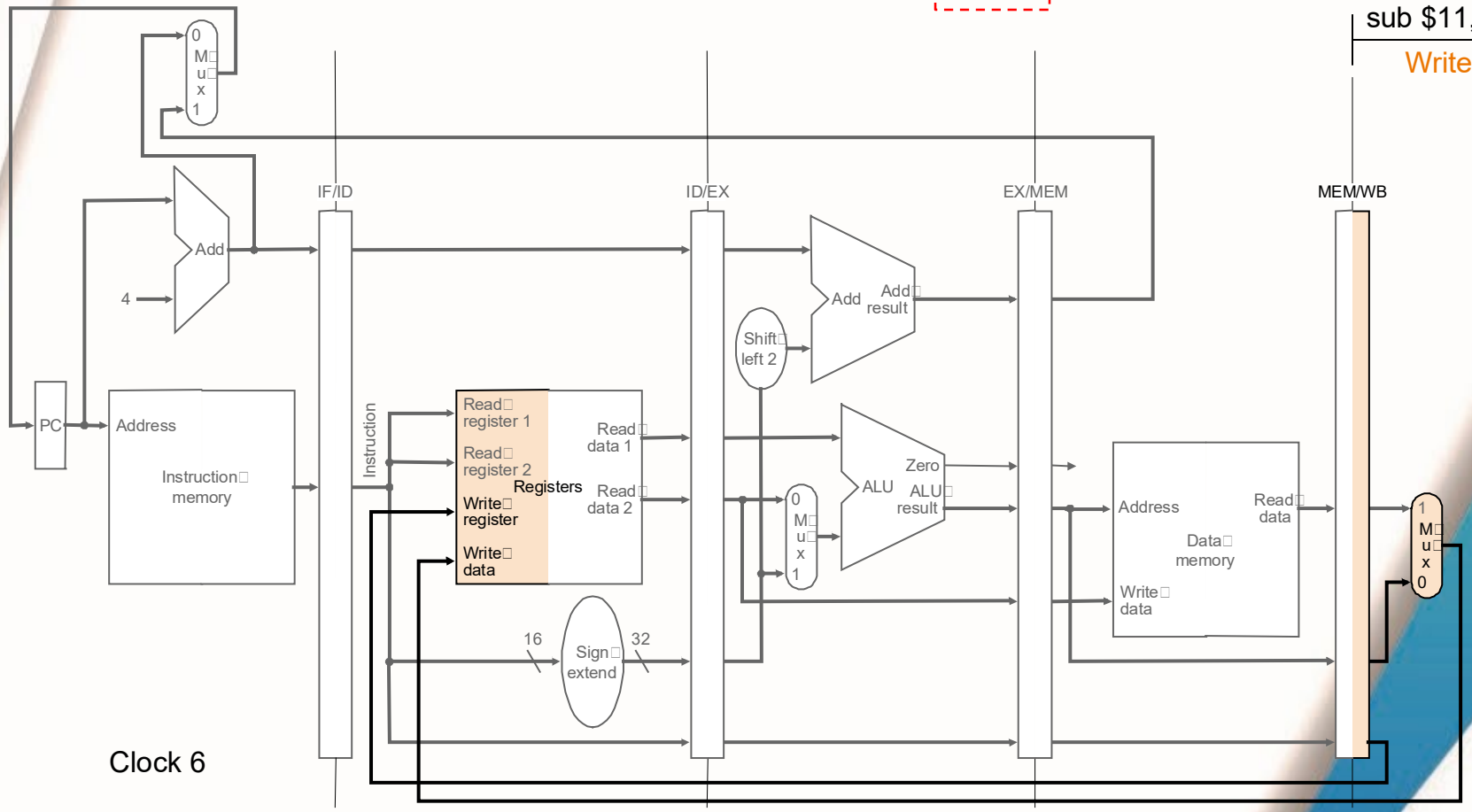



Clock 5

# Cycle 6

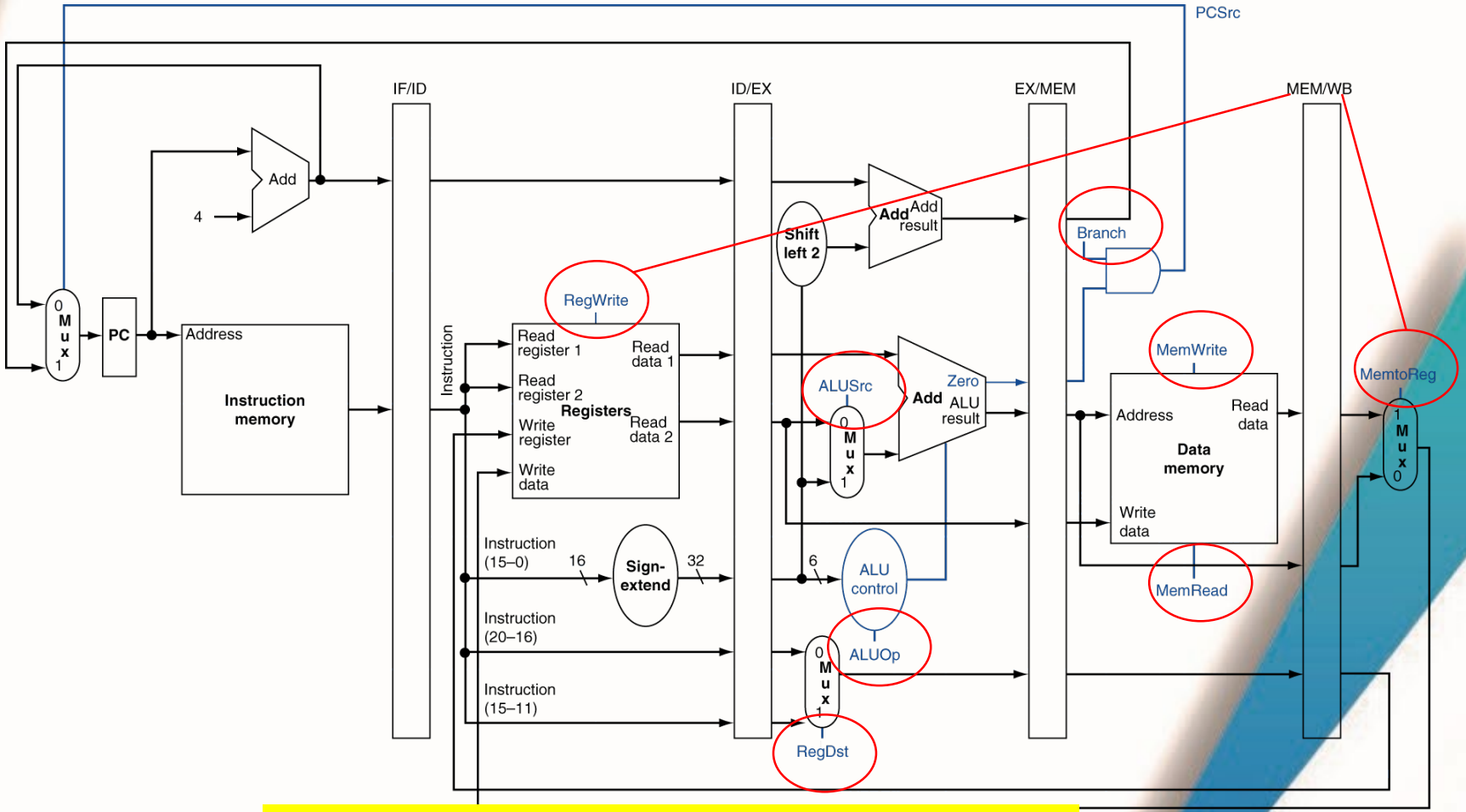



Clock 6

# Pipelined Control (Simplified)

*Ignoring branch instructions here for now*




EX, MEM, and WB use control signals since all the signals are decoded in ID stage

# Group Signals According to Stages

- Use the same the control signals as those of a single-cycle CPU
- Group them in three categories

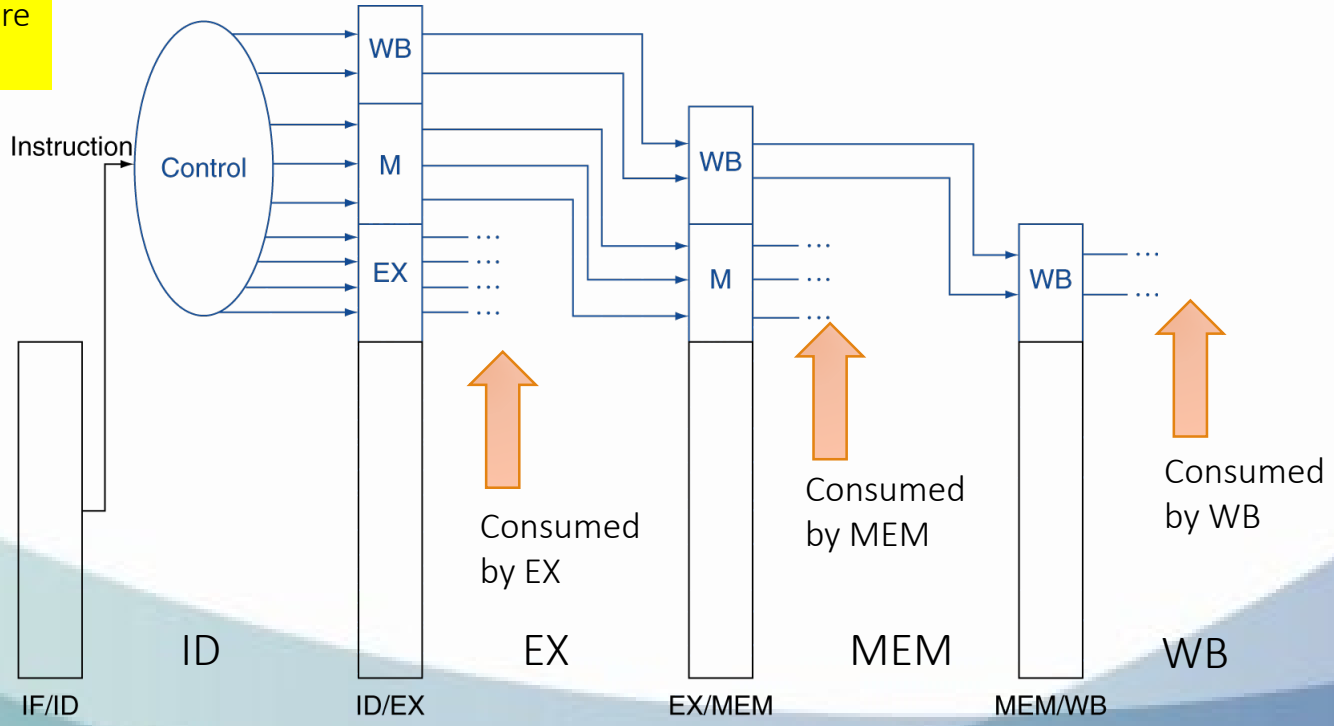
| | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-type | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

Ref: Prof. T.-T. Huang, NTHU, CS4100 course slides, Pipelining

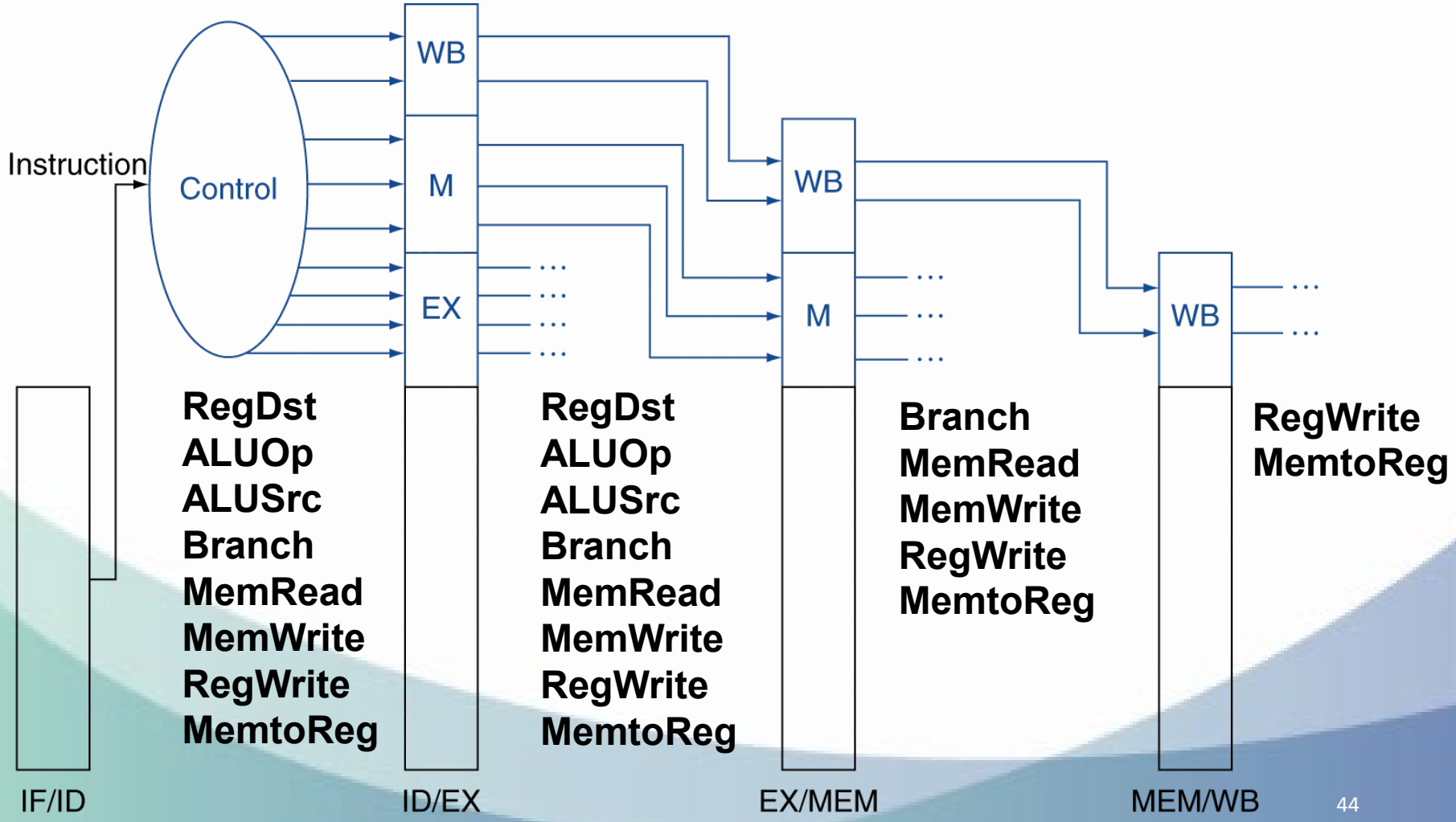
# Data Stationary Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - These signals need to be store into pipeline register

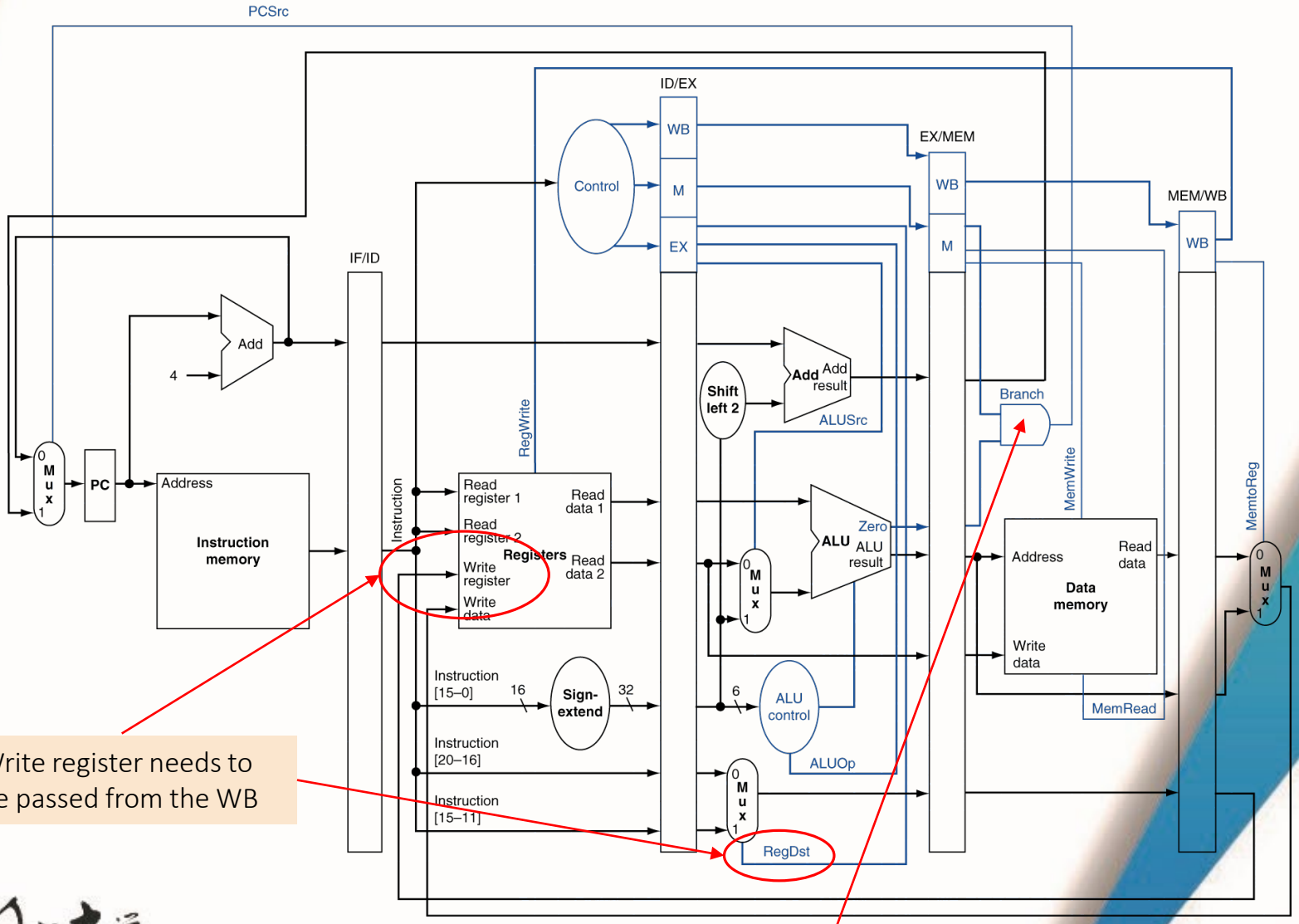
All the control signals are decoded in ID stage



# Data Stationary Control



# Pipelined Control



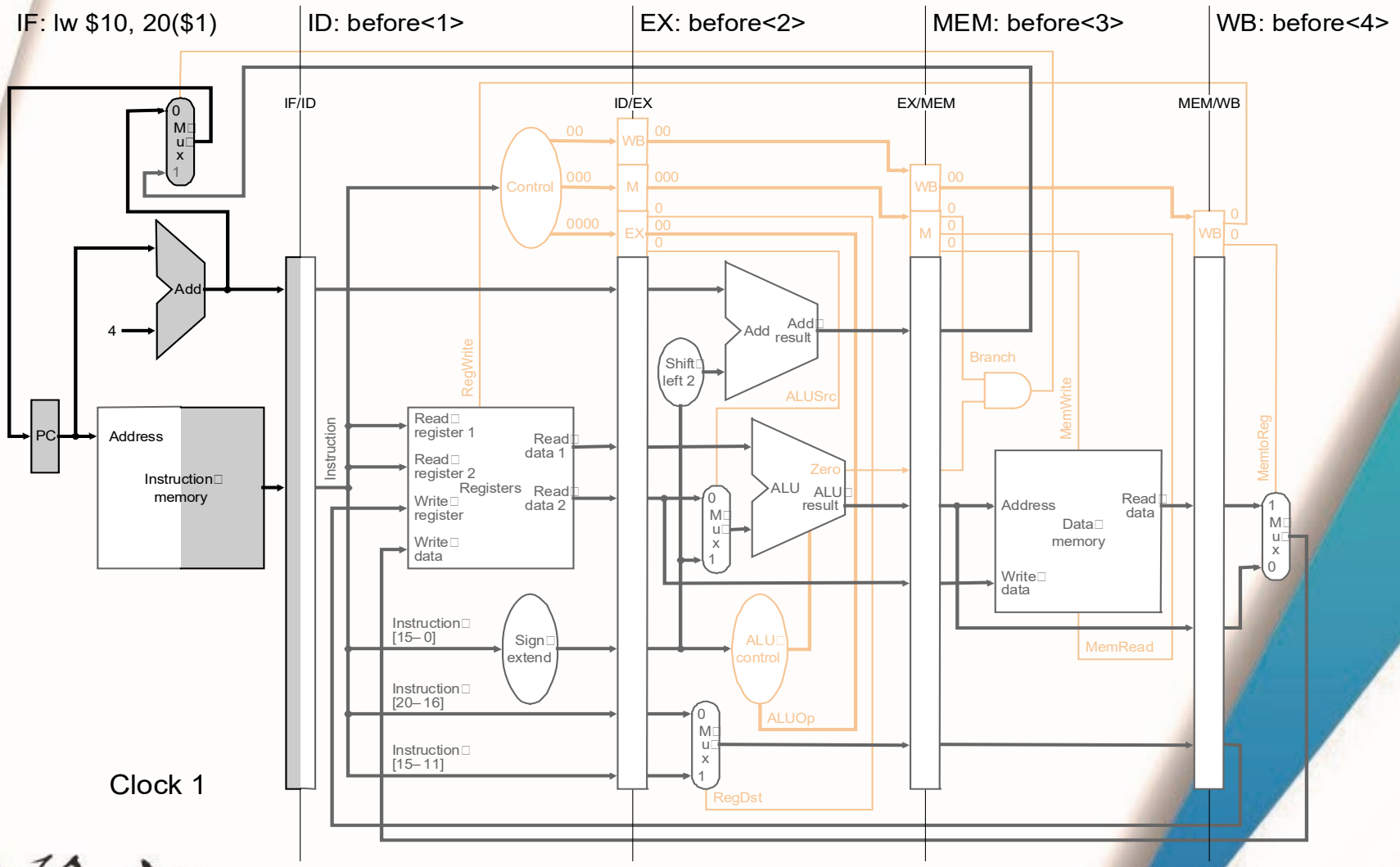

Write register needs to be passed from the WB

Branch could be determined in ID stage

# Example

```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or   $13, $6, $7
add  $14, $8, $9
```

# Cycle 1

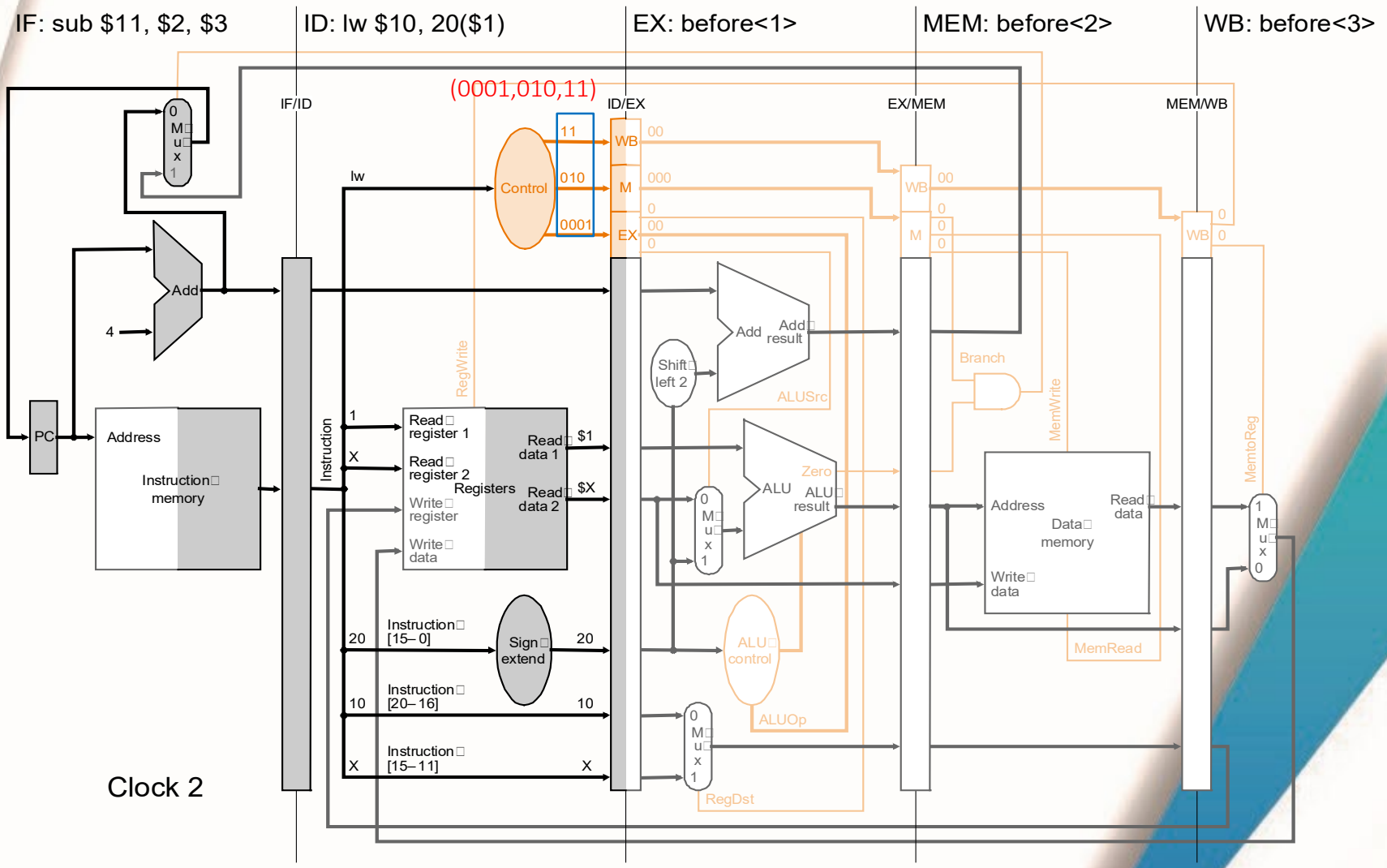

IF: lw $10, 20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Control
WB
M
EX
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
Sign extend
Instruction [20–16]
Instruction [15–11]
ALU control
ALUOp
RegDst
Clock 1

# Group Signals According to Stages

lw $10, 20($1)

| | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-type | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

# Cycle 2

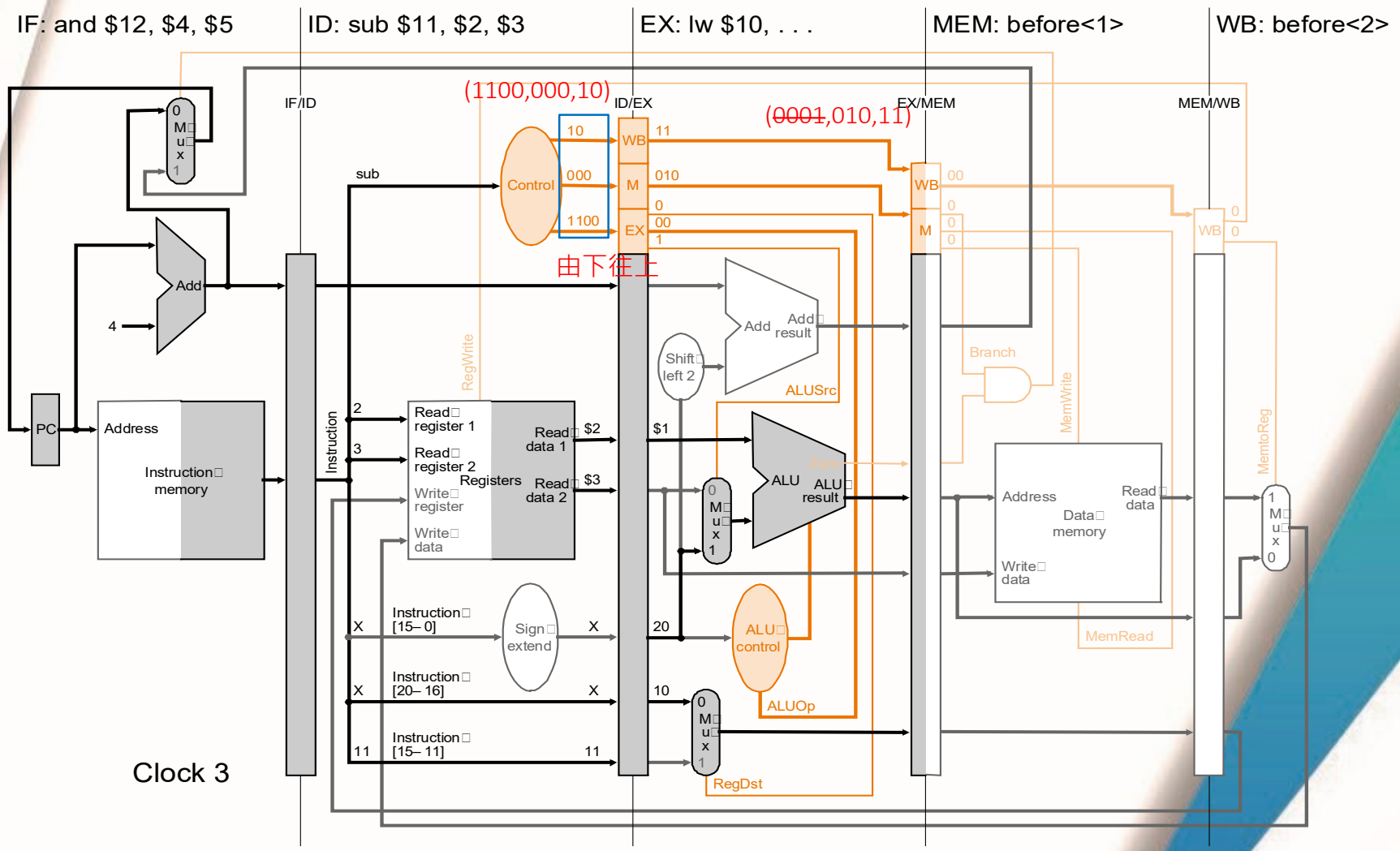

IF: sub $11, $2, $3
ID: lw $10, 20($1)
EX: before<1>
MEM: before<2>
WB: before<3>
(0001,010,11)
IF/ID
ID/EX
EX/MEM
MEM/WB
lw
Control
11
010
0001
WB
M
EX
00
000
0
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$1
$X
Shift left 2
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
MemRead
MemtoReg
Data memory
Read data
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign extend
20
10
X
Clock 2

# Group Signals According to Stages

```
sub    $11, $2, $3
```

| | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-type | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

# Cycle 3

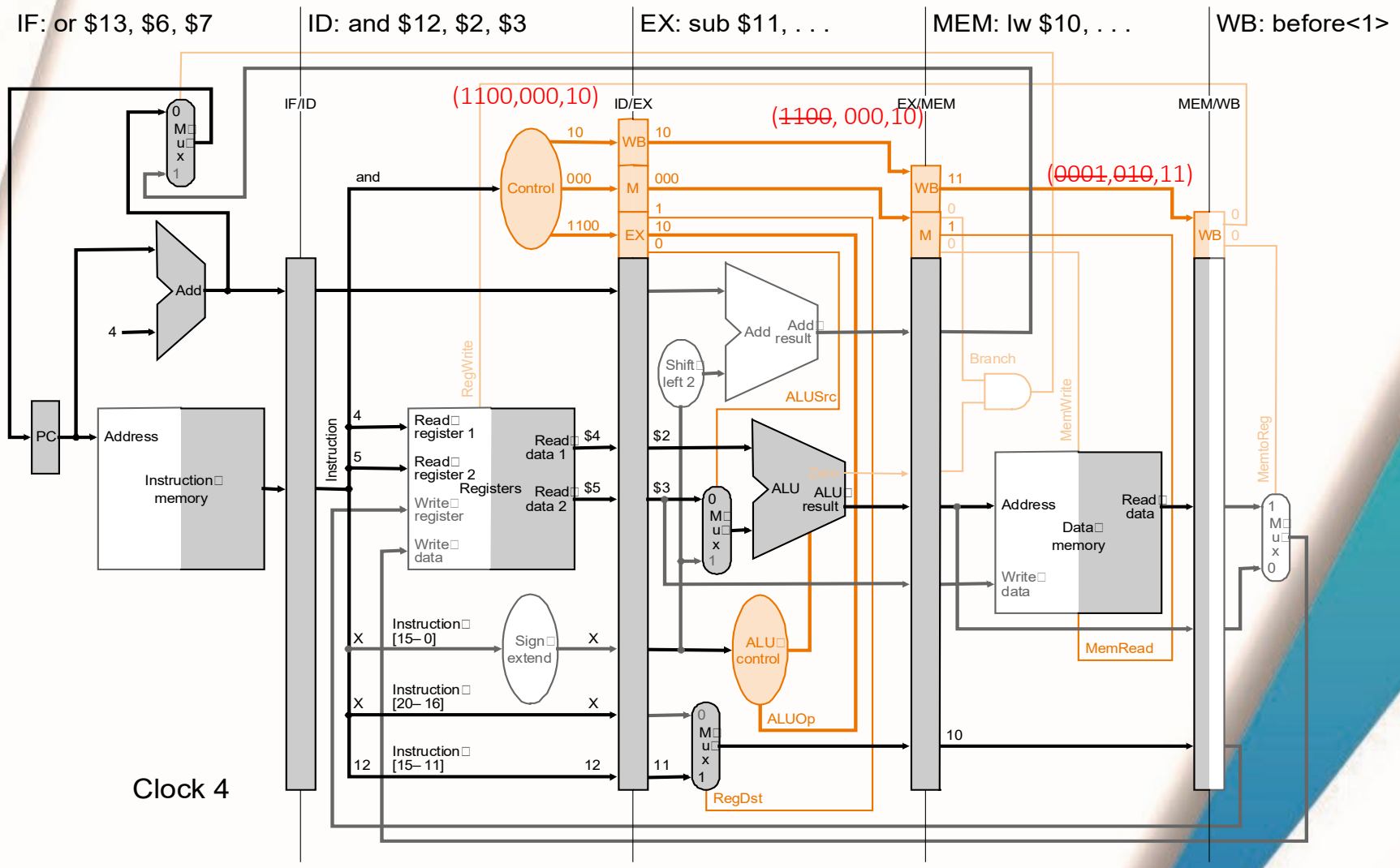

IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100,000,10)
(0001,010,11)
由下往上
sub
Control
10
000
1100
WB
M
EX
11
010
0
00
1
00
0
0
0
0
0
0
Add
4
PC
Address
Instruction memory
Instruction
2
3
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$2
$3
RegWrite
Shift left 2
Add
Add result
ALUSrc
$1
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign extend
X
20
10
11
ALU control
ALUOp
RegDst
Clock 3

# Group Signals According to Stages

```
and    $12, $4, $5
```

| | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-type | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

# Cycle 4

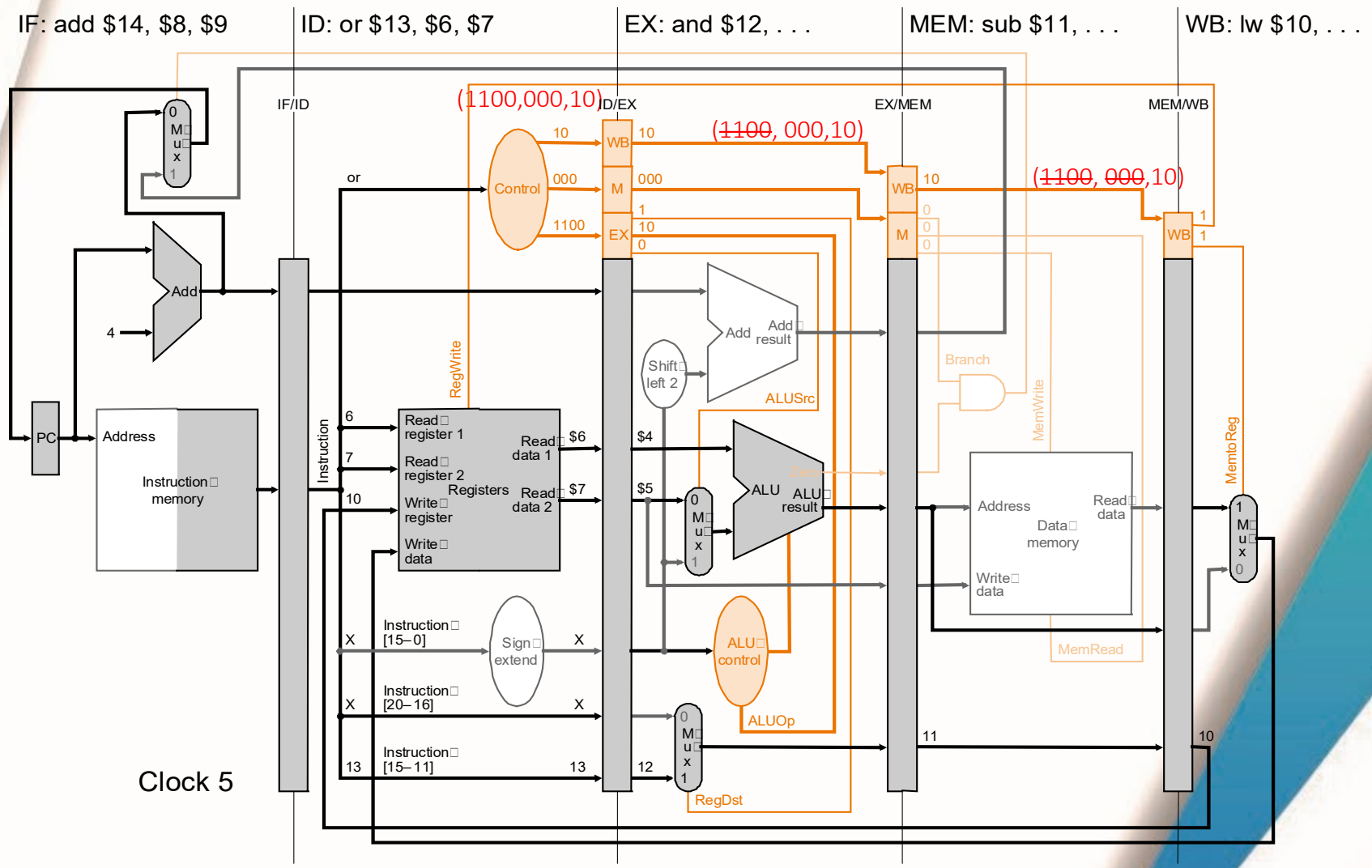

IF: or $13, $6, $7
ID: and $12, $2, $3
EX: sub $11, . . .
MEM: lw $10, . . .
WB: before<1>
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100,000,10)
(1100, 000,10)
(0001,010,11)
and
Control
10
000
1100
WB
M
EX
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$4
$5
$2
$3
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
MemRead
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign extend
ALU control
ALUOp
RegDst
Clock 4

# Group Signals According to Stages

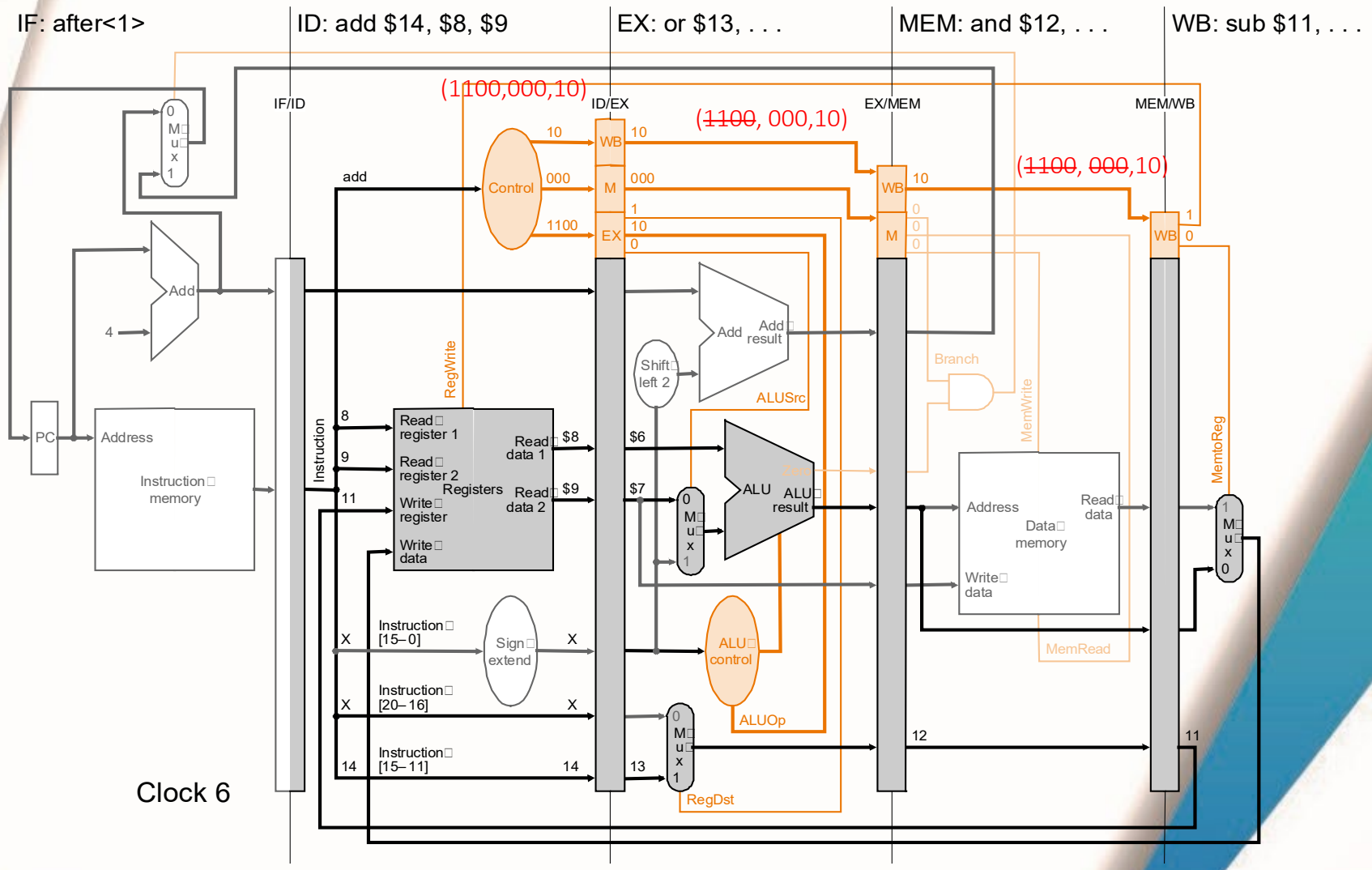
```
or      $13, $6, $7
```

| | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-type | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

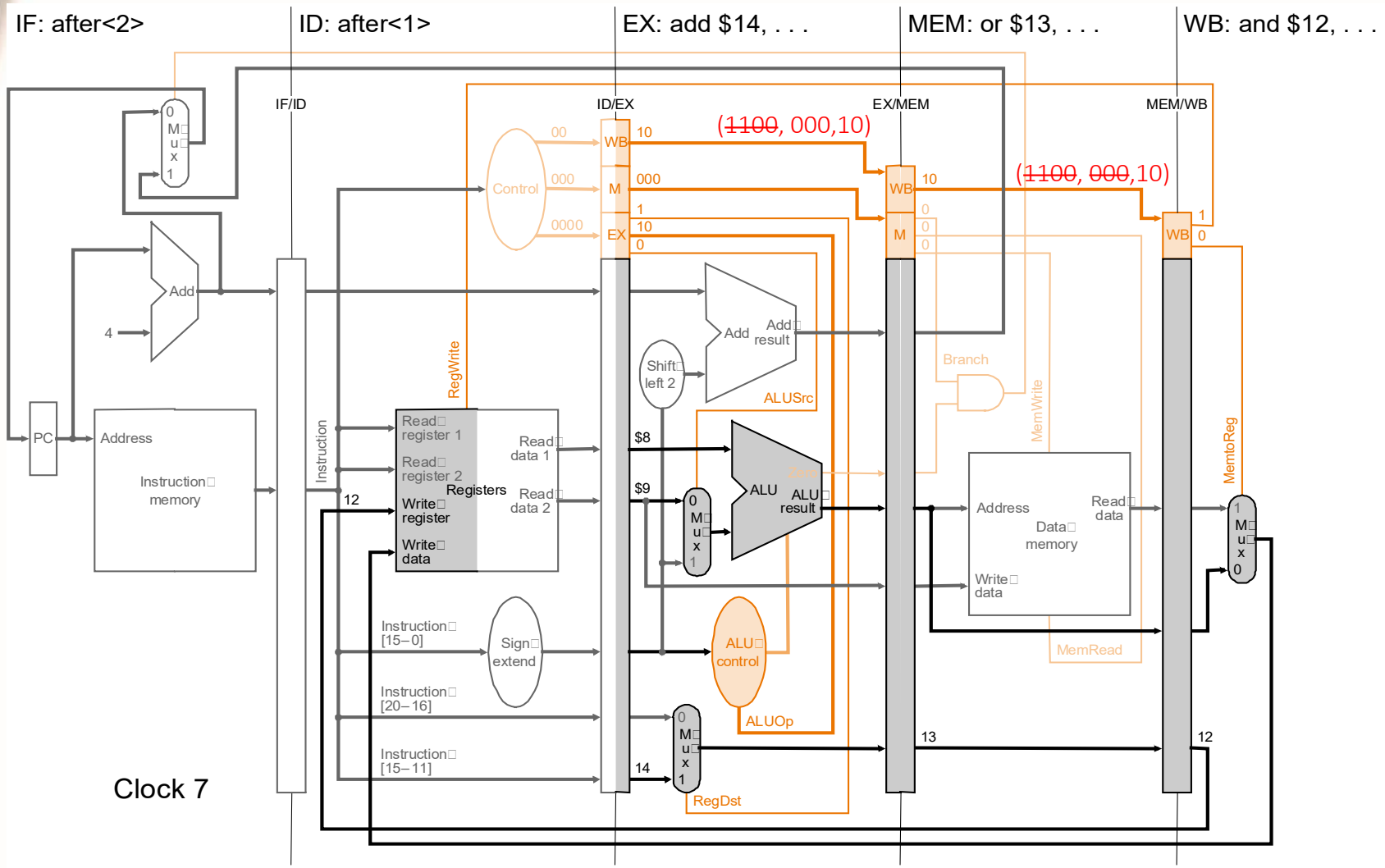
# Cycle 5


IF: add $14, $8, $9
ID: or $13, $6, $7
EX: and $12, . . .
MEM: sub $11, . . .
WB: lw $10, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100,000,10)
(1100, 000,10)
(1100, 000,10)
Control
WB
M
EX
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
MemRead
MemtoReg
Address
Data memory
Read data
Write data
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign extend
ALU control
ALUOp
RegDst
Clock 5

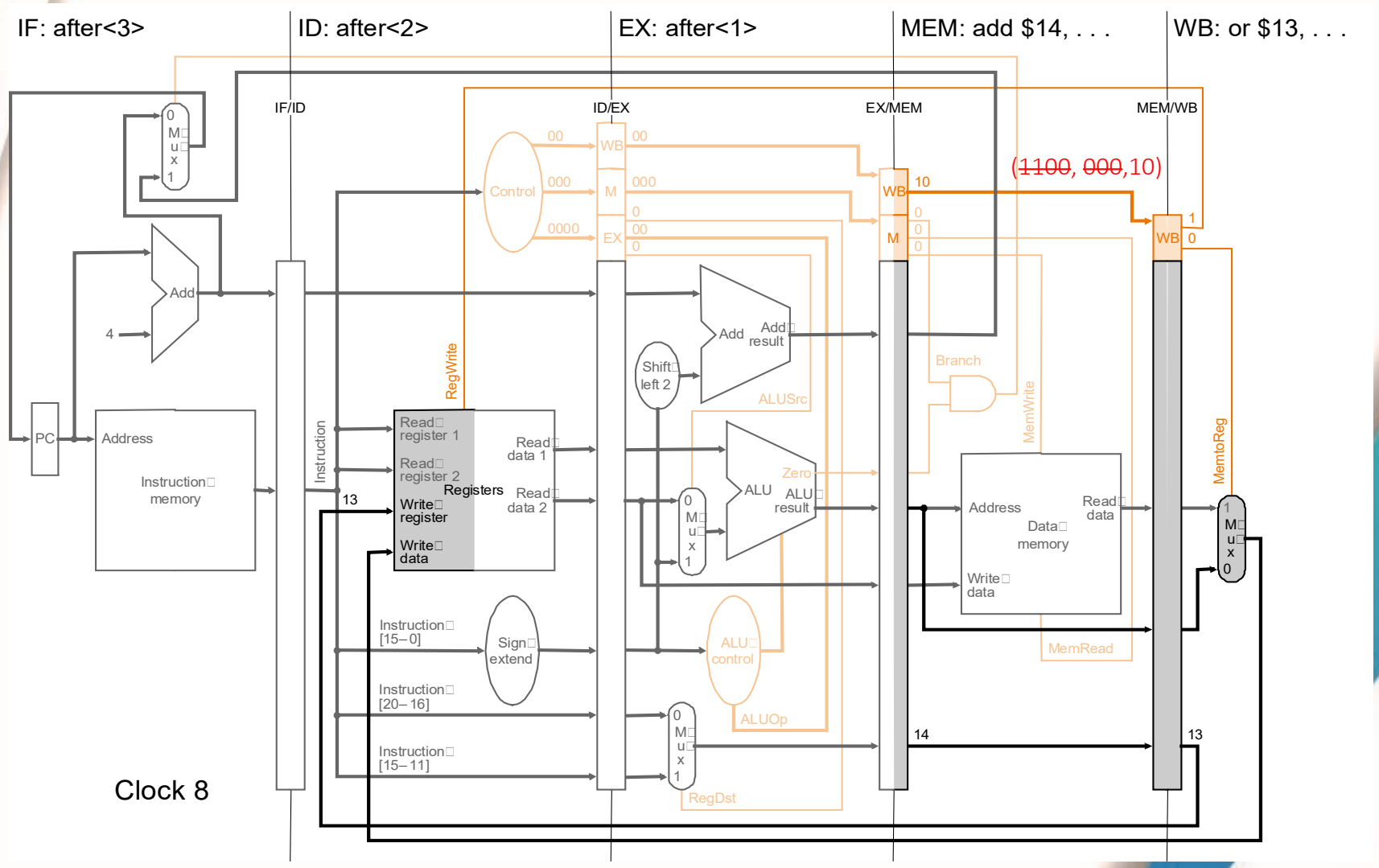
# Cycle 6


IF: after<1>
ID: add $14, $8, $9
EX: or $13, . . .
MEM: and $12, . . .
WB: sub $11, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100,000,10)
(1100, 000,10)
(1100, 000,10)
Control
WB
M
EX
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
ALU control
ALUOp
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
RegDst
Branch
MemWrite
MemRead
Address
Data memory
Read data
Write data
MemtoReg
Clock 6

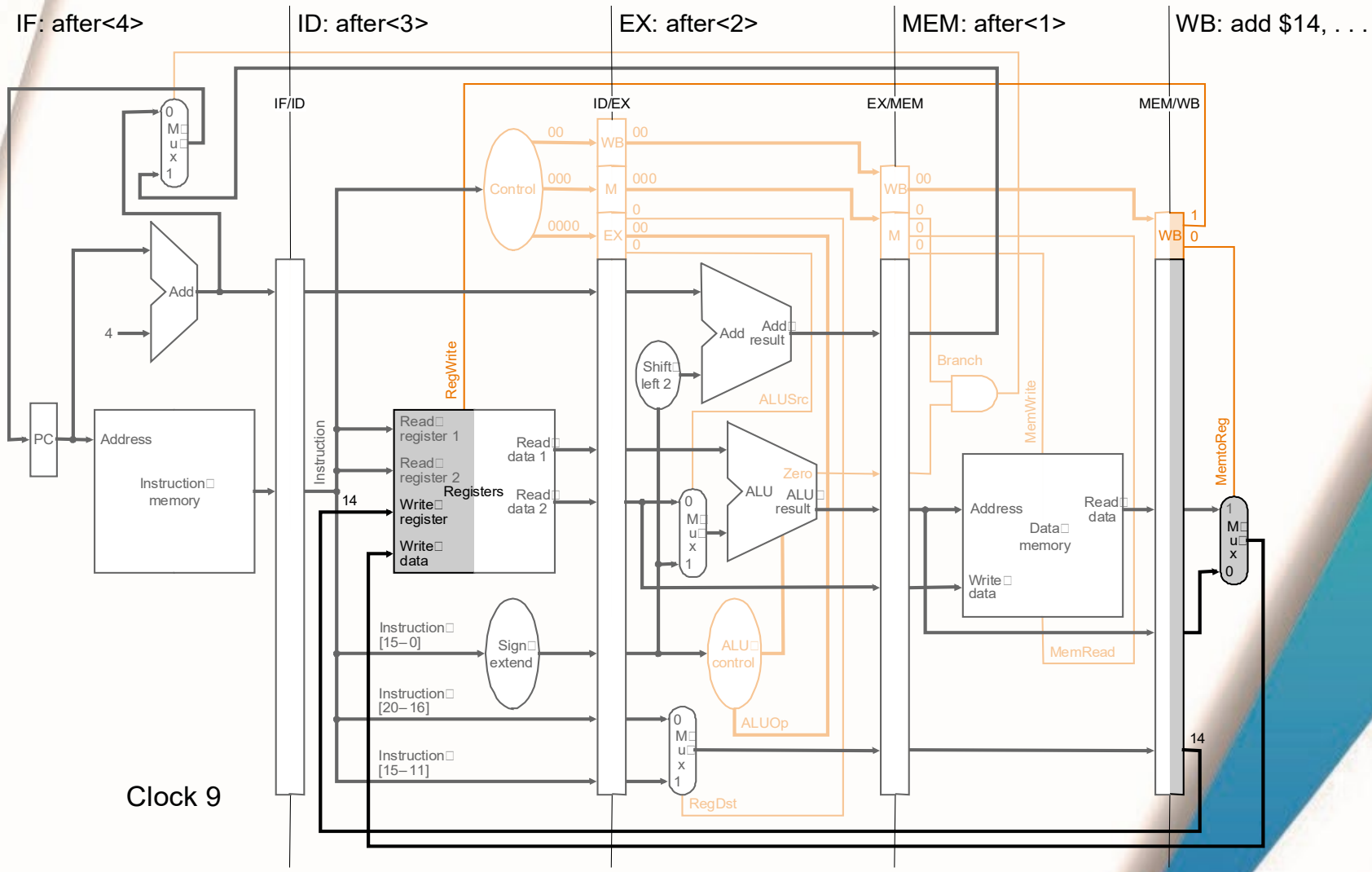
# Cycle 7


IF: after<2>
ID: after<1>
EX: add $14, . . .
MEM: or $13, . . .
WB: and $12, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100, 000,10)
(1100, 000,10)
Control
00
000
0000
WB
M
EX
10
000
1
10
0
WB
M
10
0
0
0
WB
1
0
Add
4
Add
Add result
Shift left 2
ALUSrc
RegWrite
Branch
MemWrite
MemtoReg
PC
Address
Instruction memory
Instruction
12
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$8
$9
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
MemRead
1 M u x 0
Instruction [15–0]
Sign extend
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
0 M u x 1
14
RegDst
13
12
Clock 7

# Cycle 8


IF: after<3>
ID: after<2>
EX: after<1>
MEM: add $14, . . .
WB: or $13, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
(1100, 000,10)
Clock 8

# Cycle 9


IF: after<4>
ID: after<3>
EX: after<2>
MEM: after<1>
WB: add $14, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
14
Control
WB
M
EX
00
000
0000
Add
Add result
Shift left 2
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
ALU control
ALUOp
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
RegDst
Clock 9

# Short Summary

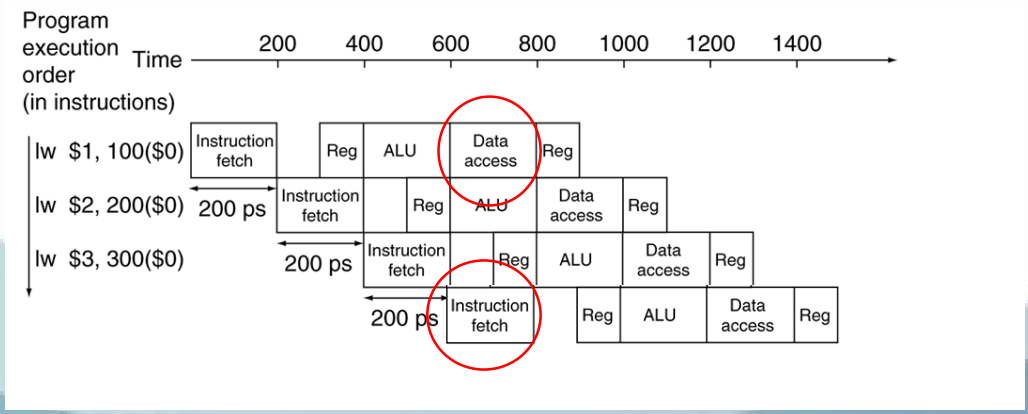
- Pipelining features
  - Each stage uses separate resources
- Pipelining
  - Executes multiple instructions at the same time
  - Set the clock cycle to the longest stage
  - How to detect hazards and resolve them
- MIPS facilitates pipelining due to
  - Only three instruction types and the instructions have the same length
  - Operating memory with only loads and stores

# Hazards

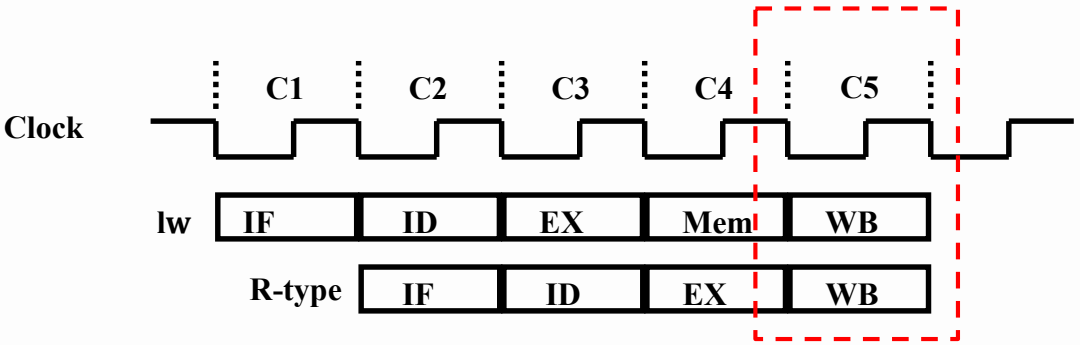
- Situations that prevent starting the next instruction in the next cycle
- Three Hazards: Structure, data, and control hazards
- Structure hazards
  - A required resource is busy
  - Competing the same hardware resource in one cycle
- Data hazard
  - Need to wait for data outputted from previous instruction
  - EX(r-type、lw)、ID (beq) stage
- Control hazard
  - Taken branch decision from the previous instruction needs to void the current instruction
- Hazards can be resolved by waiting (stall)
  1. Hazard detection
  2. Resolving hazards by waiting or other mechanisms.

# Structure Hazard

- Resource conflict
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches



# Pipelining R-type and load

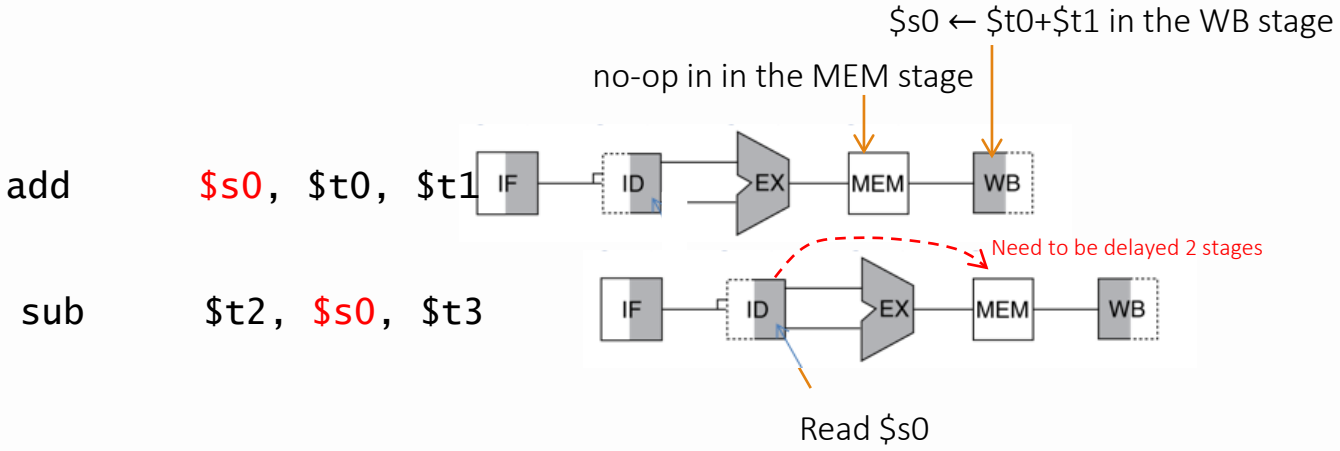



- If R-type takes four cycles and lw taks five cycles, we then have a *structural hazard*:
  - lw and R-type instruction both write results into the register file on the same stage but the register only has one write port.

Ref: Prof. T.-T. Huang, NTHU, CS4100 course slides, Pipelining

# Data Hazards

- Data Hazard
  - Attempt to use data before it is ready
  - The pipeline must be stalled for one step, waiting for another to complete



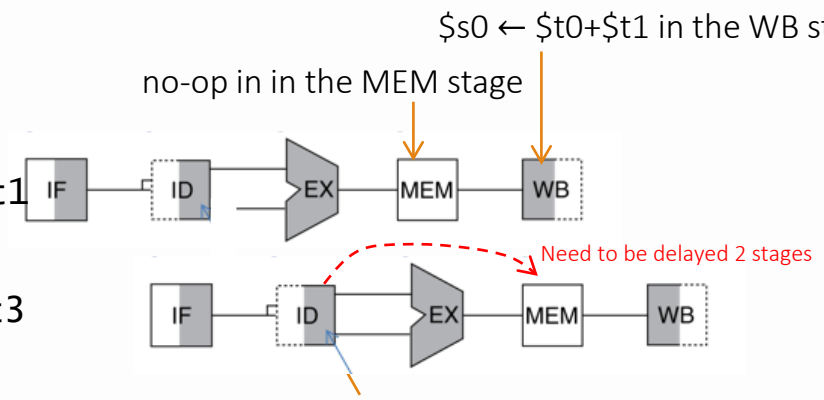
# Different Data Hazards

- RAW (read after write):
  The latter instruction needs to read the register after the former instruction writes it
- WAR (write after read):
  The latter instruction needs to write the register after the former instruction reads it
  - Won't happen in MIPS (5 stages)
- WAW (write after write):
  The latter instruction needs to write the register after the former instruction writes it
  - Won't happen in MIPS (5 stages)

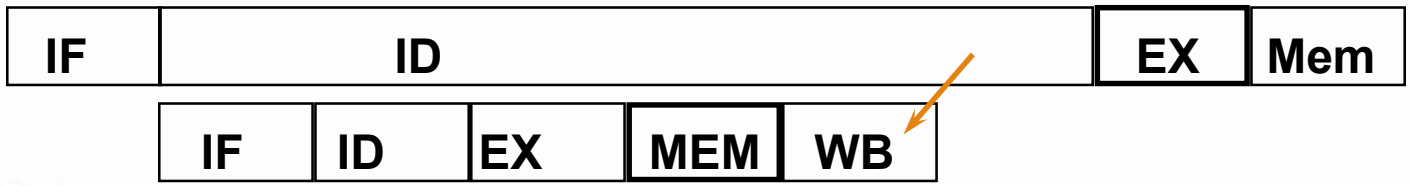
# Data Hazards

RAW

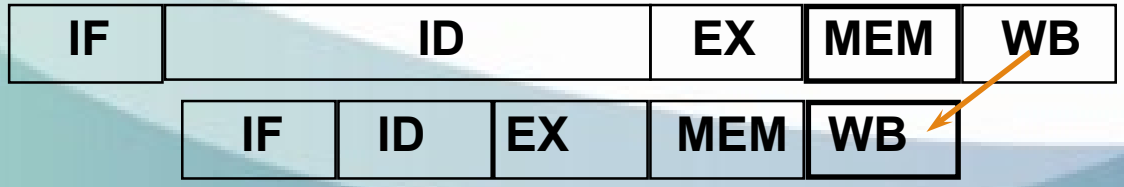
```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```




WAR




WAW



# Resolve Data Hazard

- Solutions
  - Compiler inserts NOP (software solution)
  - Forward (hardware solution)
  - Stall (hardware solution)
- Inserting NOPs
  - Compiler inserts no operation.

```
add   $s0,$t0,$t1
sub   $t2,$s0,$t3
```



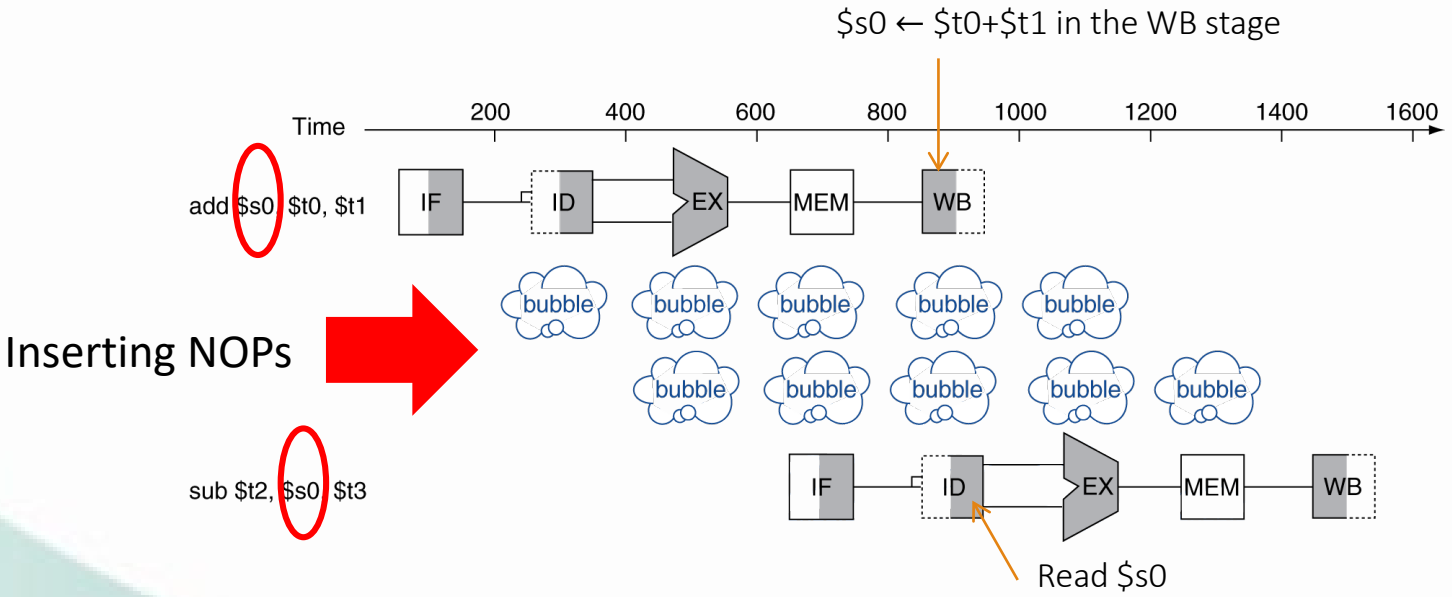
```
add   $1,  $2, $3
Nop
Nop
sub   $4,  $1, $5
```

Drawback: low efficiency

- Forward
  - Internal forwarding from pipeline registers
- Stall
  - Stall instruction in IF and ID: PC and IF/ID remain changing => the stages re-execute the instructions
  - What to move into EX: changing EX, MEM, WB control fields of ID/EX pipeline register to 0
    - **as control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers or memories are written**

# Data Hazards

Inserting NOPs - Compiler inserts no operation.



# Forwarding

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

(sub) Need to access $s0 in ID stage

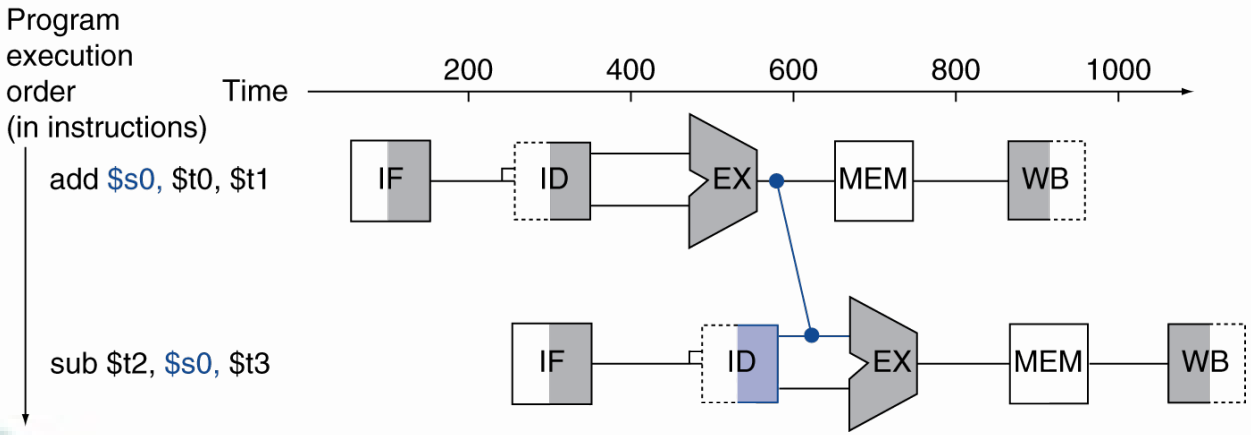
| | Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|---|---|---|---|---|---|
| (IF) | Instruction fetch | IR = Memory[PC] | | | |
| | | PC = PC + 4 | | | |
| (ID) | Instruction | A = Reg [IR[25-21]] | | | |
| | decode/register fetch | B = Reg [IR[20-16]] | | | |
| (EXE) | Execution, address | ALUOut = A op B | ALUOut = A + sign-extend | if (A ==B) then | PC = PC [31-28] II |
| | computation, branch/ | | (IR[15-0]) | PC = PC+ | (IR[25-0]<<2) |
| | jump completion | | | sign-ext(IR[15-0])<<2 | |
| (MEM) | Memory access or R-type | Reg [IR[15-11]] = | Load: MDR = Memory[ALUOut] | | |
| | completion | ALUOut | or | | |
| | | (WB) | Store: Memory [ALUOut] = B | | |
| (WB) | Memory read completion | | Load: Reg[IR[20-16]] = MDR | | |
| | | | | | |

(add) $s0 is calculated in EX stage

(add) save the result in $s0 in WB stage

# Forwarding (Bypassing)

- Use the result immediately when computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



# Hardware Implementation for Resolving Data Hazards

- Consider this sequence:

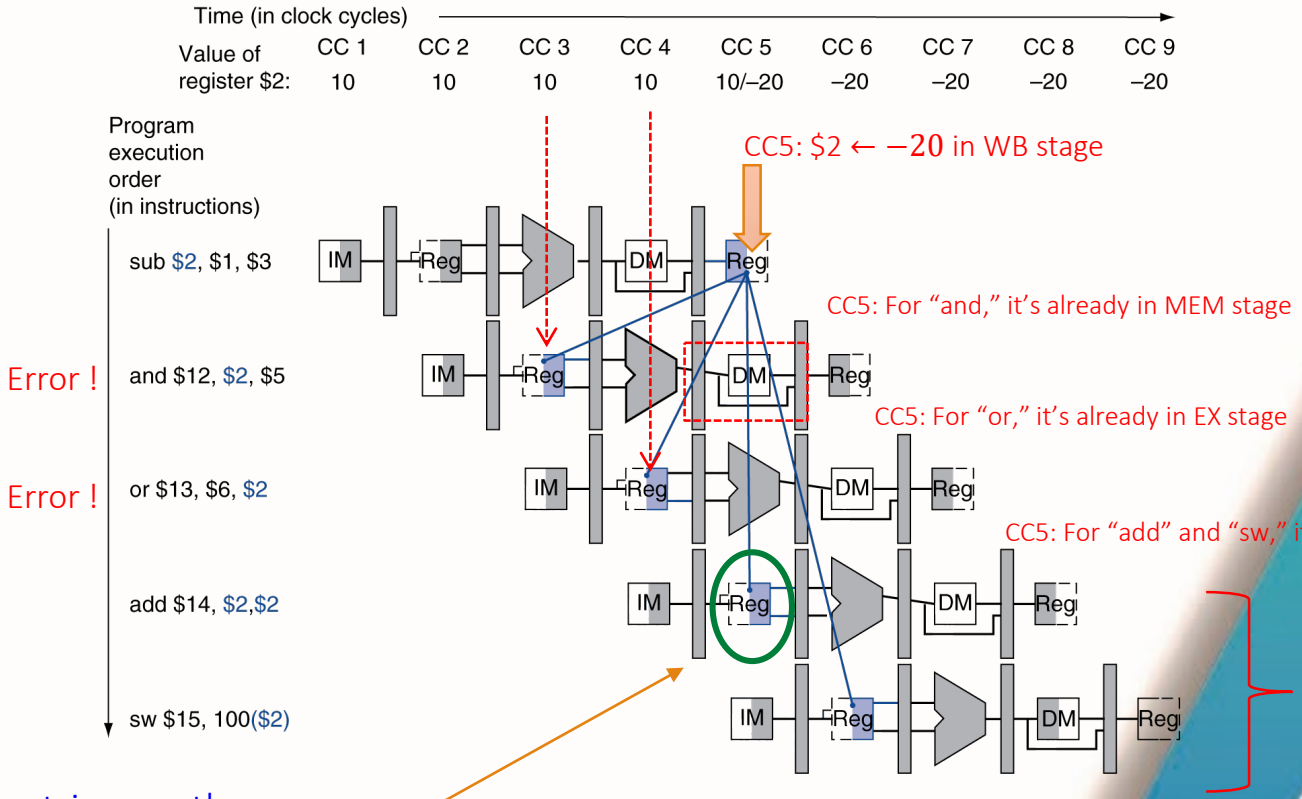
```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

Data dependency: $2

# Example

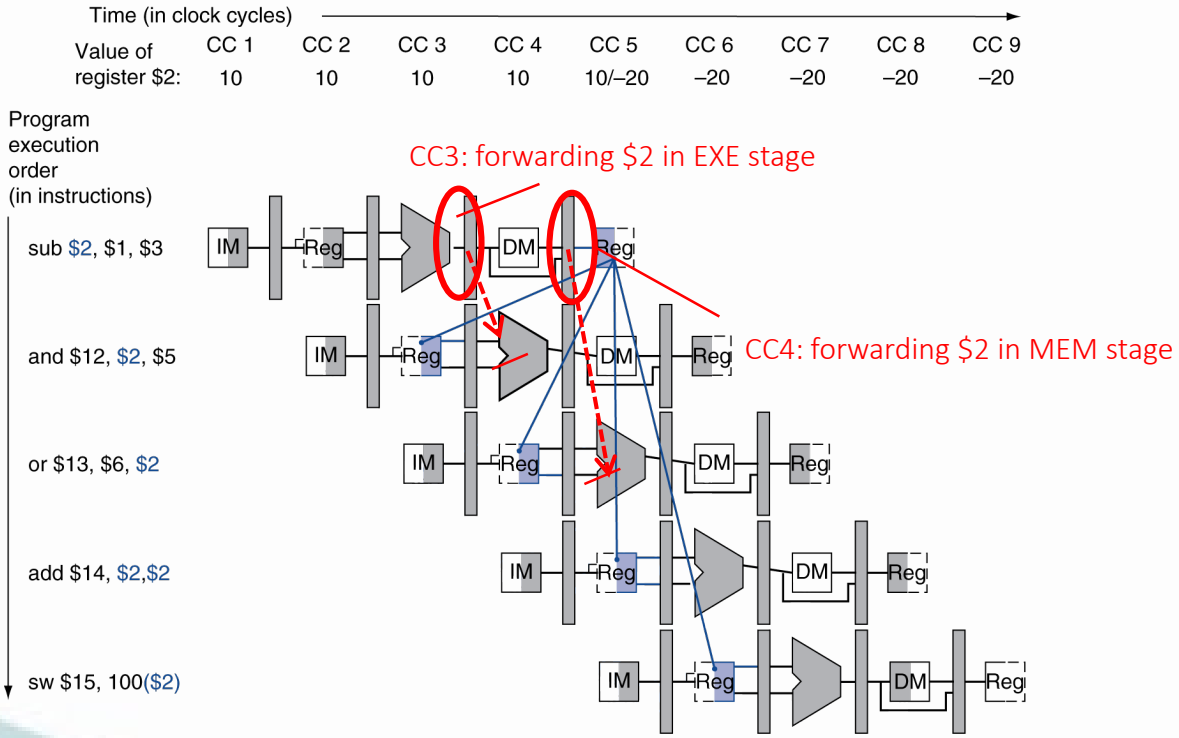
Consider this sequence:

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

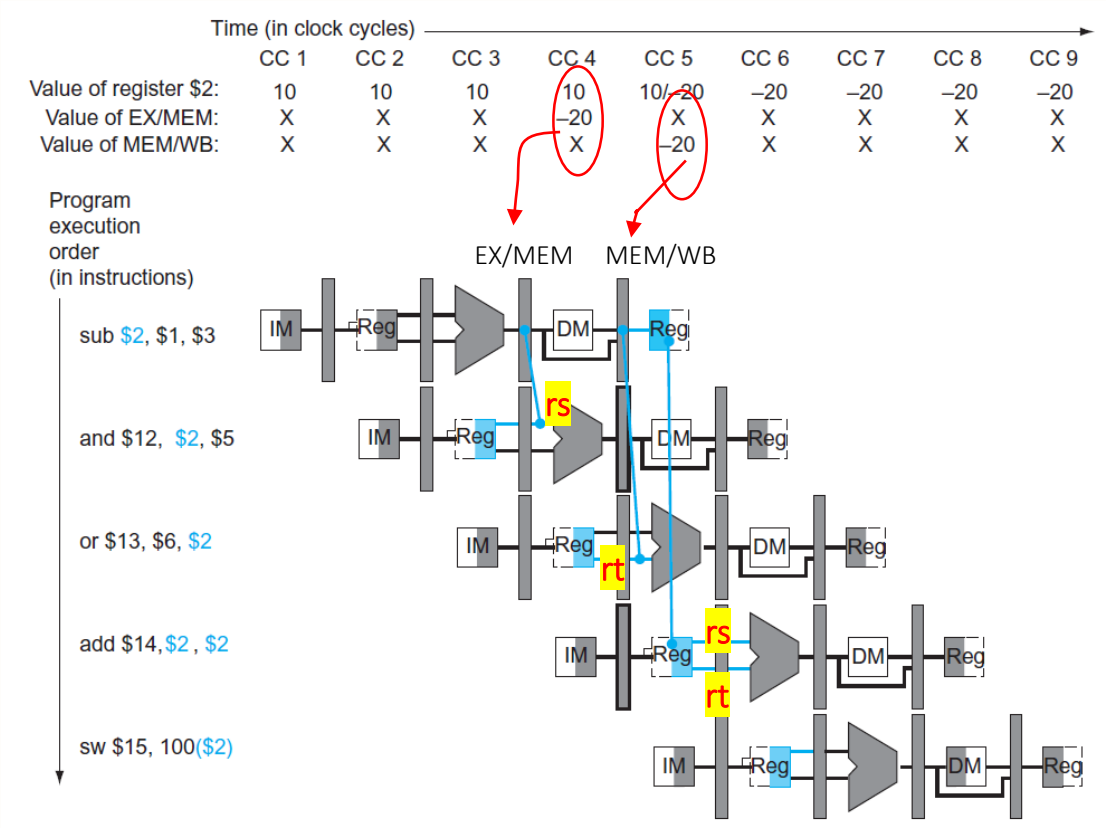



What the clock edge triggers, the data will be written into $s2 in WB stage of “sub”. Then $s2 is read in ID stage of “add”.

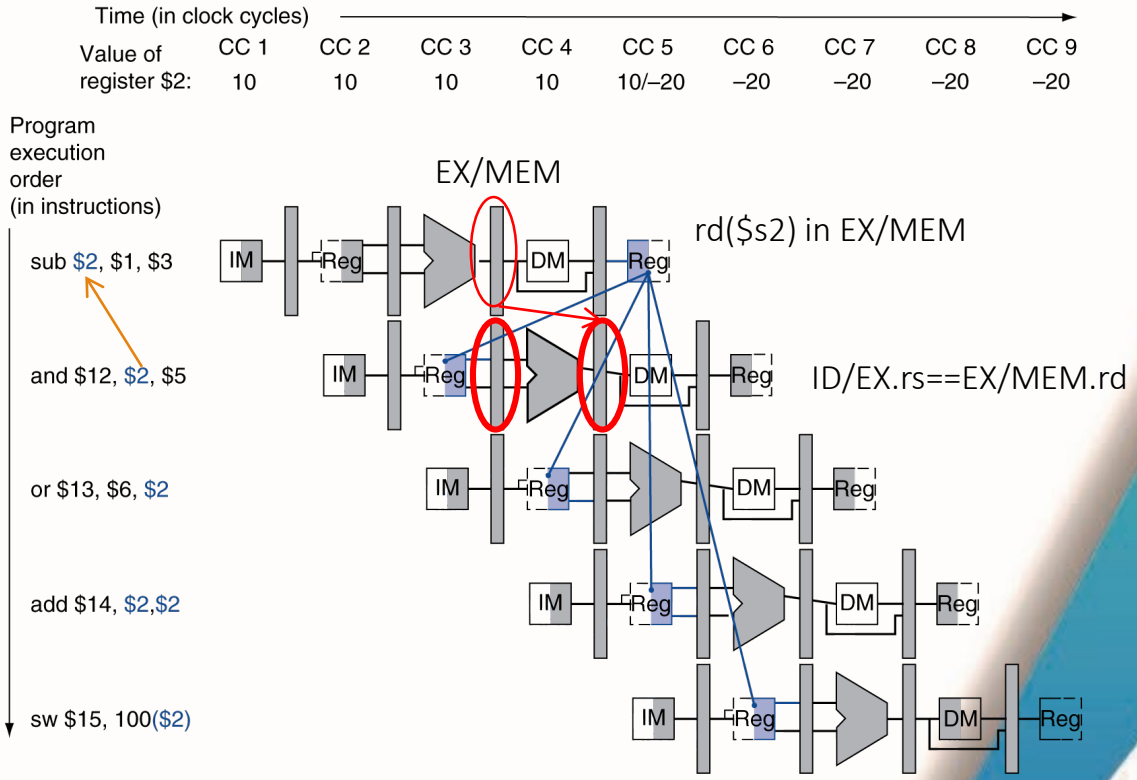
# Resolving Data Hazard by Forwarding

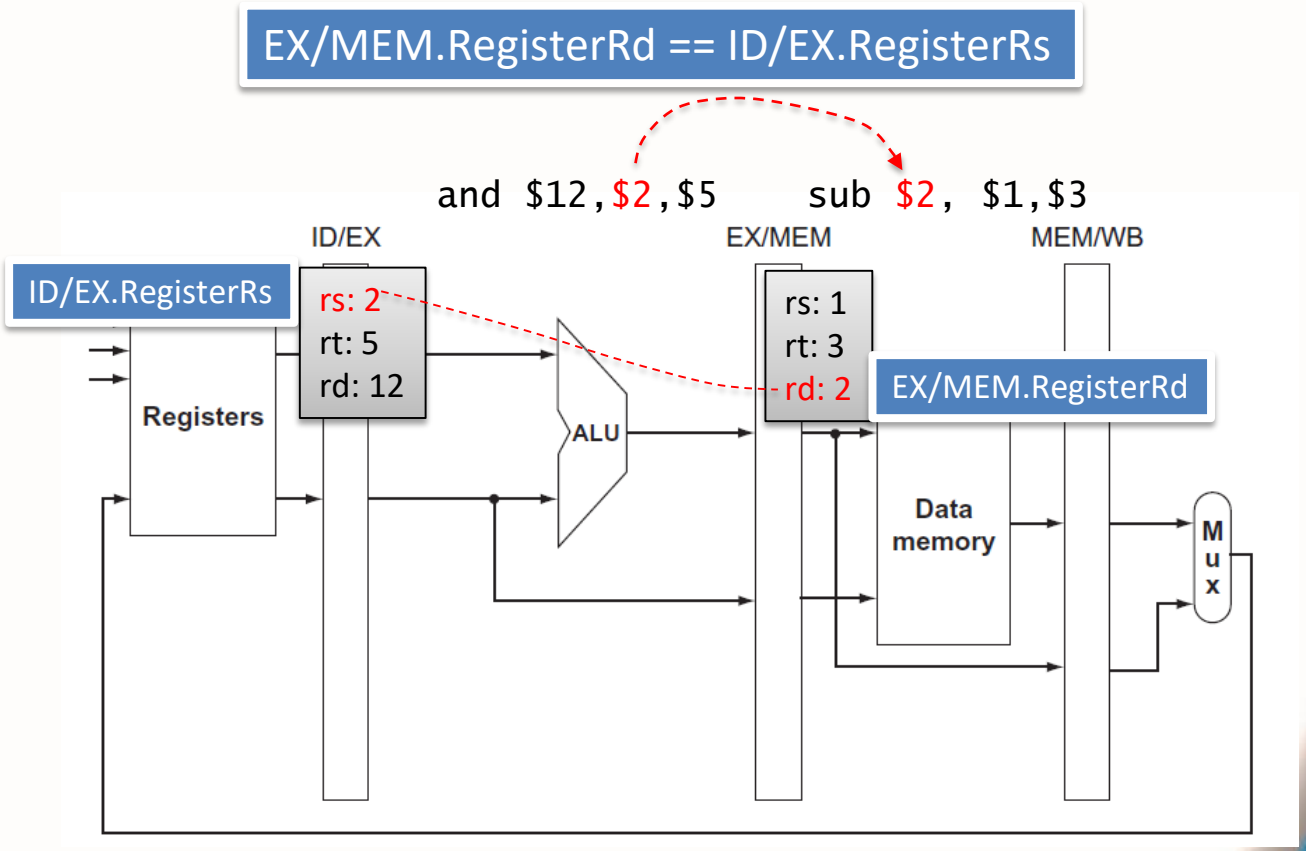


For sub

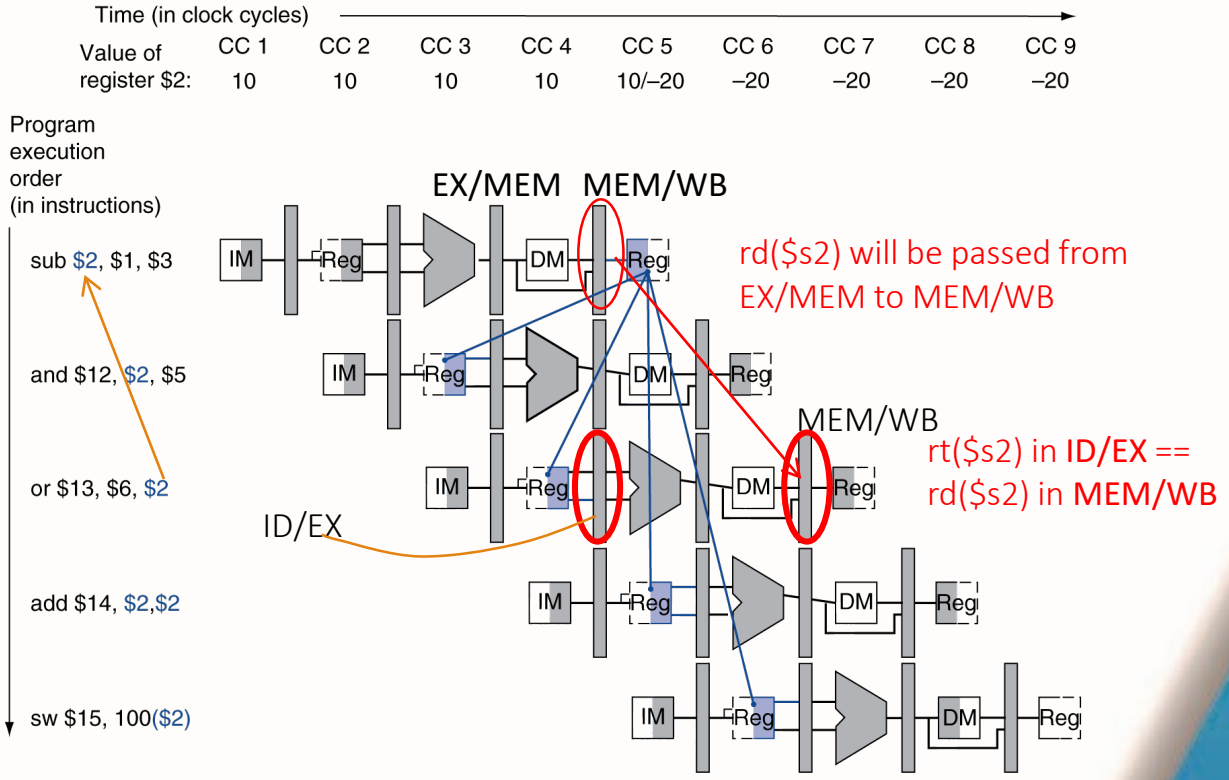

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Value of EX/MEM: X X X –20 X X X X X
Value of MEM/WB: X X X X –20 X X X X
Program execution order (in instructions)
EX/MEM
MEM/WB
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14,$2 , $2
sw $15, 100($2)
IM
Reg
DM
rs
rt

# When Data Hazard Occurs - 1

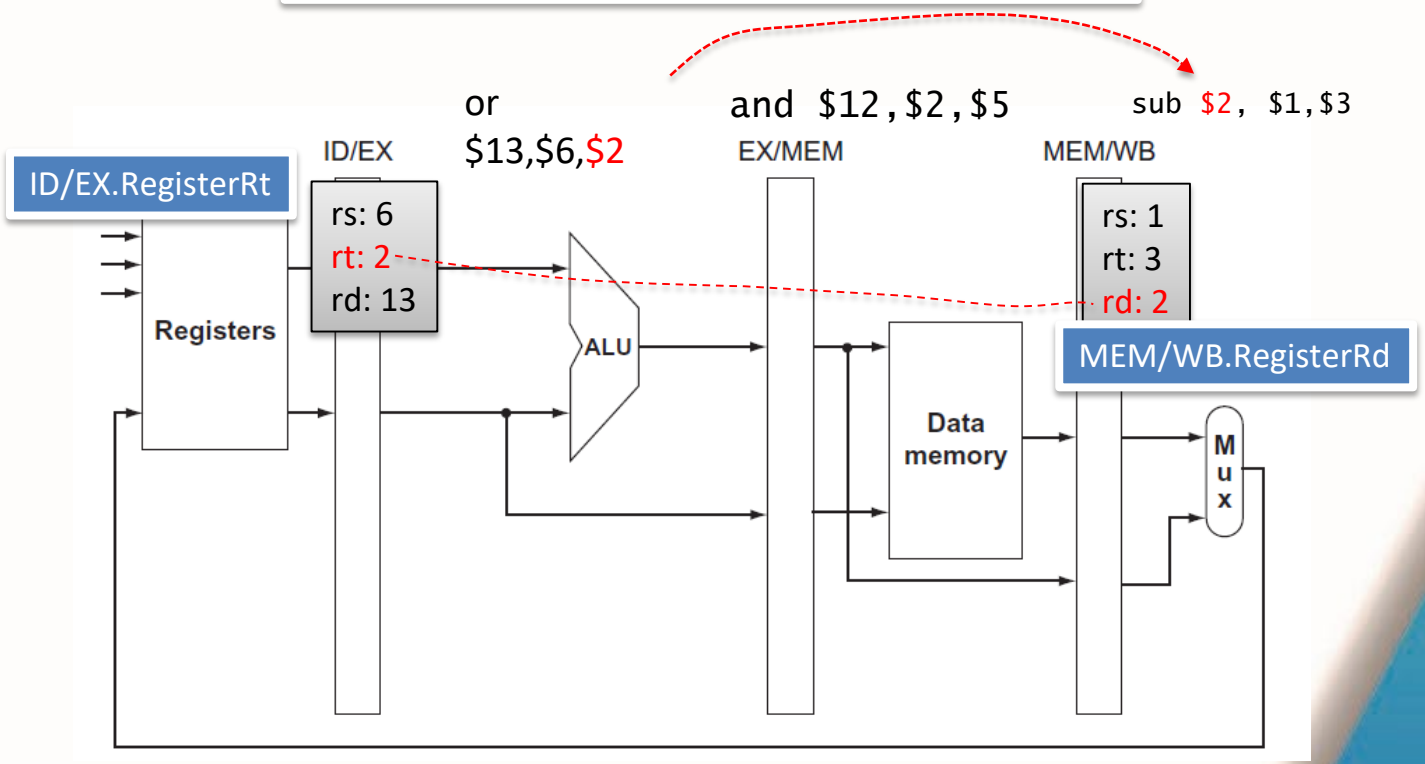



EX/MEM.RegisterRd == ID/EX.RegisterRs
and $12,$2,$5
sub $2, $1,$3
ID/EX
EX/MEM
MEM/WB
ID/EX.RegisterRs
rs: 2
rt: 5
rd: 12
rs: 1
rt: 3
rd: 2
EX/MEM.RegisterRd
Registers
ALU
Data memory
M u x

# When Data Hazard Occurs - 2



MEM/WB.RegisterRd = =ID/EX.RegisterRt



# Detecting the Need to Forward

- Data hazards when

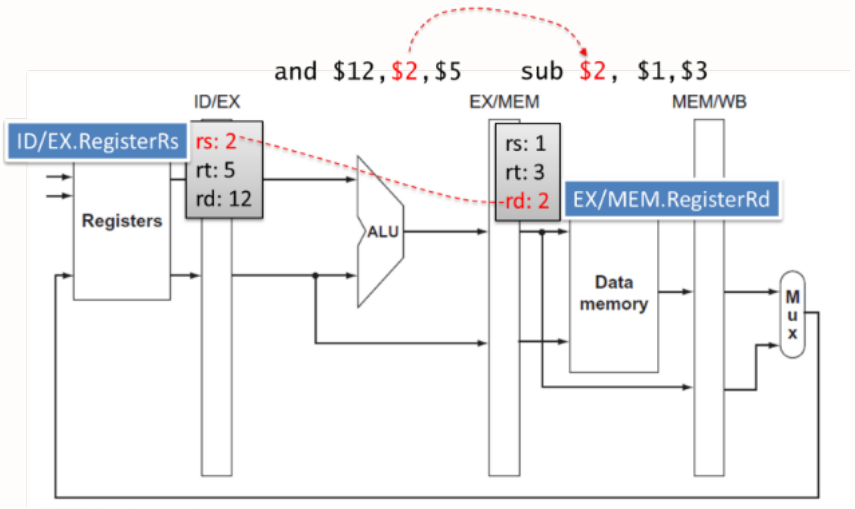
It's Reg Num, not value in Reg

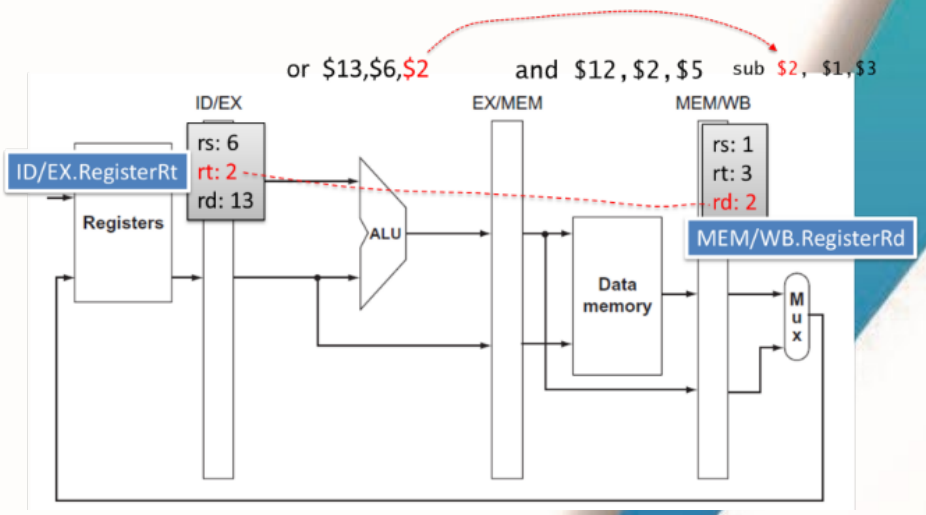
1a. EX/MEM.RegisterRd == ID/EX.RegisterRs
1b. EX/MEM.RegisterRd == ID/EX.RegisterRt

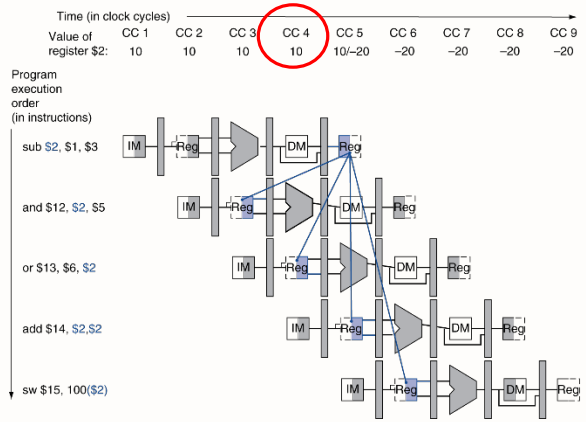
Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd == ID/EX.RegisterRs
2b. MEM/WB.RegisterRd == ID/EX.RegisterRt

Fwd from MEM/WB pipeline reg

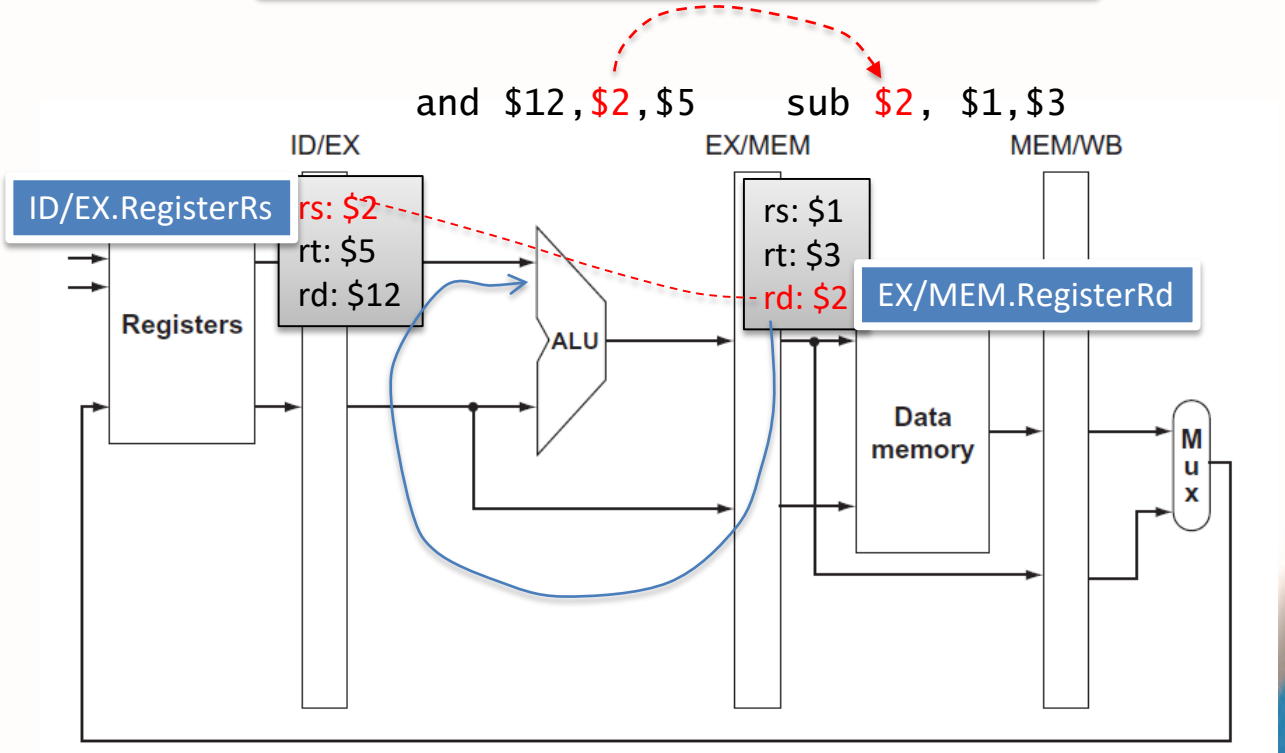


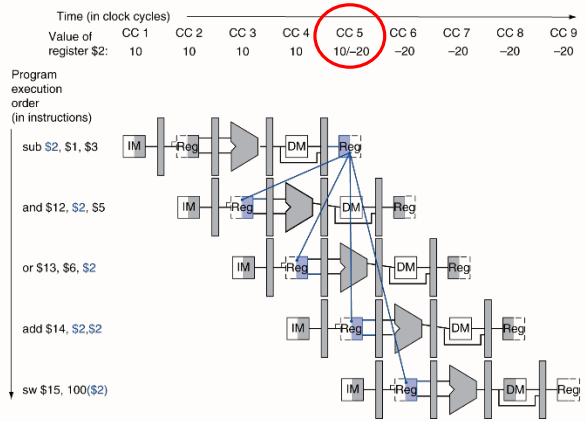






# CC4 Forward

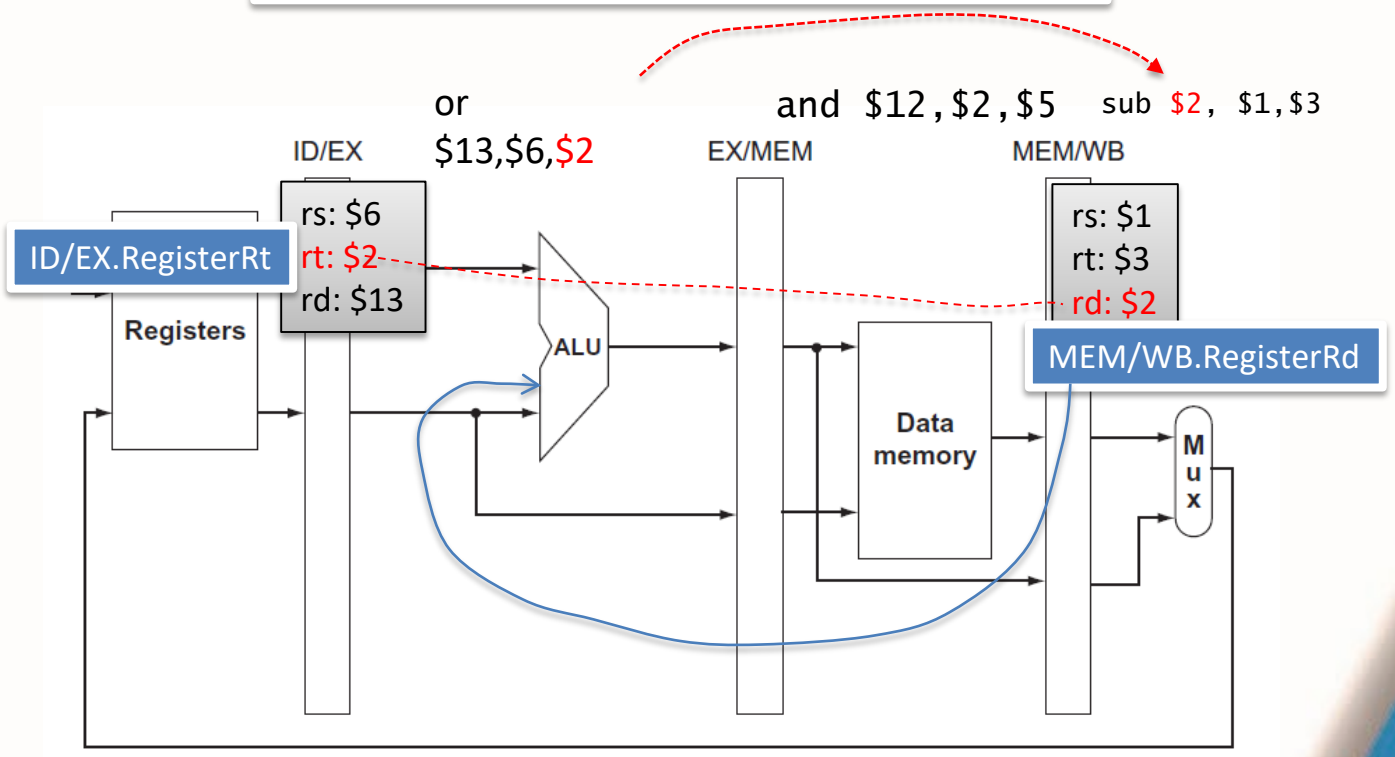
EX/MEM.RegisterRd == ID/EX.RegisterRs



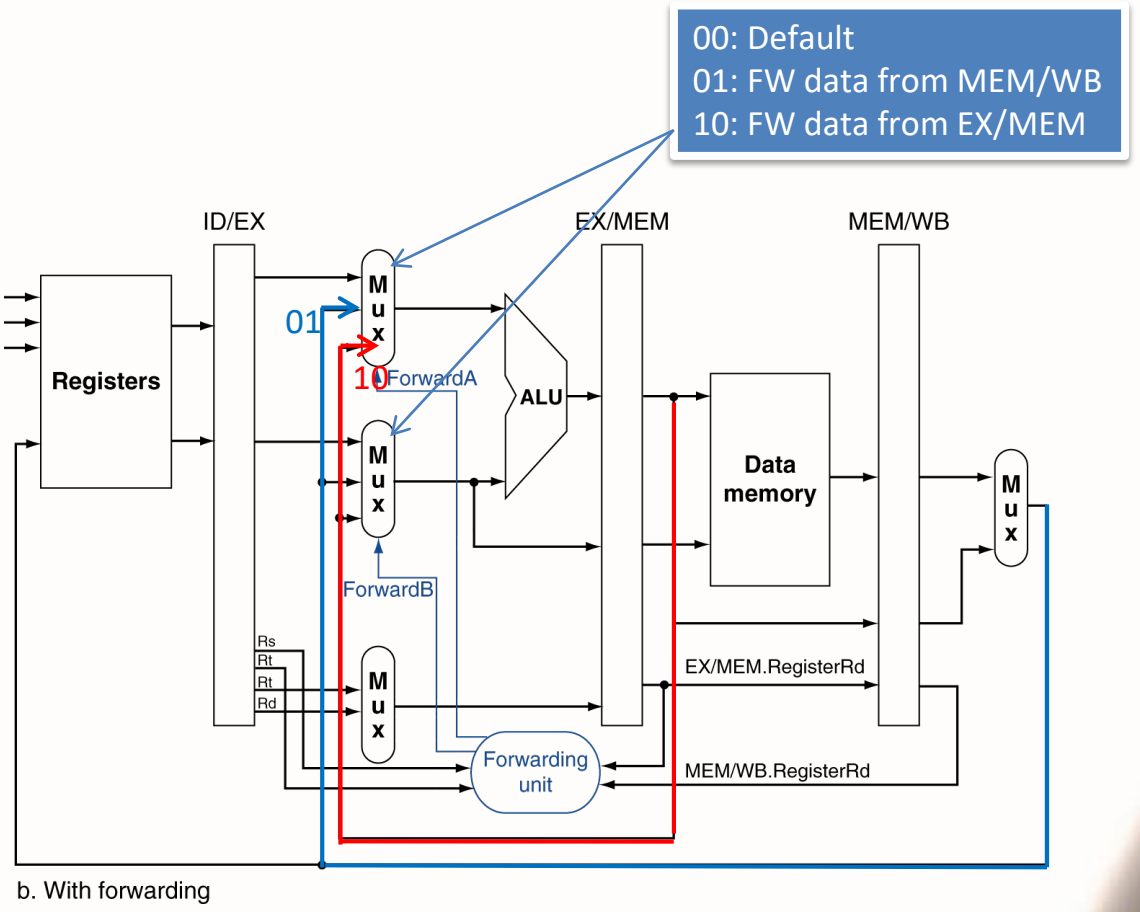



# CC5 Forward

MEM/WB.RegisterRd == ID/EX.RegisterRt



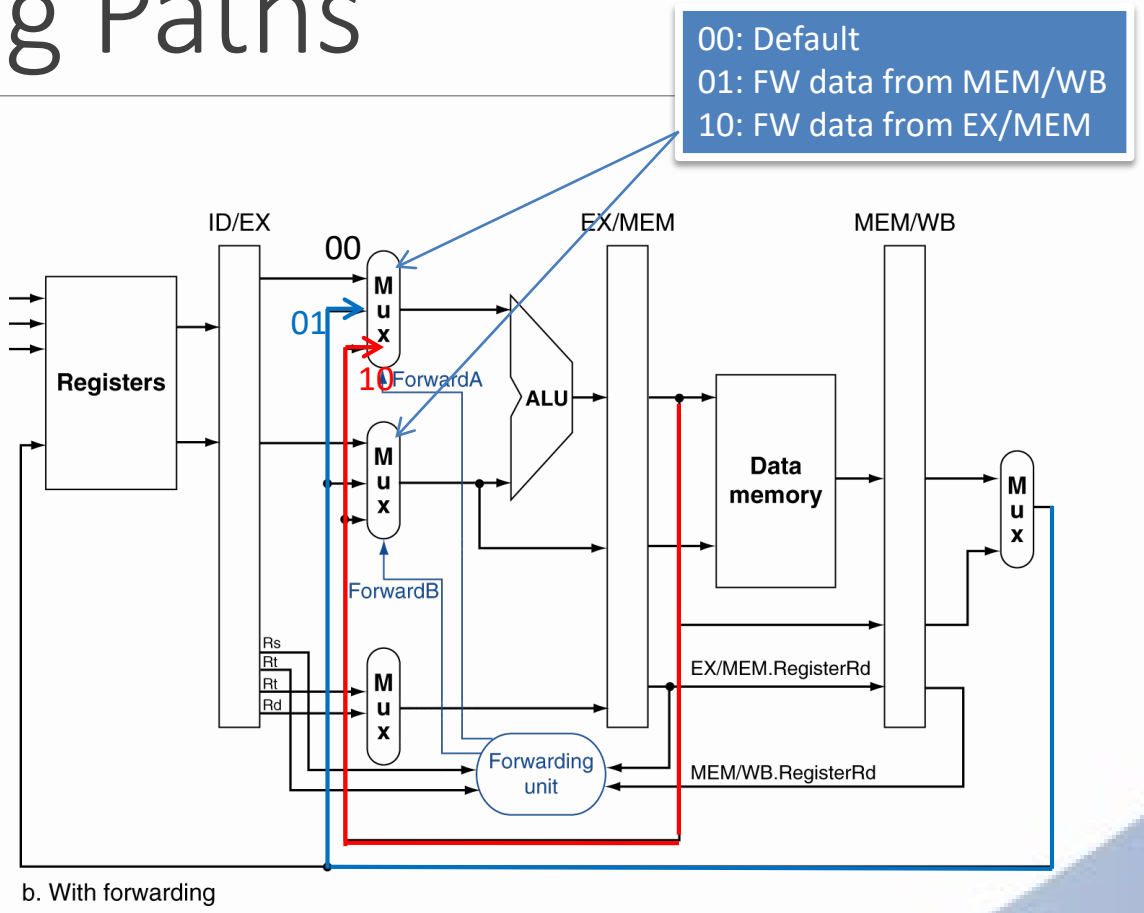
# Forwarding Paths




b. With forwarding

# Forwarding Paths

Mux Select Signals
- ForwardA
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM
- ForwardB
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM



b. With forwarding

# Two Exceptions in Forwarding

- No Forwarding
  - Do not write data into Register
    - EX/MEM.RegWrite==0 or MEM/WB.RegWrite==0
  - If writing data into $zero (forbidden)
    - EX/MEM.RegisterRd == 0,
      MEM/WB.RegisterRd == 0

# Forwarding Conditions

- EX data hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs))

    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRt))

    ForwardB = 10
- MEM data hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRs))

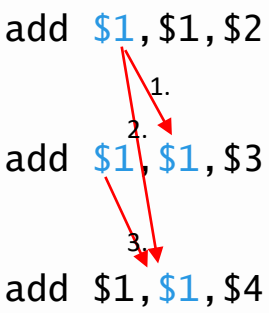
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRt))

    ForwardB = 01

Mux Select Signals
- ForwardA
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM
- ForwardB
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM

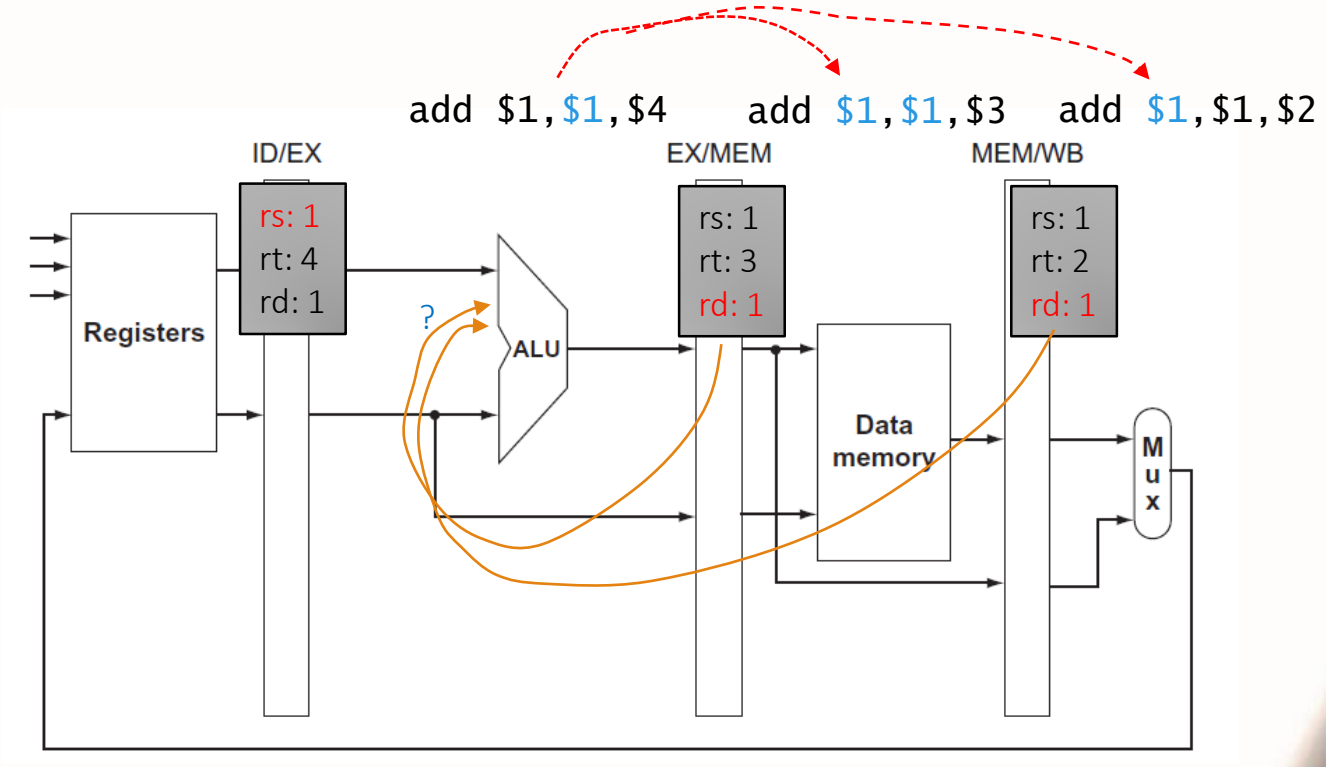
# Double Data Hazard

- Consider the sequence:

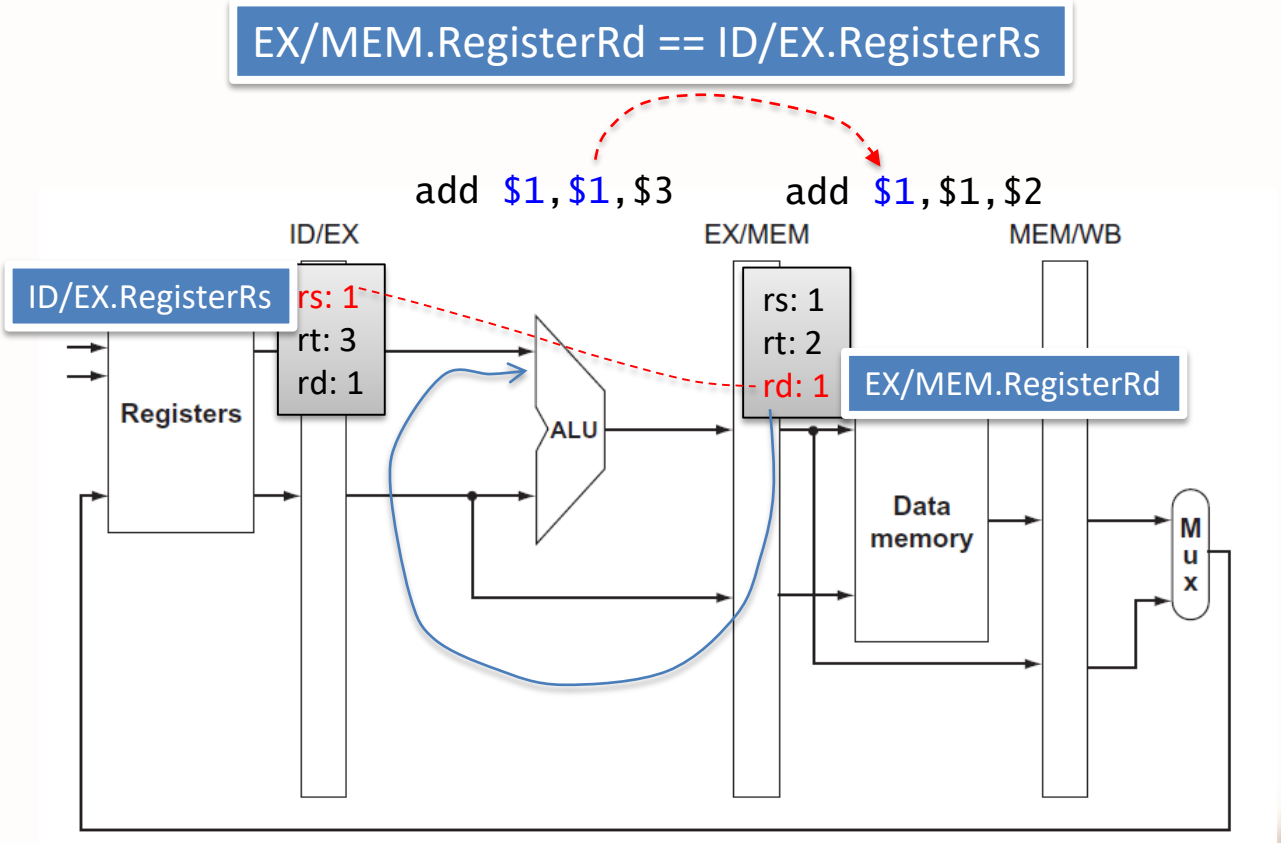



1. EX/MEM.RegisterRd == ID/EX.RegisterRs
2. MEM/WB.RegisterRd == ID/EX.RegisterRs
3. EX/MEM.RegisterRd == ID/EX.RegisterRs

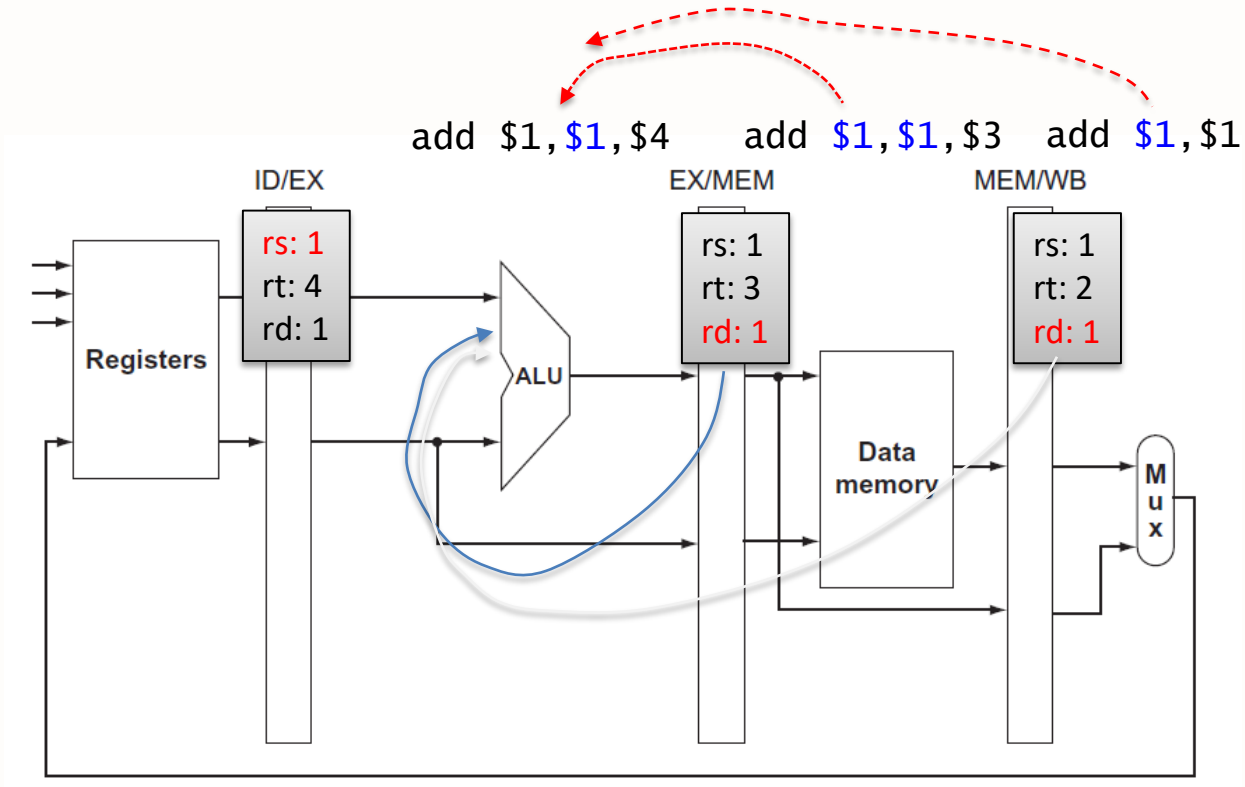
Two data hazard detected:
1. MEM/WB: ForwardA = 01
2. EX/MEM: ForwardA = 10


add $1,$1,$4
add $1,$1,$3
add $1,$1,$2
ID/EX
EX/MEM
MEM/WB
Registers
rs: 1
rt: 4
rd: 1
?
ALU
rs: 1
rt: 3
rd: 1
Data memory
rs: 1
rt: 2
rd: 1
M
u
x

EX/MEM.RegisterRd == ID/EX.RegisterRs



## When a double data hazard occurs，Forward data from EX/MEM (most recent result)



# Revised Forwarding Condition

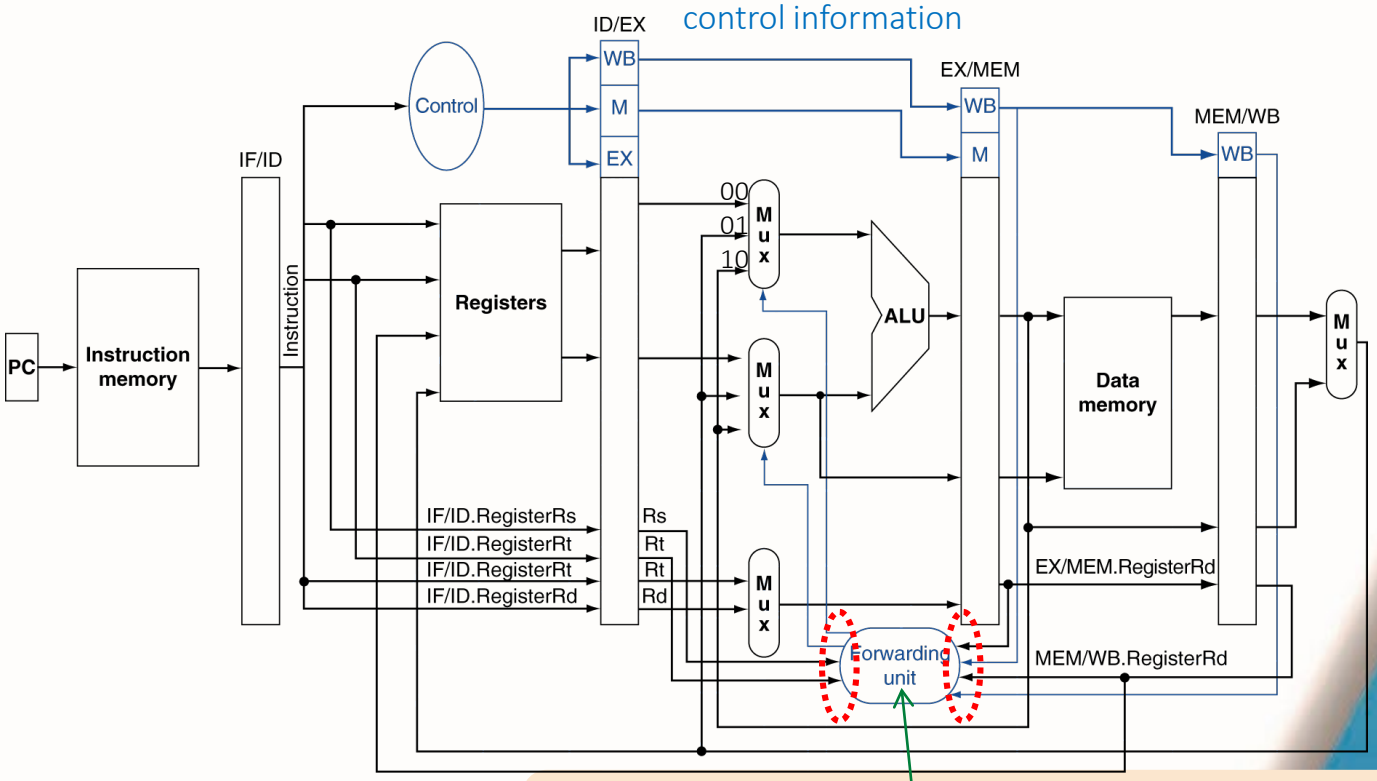
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs))

    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRt))

    ForwardB = 01

Mux Select Signals
- ForwardA
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM
- ForwardB
  - 00: Default
  - 01: FW data from MEM/WB
  - 10: FW data from EX/MEM

# Datapath with Forwarding




Input: **rs, rt (from** ID/EX), rd (from EX/MEM), and rd (from MEM/WB)

**EX/MEM.RegWrite**, and MEM/WB.**RegWrite**

# Load/Store

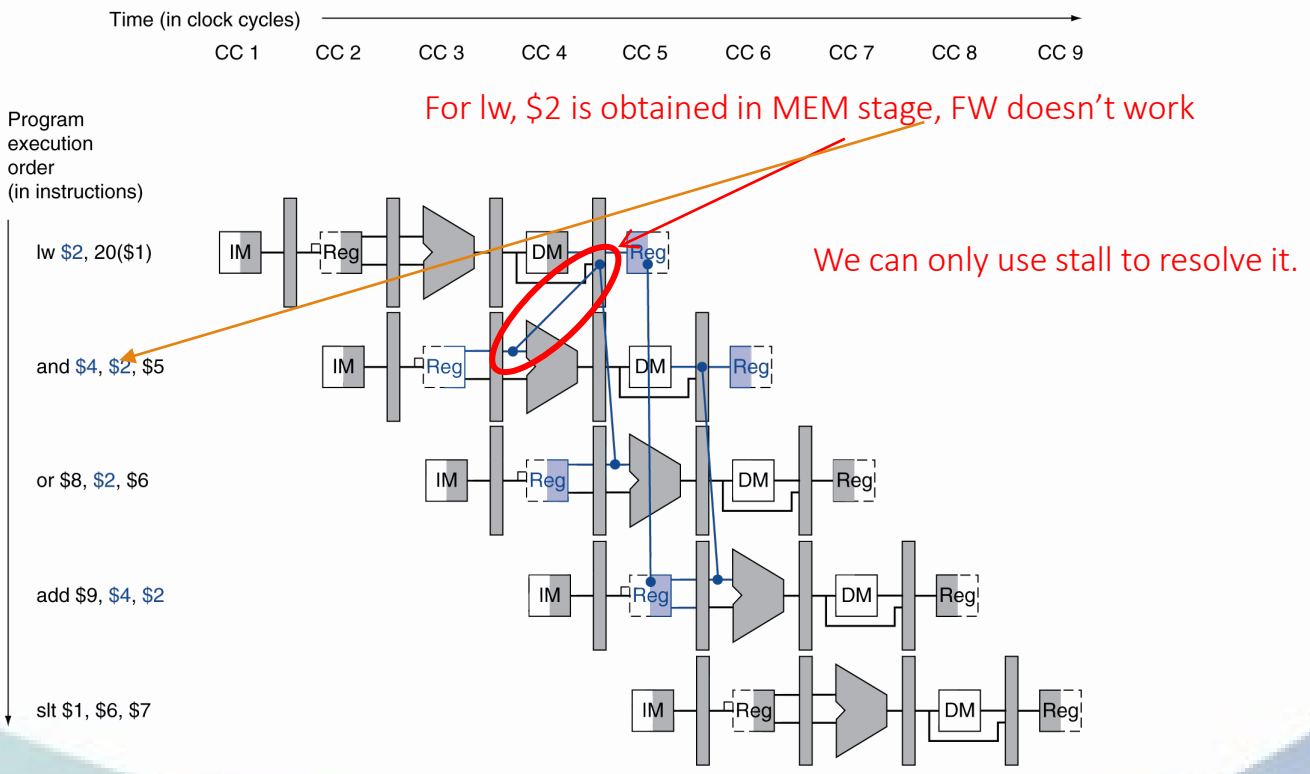
| Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|---|---|---|---|---|
| Instruction fetch | IR = Memory[PC] | | | |
| | PC = PC + 4 | | | |
| Instruction | A = Reg [IR[25-21]] | | | |
| decode/register fetch | B = Reg [IR[20-16]] | | | |
| Execution, address | ALUOut = A op B | ALUOut = A + sign-extend | if (A ==B) then | PC = PC [31-28] II |
| computation, branch/ | | (IR[15-0]) | PC = PC+ | (IR[25-0]<<2) |
| jump completion | | | sign-ext(IR[15-0])<<2 | |
| Memory access or R-type | Reg [IR[15-11]] = | Load: MDR = Memory[ALUOut] | | |
| completion | ALUOut | or | | |
| | (WB) | Store: Memory [ALUOut] = B | | |
| Memory read completion | | Load: Reg[IR[20-16]] = MDR | | |
| | | | | |

lw:

data read from memory **loaded** into the register in WB stage

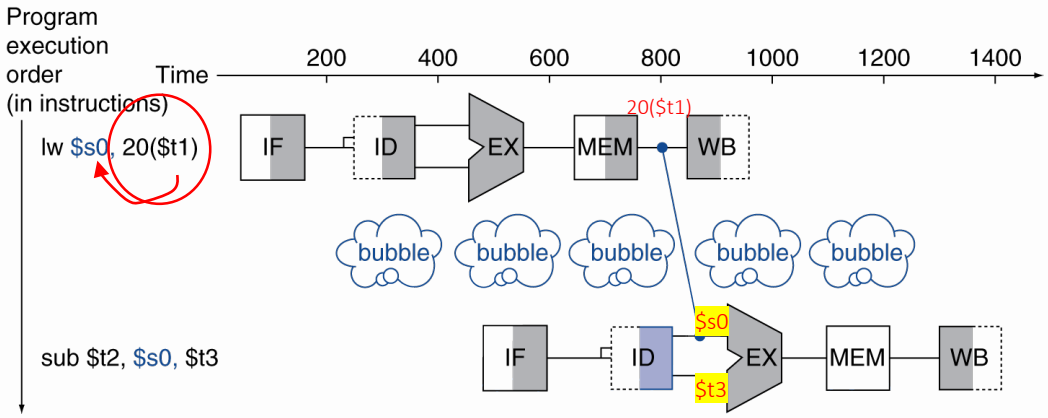
(it can access memory in MEM stage)

# Load-Use Data Hazard



# Load-Use Data Hazards

- It cannot always avoid stalls by forwarding
  - It still incurs a stall even using forwarding
    - Can only access the data as early as in MEM stage

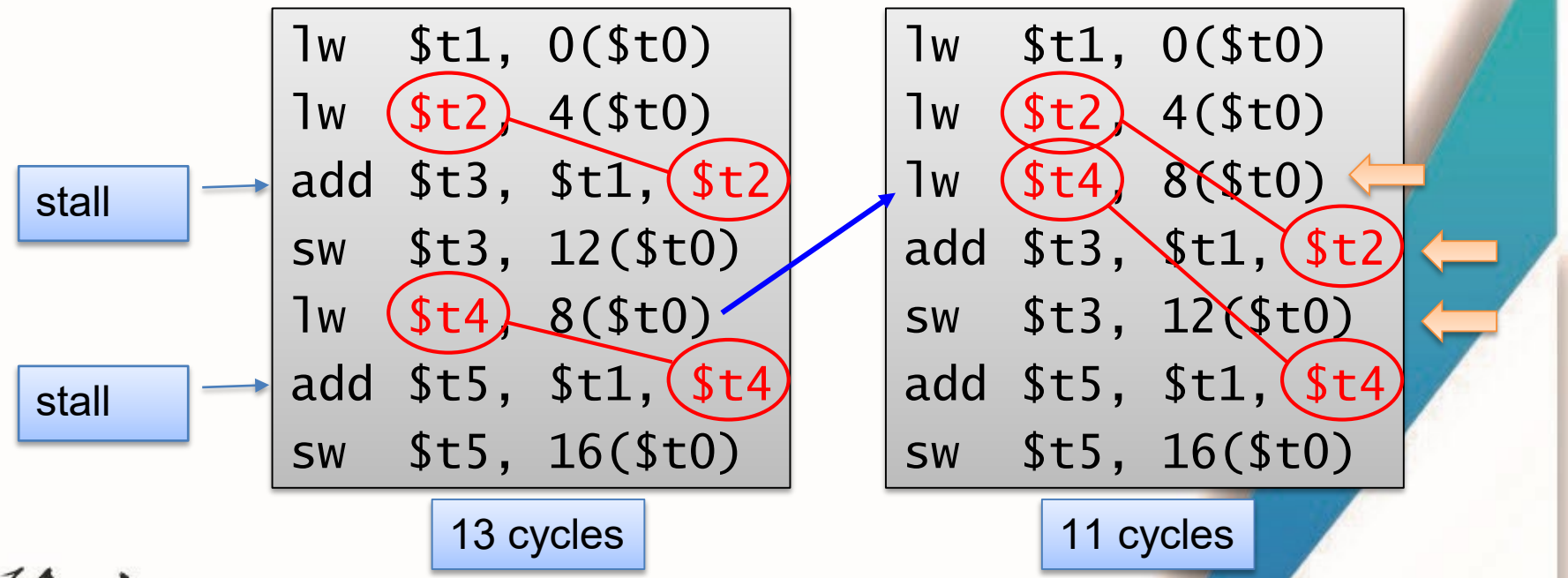



Compared to R-type instructions?

# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of the loaded result in the next instruction
- C code : `a = b + e;`
  `c = b + f;`

After reordering the code, the results of two lw's would be used one instruction after the lw instruction



```
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)
```

| Instr | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 | CC11 | CC12 | CC13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | IF | ID | EXE | MEM | WB | | | | | | | | |
| lw | | IF | ID | EXE | MEM | WB | | | | | | | |
| | | | | | | 4($t0) | | | | | | | |
| add | | | | IF | ID | EXE | MEM | WB | | | | | |
| sw | | | | | IF | ID | EXE | MEM | WB | | | | |
| | | | | | | | | $t1+$t2 | | | | | |
| lw | | | | | | IF | ID | EXE | MEM | WB | | | |
| | | | | | | | | | | 8($t0) | | | |
| add | | | | | | | | IF | ID | EXE | MEM | WB | |
| sw | | | | | | | | | IF | ID | EXE | MEM | WB |

```
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)
```

| Instr | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 | CC11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | IF | ID | EXE | MEM | WB | | | | | | |
| lw | | IF | ID | EXE | MEM | WB | | | | | |
| lw | | | IF | ID | EXE | MEM | WB | | | | |
| add | | | | IF | ID | EXE | MEM | WB | | | |
| sw | | | | | IF | ID | EXE | MEM | WB | | |
| add | | | | | | IF | ID | EXE | MEM | WB | |
| sw | | | | | | | IF | ID | EXE | MEM | WB |

# Example

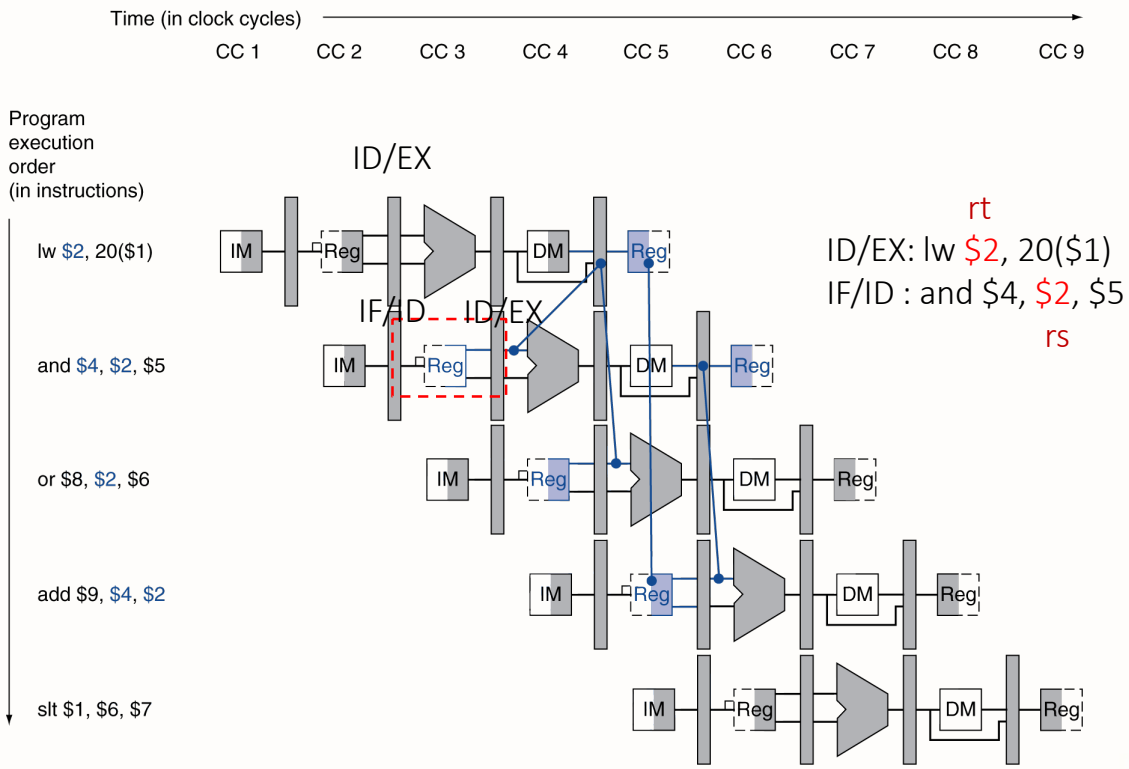
For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

**Check Yourself**

| Sequence 1 | Sequence 2 | Sequence 3 |
|---|---|---|
| `lw $t0,0($t0)`<br>`add $t1,$t0,$t0` | `add $t1,$t0,$t0`<br>`addi $t2,$t0,#5`<br>`addi $t4,$t1,#5` | `addi $t1,$t0,#1`<br>`addi $t2,$t0,#2`<br>`addi $t3,$t0,#2`<br>`addi $t3,$t0,#4`<br>`addi $t5,$t0,#5` |

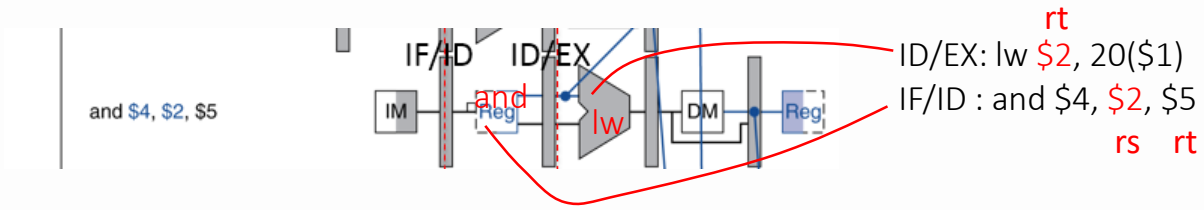
lw: 1 stall even with forwarding
r-type: no stall with forwarding
branch: 1 stall

# Detect Load-Use Hazard in ID stage (since we need a stall anyway, do it as early as it can)



# Load-Use Hazard Detection

- Check in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt

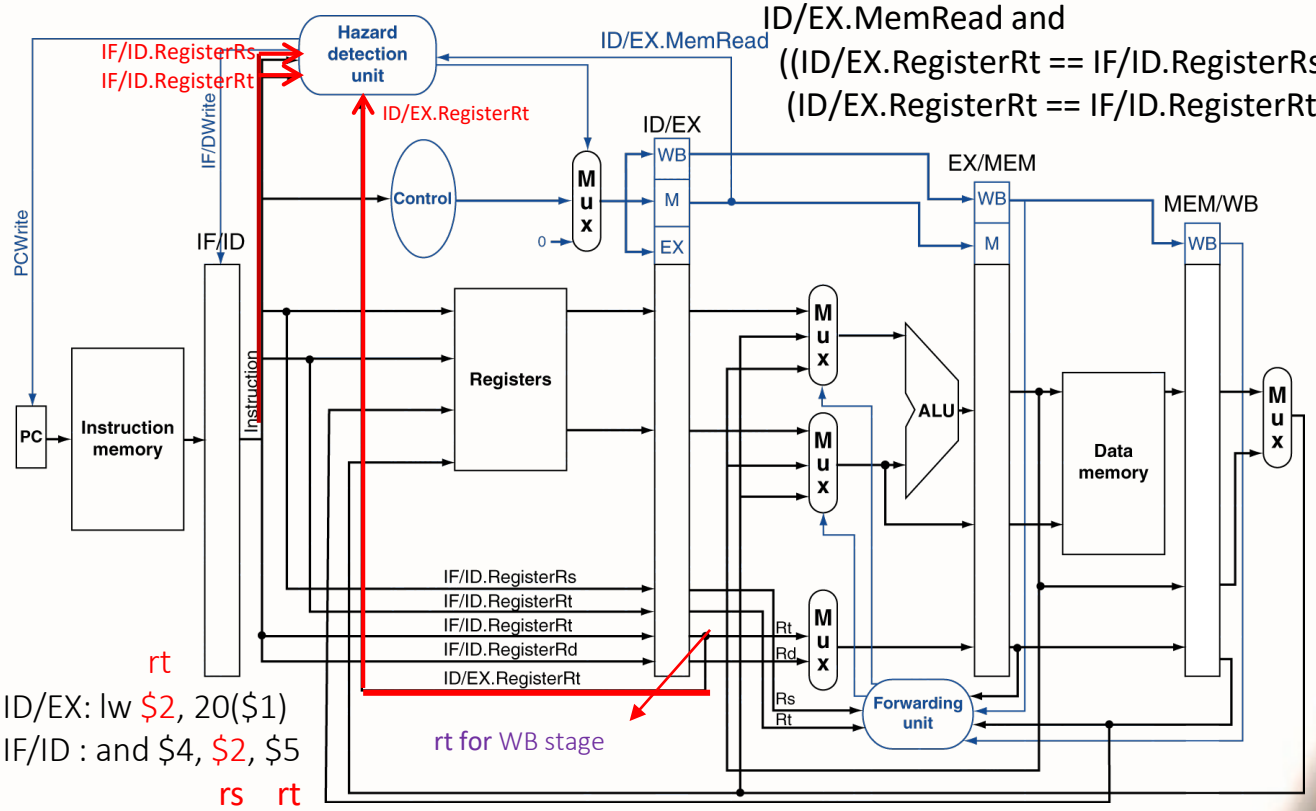



- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt == IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))

    **stall the pipeline**

ID/EX.MemRead==1 -> load instruction

# Datapath with Hazard Detection

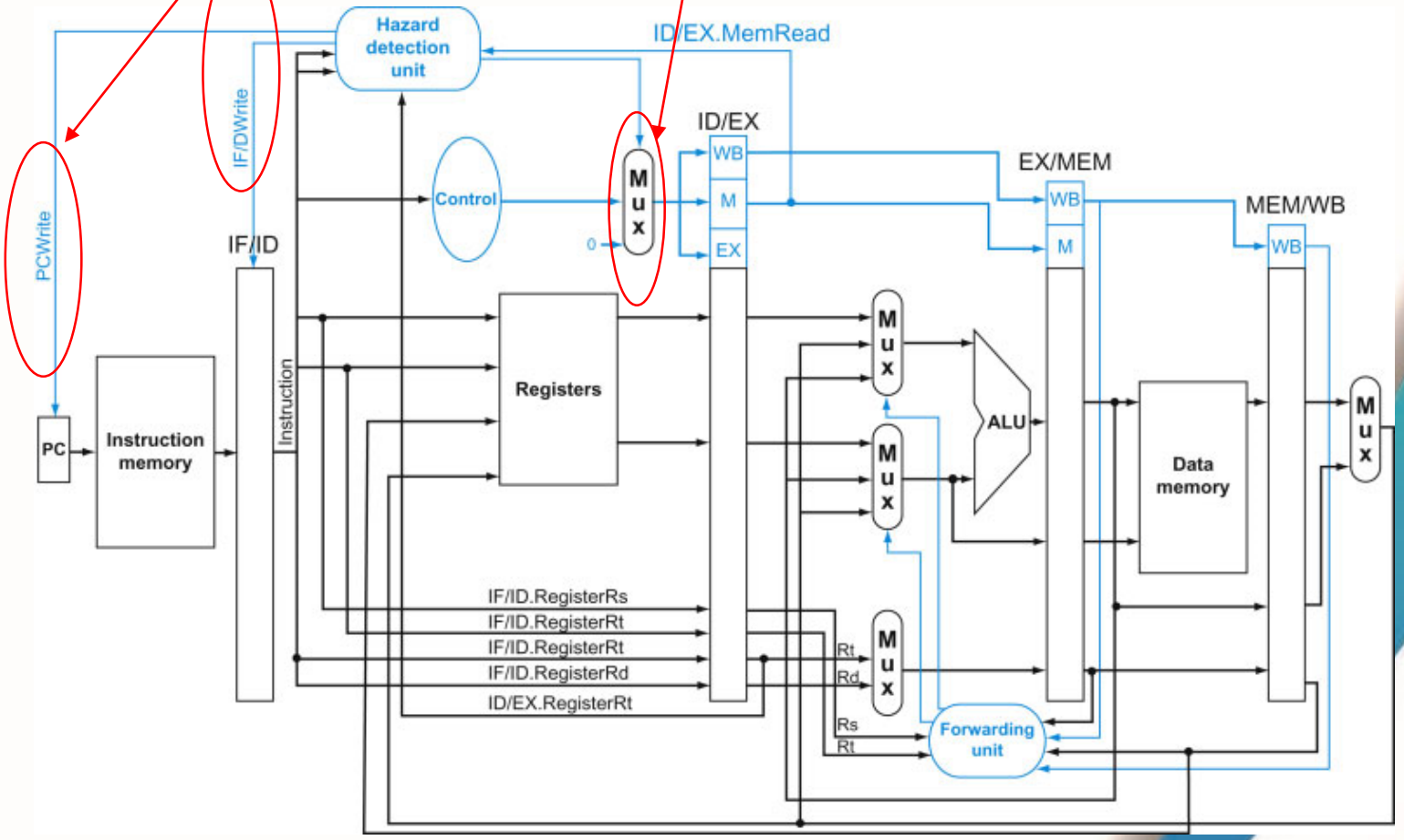


# How to Stall the Pipeline

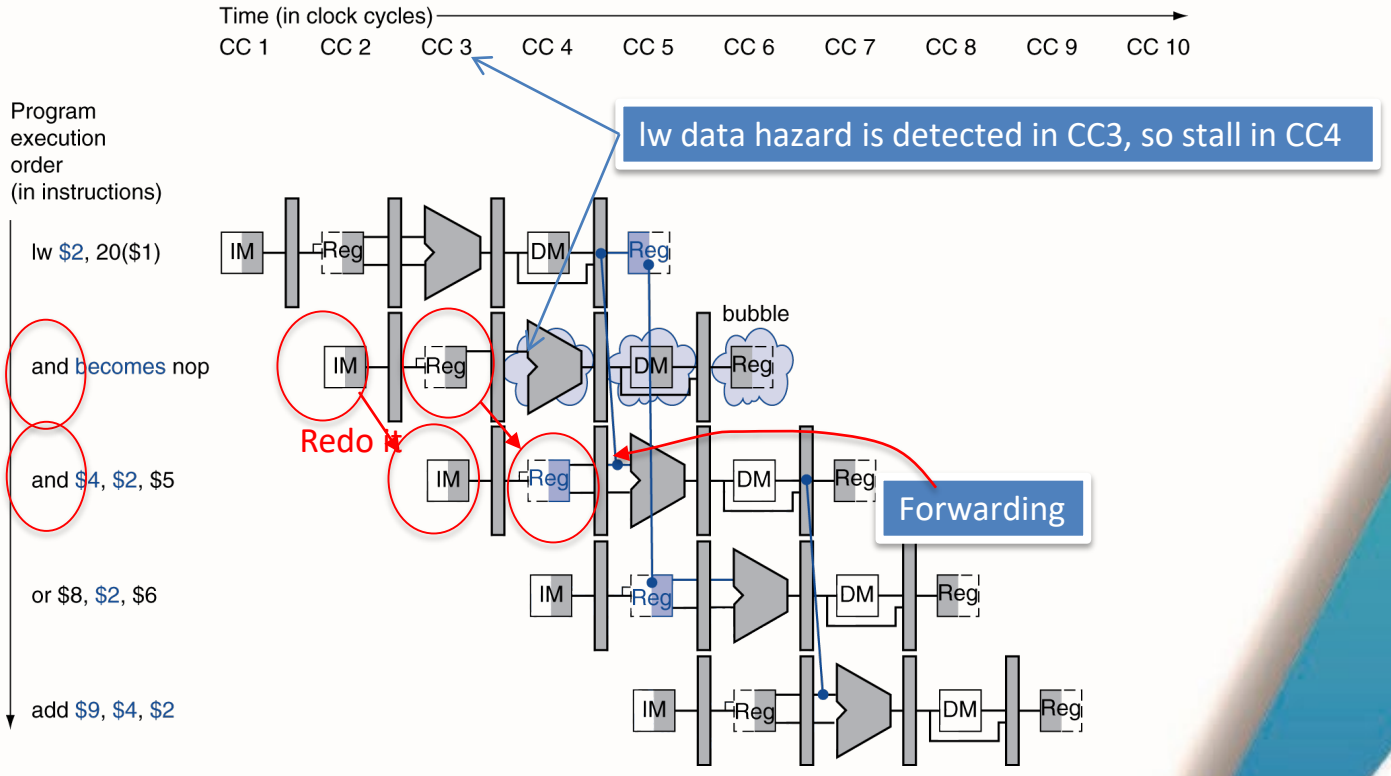
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Current instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Prevent update of PC and IF/ID register

Force control values in ID/EX register to 0



# Stall/Bubble in the Pipeline (CC3)



# Stalls and Performance

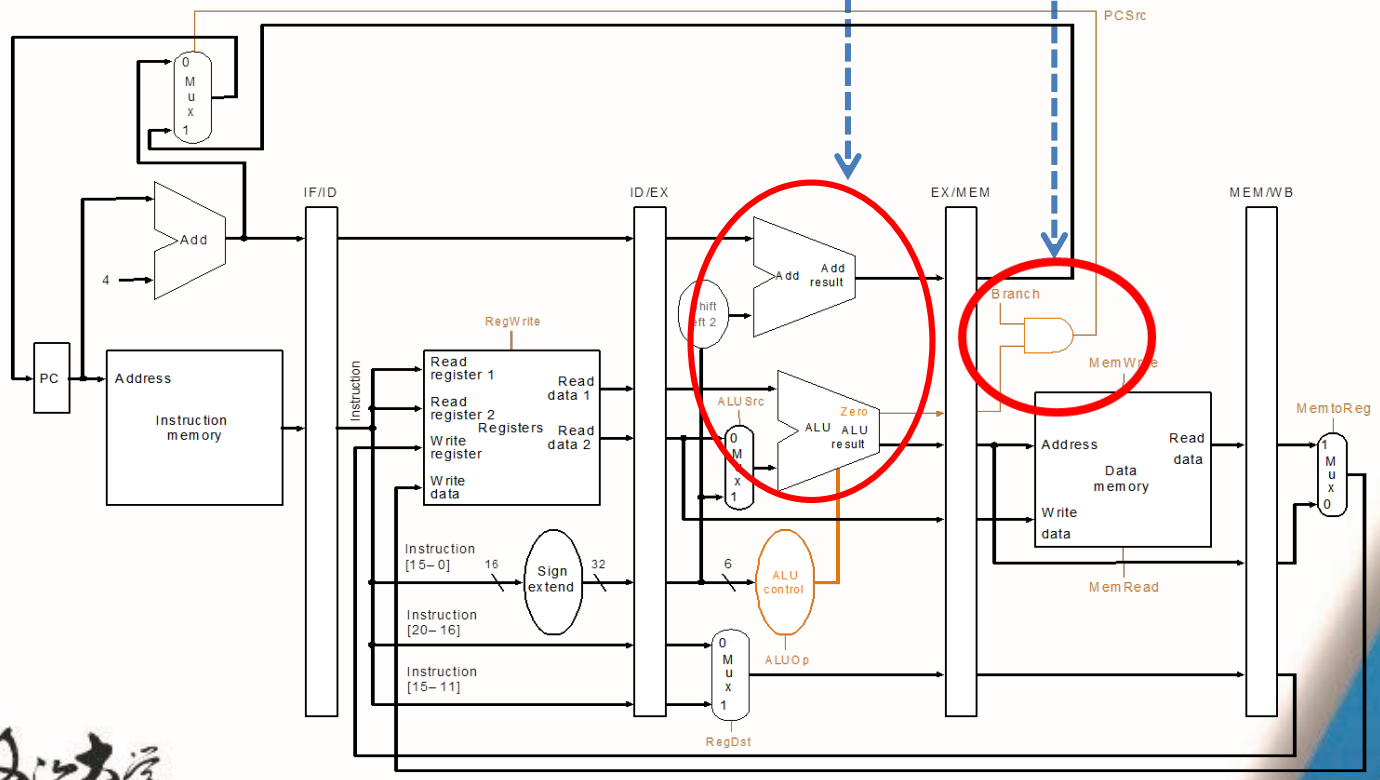
- Stalls reduce performance
  - It is required to obtain correct results
- Compiler can re-arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

# Branch

| Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|---|---|---|---|---|
| Instruction fetch | IR = Memory[PC] | | | |
| | PC = PC + 4 | | | |
| Instruction | A = Reg [IR[25-21]] | | | |
| decode/register fetch | B = Reg [IR[20-16]] | | | |
| Execution, address | ALUOut = A op B | ALUOut = A + sign-extend | if (A ==B) then | PC = PC [31-28] II |
| computation, branch/ | | (IR[15-0]) | PC = PC+ | (IR[25-0]<<2) |
| jump completion | | | sign-ext(IR[15-0])<<2 | |
| Memory access or R-type | Reg [IR[15-11]] = | Load: MDR = Memory[ALUOut] | | |
| completion | ALUOut | or | | |
| | | Store: Memory [ALUOut] = B | | |
| Memory read completion | | Load: Reg[IR[20-16]] = MDR | | |
| | | | | |

# Branch Outcome

- Branch in MEM stage
  - Know rs-rt == 0 in EXE stage
  - Control PCSrc in MEM stage



# Control Hazards (Branch Hazards)

- The fetched instruction is not the one that shall be executed because of branch conditions
- Branch determines flow of control
  - Fetching next instruction (IF) depends on branch outcome (MEM)
  - Pipeline can't always fetch correct instruction
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

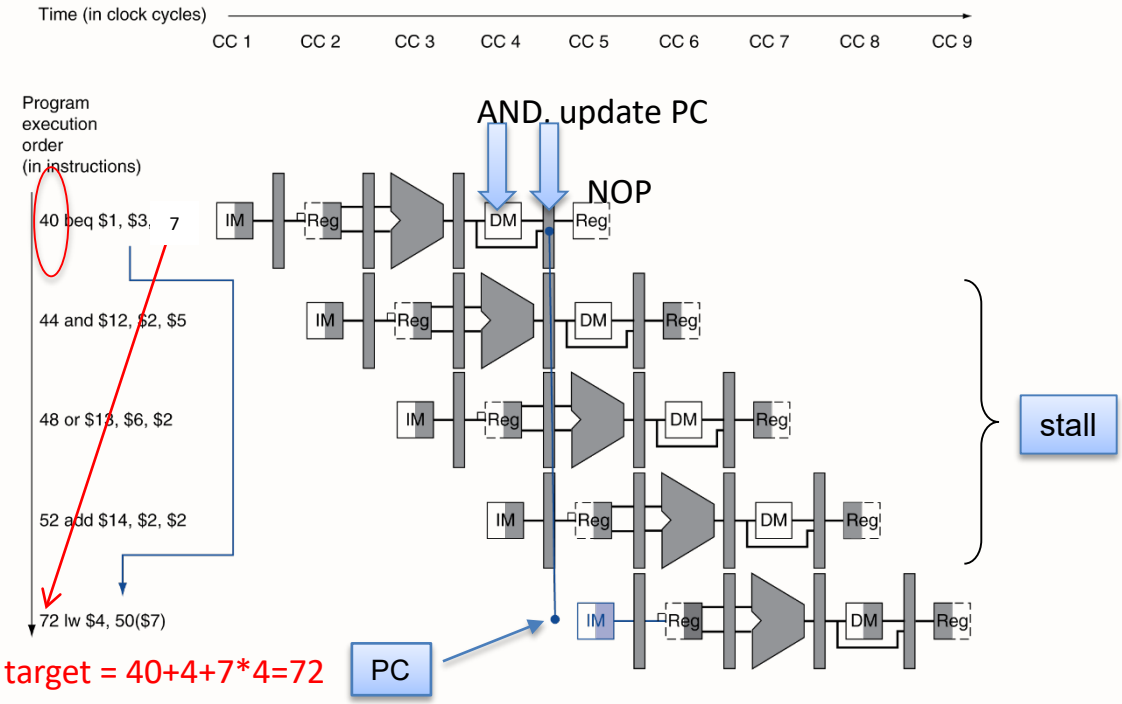
See P.320, Fig.4.62

Add extra hardware to test registers, calculate the branch address, and update the PC during ID stage

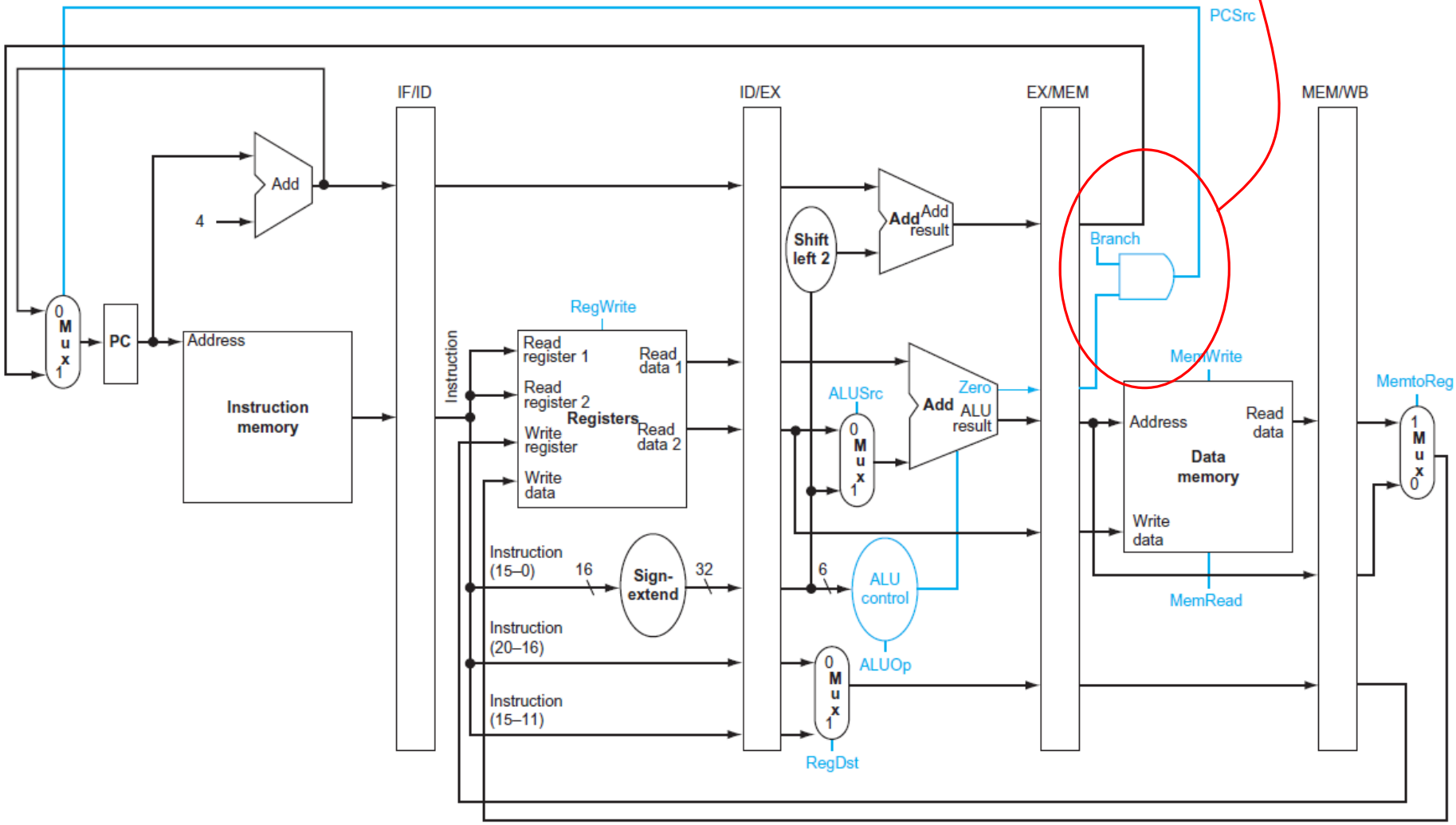
Note that it is often the case for longer pipelines that the branch cannot be resolved in ID stage. (larger slowdown)

# Control Hazards

Branch taken incurs three stalls (3 cycles)



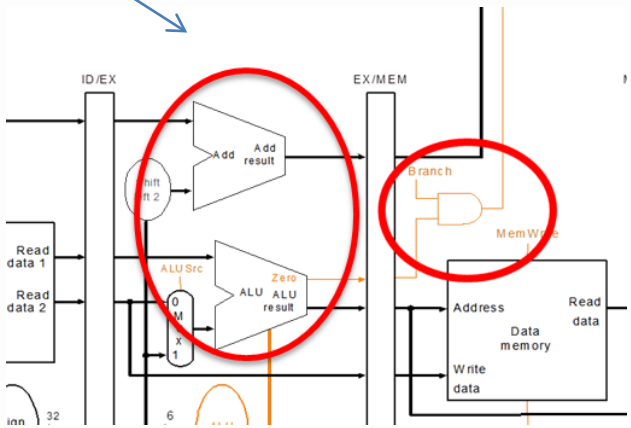
branch finished in MEM



# Reducing Branch Delay

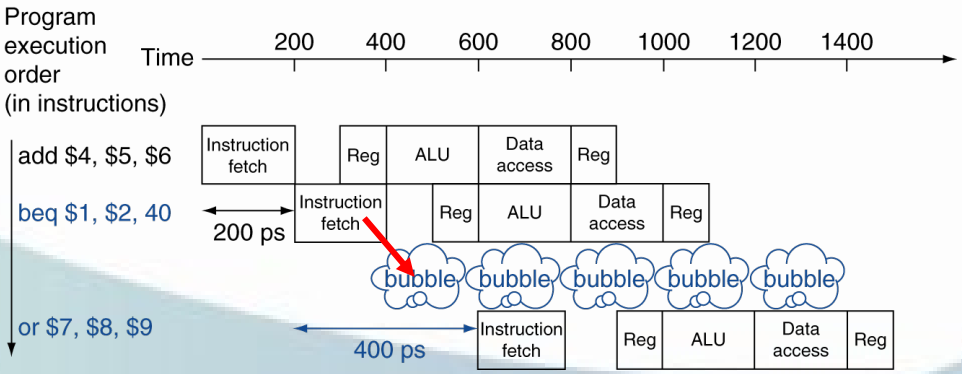
- Determining branch condition in ID stage (using additional hardware)
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```



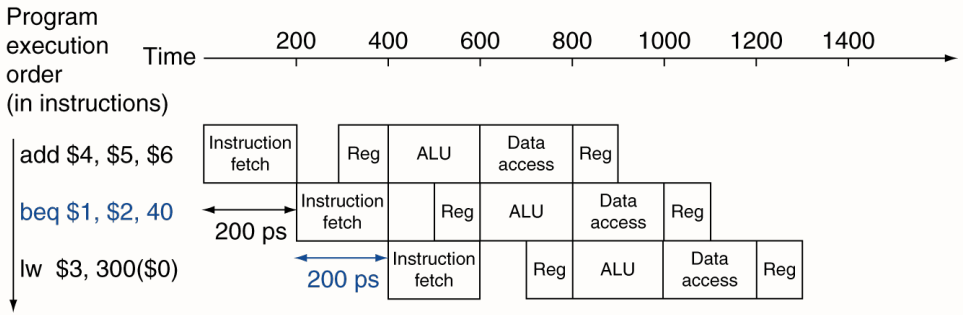
# Stall on Branch

- If branch result comes out in MEM stage, we need to flush instructions in IF/ID, ID/EX registers.
- Even if we know it's taken in ID stage, a stall is needed
- Use **prediction** to handle branches
  - Always predict that branches will not be taken.
  - When the prediction is right, the pipeline can proceed at the full speed; when the branches are taken, the pipeline will stall anyway even without prediction.

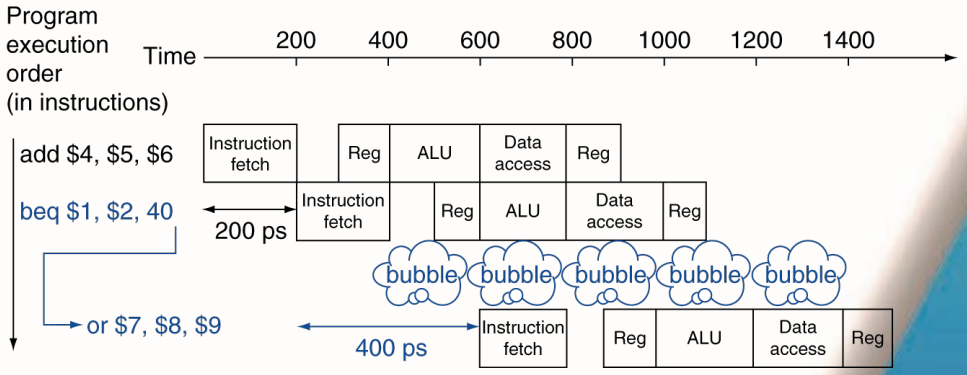


# MIPS with Predict Not Taken

Prediction correct




Prediction incorrect



## Pipelined Branch

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage:

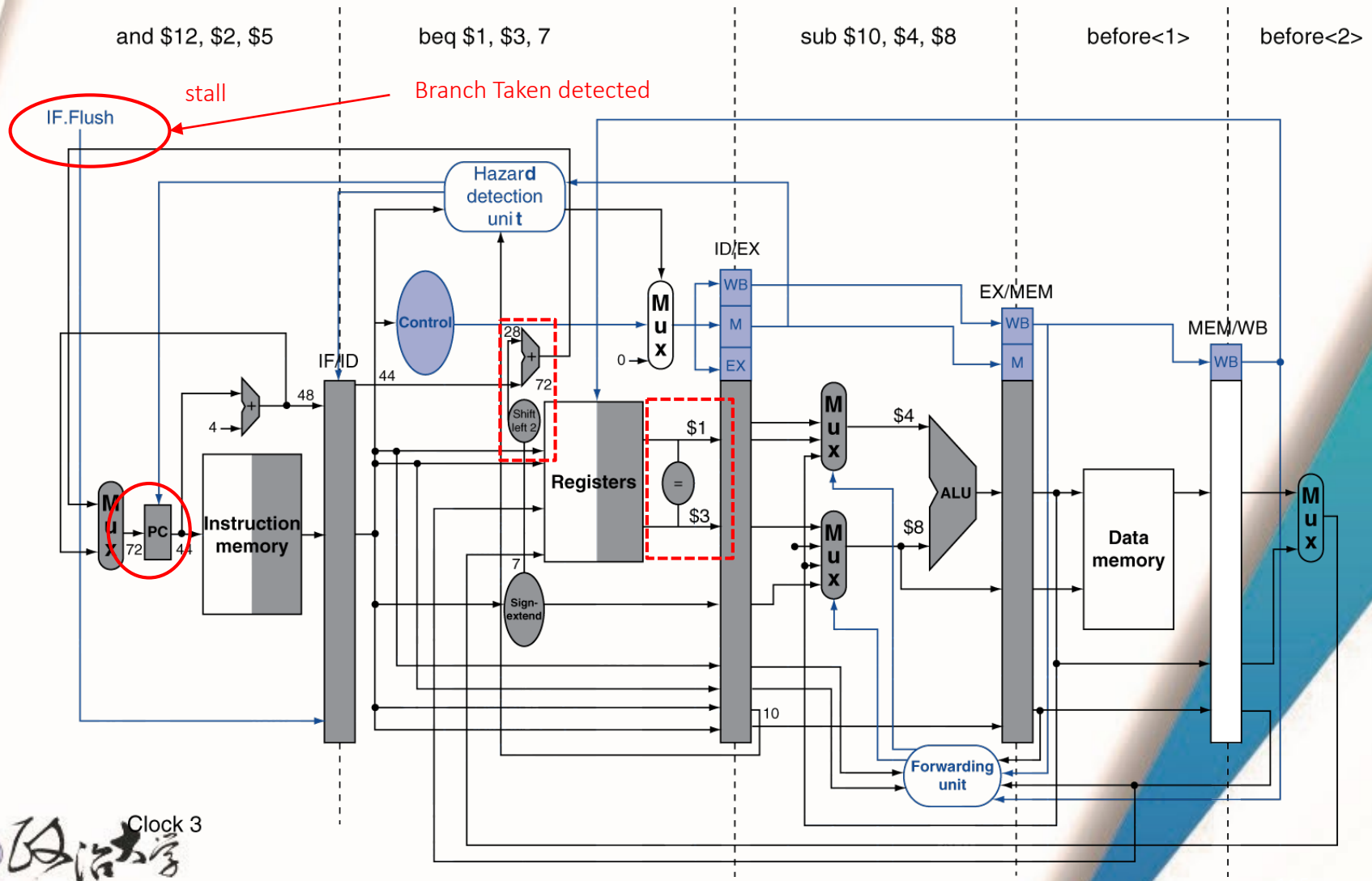
```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```



Taken from p. 319 D. Patterson and J. Hennessy, Computer Organization and Design, 5th edition

# Example: Branch Taken

At the ID stage of CC3, branch taken is detected

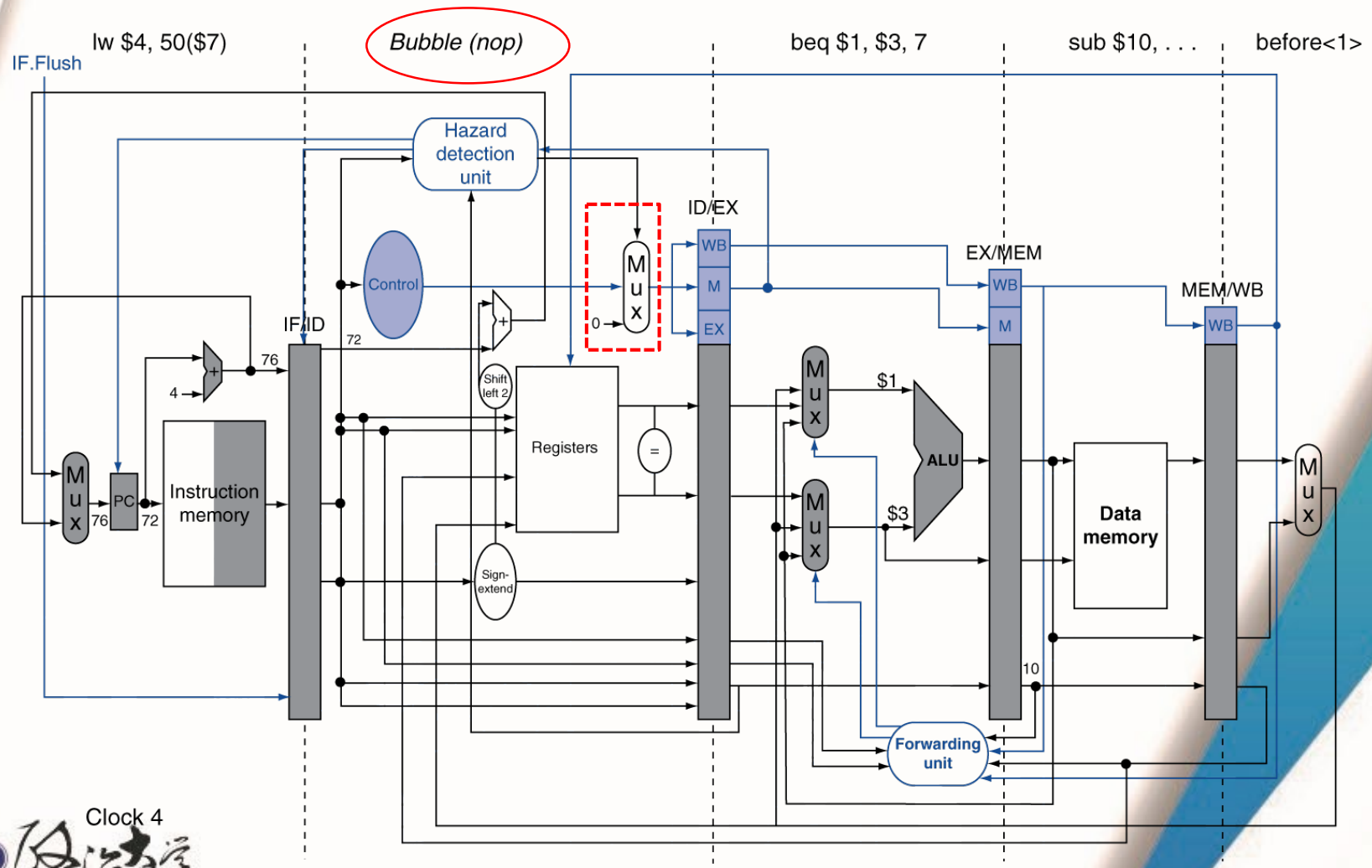



Clock 3

# Example: Branch Taken

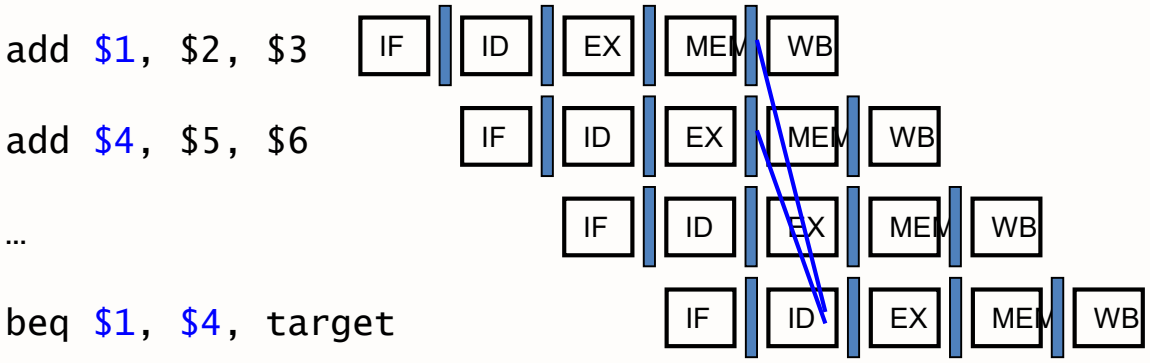
72 is selected as the next PC

Zeros are filled for ID/EX and the instruction (nop: sll $0, $0, 0)



# Data Hazards for Branches

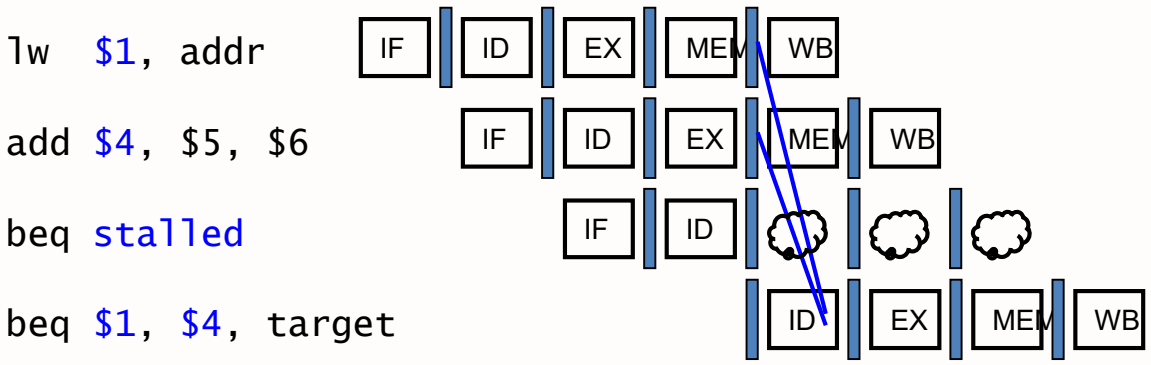
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction




- Can resolve using forwarding

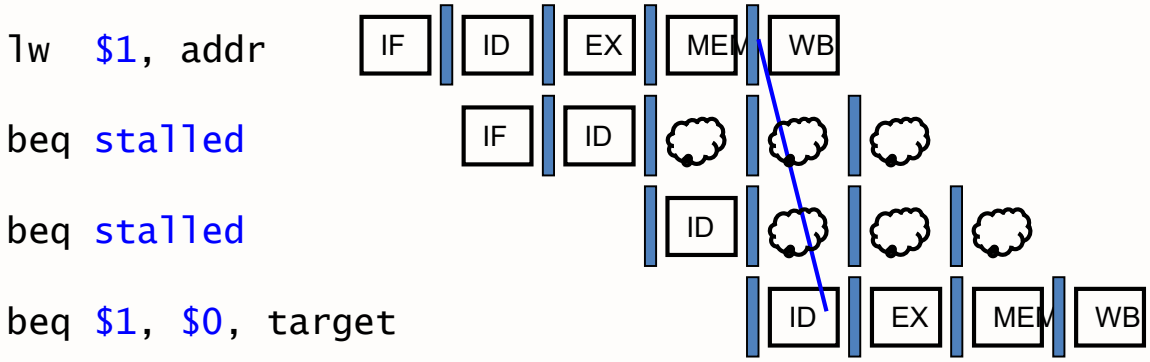
# Data Hazards for Branches

- If a comparison register is a destination of preceding R-type instruction or 2nd preceding load instruction
  - Need 1 stall cycle



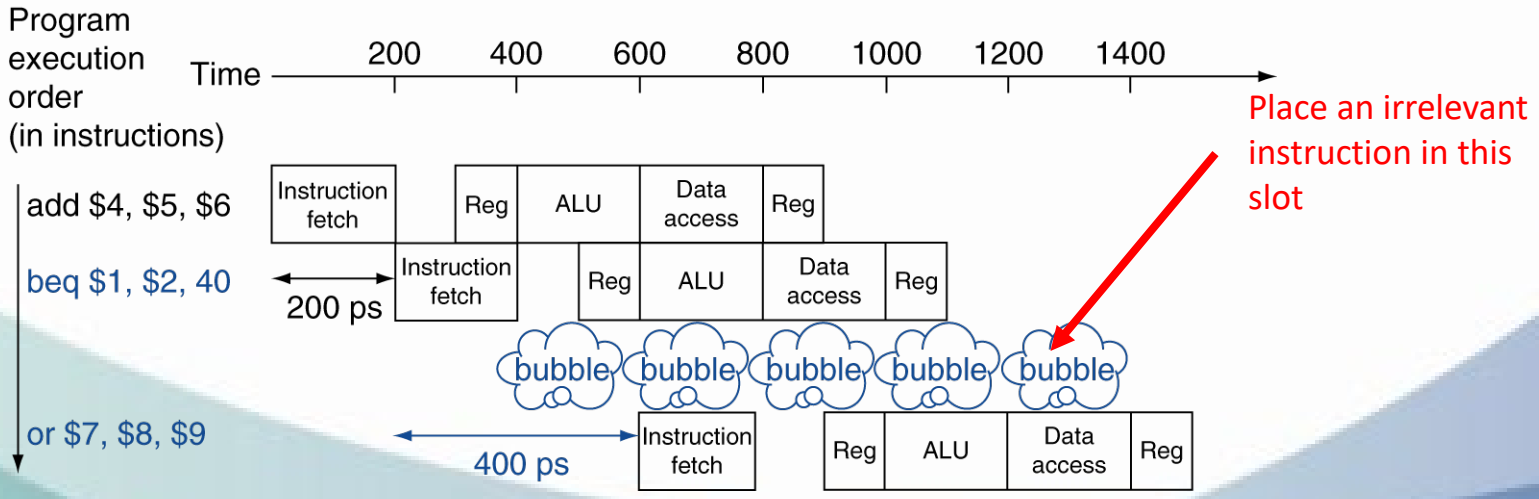
# Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

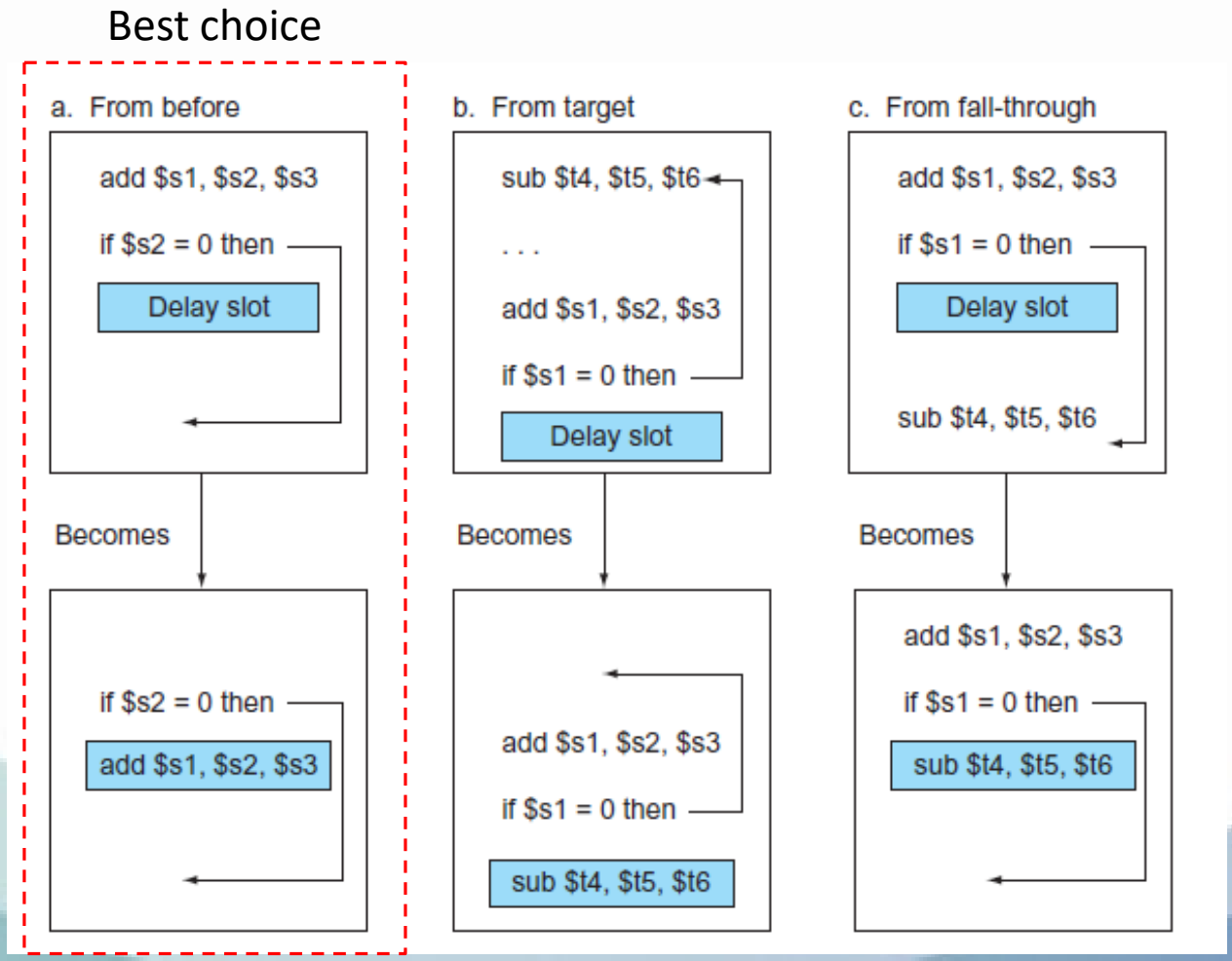


# Delayed Branch

- Combine Predict-not-taken and branch decision in ID stage
  - the following instruction is always executed
  - Insert one irrelevant instruction after a branch instruction
    - Compilers and assemblers would try to insert an always-executed instruction immediate after a branch instruction in the branch delay slot.
  - There is no stall needed for a branch instruction if the compiler can find an instruction to put in slot



# Scheduling a Branch Delay Slot



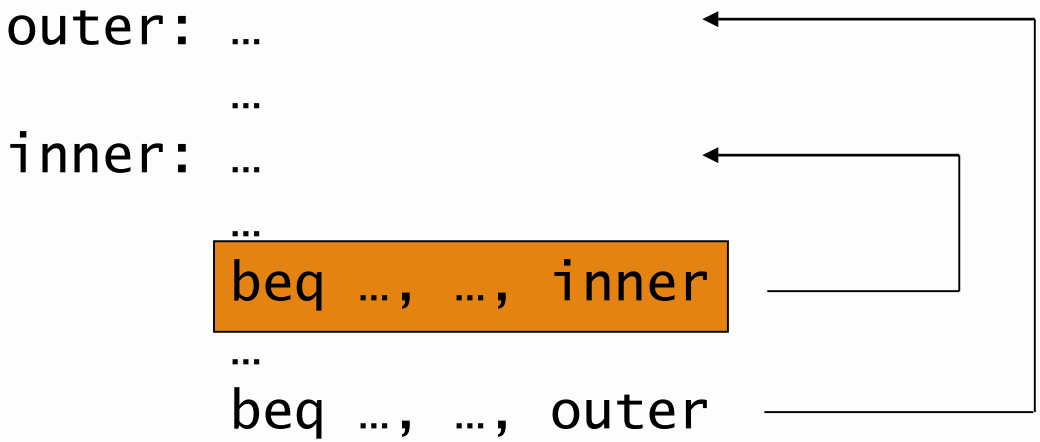
# Sophisticated Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example
    - loop: Predict backward branches taken
    - if: Predict forward branches not taken
- Dynamic branch prediction
  - Prediction of branches at runtime
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

# Dynamic Branch Prediction

- Generally, using "**predict branch not taken**" and stall should often resolve a control hazard
- Under some circumstances, it is not good; for example, "for loop," which is taken almost every time until the stopping criterion is satisfied.
- Dynamic branch prediction
  - branch prediction buffer (also called branch history table)
  - 1-bit Branch prediction buffer
    - a small memory that has a bit to indicate whether the branch was recently taken or not.
    - Using a hash table:<instruction addr, isTaken> to store branch history
    - Do the same action as the previous one did based on the hash table (branch history)

# 1-Bit Predictor




Will lead to twice mispredictions for the Inner loop (in the first and last iterations)

The last iteration: inevitable

# 1-Bit Predictor

```
for(int i=0;i<1000;i++)
{

   for(int j=0;j<1000;j++)  {

    } //A

} //B
```

| | Predict | Outcome |
|---|---|---|
| A | NT | T |
| A | T | T |
| A | T | T |
| …999 | | |
| A | T | NT |
| B | NT | T |
| A | NT | T |
| A | T | T |
| …999 | | |
| A | T | NT |
| B | T | T |
| A | NT | T |
| A | T | T |
| …999 | … | |
| A | T | NT |
| B | T | NT |

0

inevitable

Can be taken care of

Mispredict twice for an entire loop
(mispredict on the first and last loop iterations)

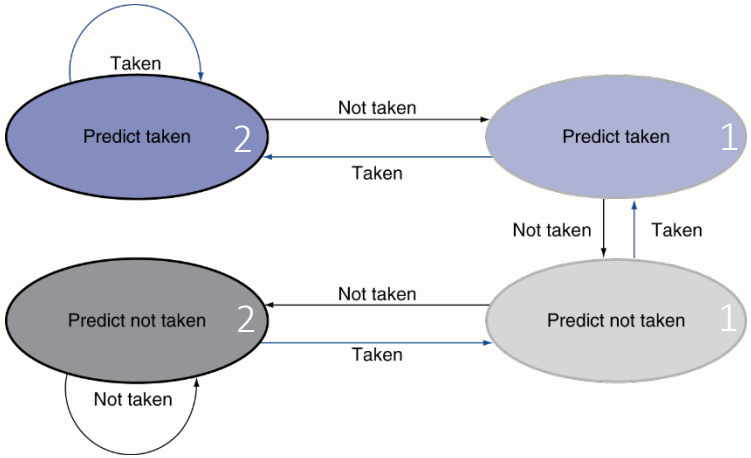
# 2-Bit Predictor

```
for(int i=0;i<1000;i++)
{

   for(int j=0;j<1000;j++)  {

    } //A

} //B
```




0

| | Predict | Outcome |
|---|---|---|
| A | NT1 | T |
| A | T1 | T |
| A | T2 | T |
| …999 | | |
| A | T2 | NT |
| B | NT1 | T |
| A | T1 | T |
| A | T2 | T |
| …999 | | |
| A | T2 | NT |
| B | T1 | T |
| A | T1 | T |
| …999 | … | |
| A | T2 | NT |
| B | T2 | NT |

inevitable

Taken care

# Calculating the Branch Target

- Even though having a predictor, we need to calculate the target address (ID stage)
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
    - The target address was stored when the branch was taken last time
  - Indexed by PC when the instruction is fetched
    - If it hits an entry in the table and was predicted taken, the target address can be fetch from the cache.

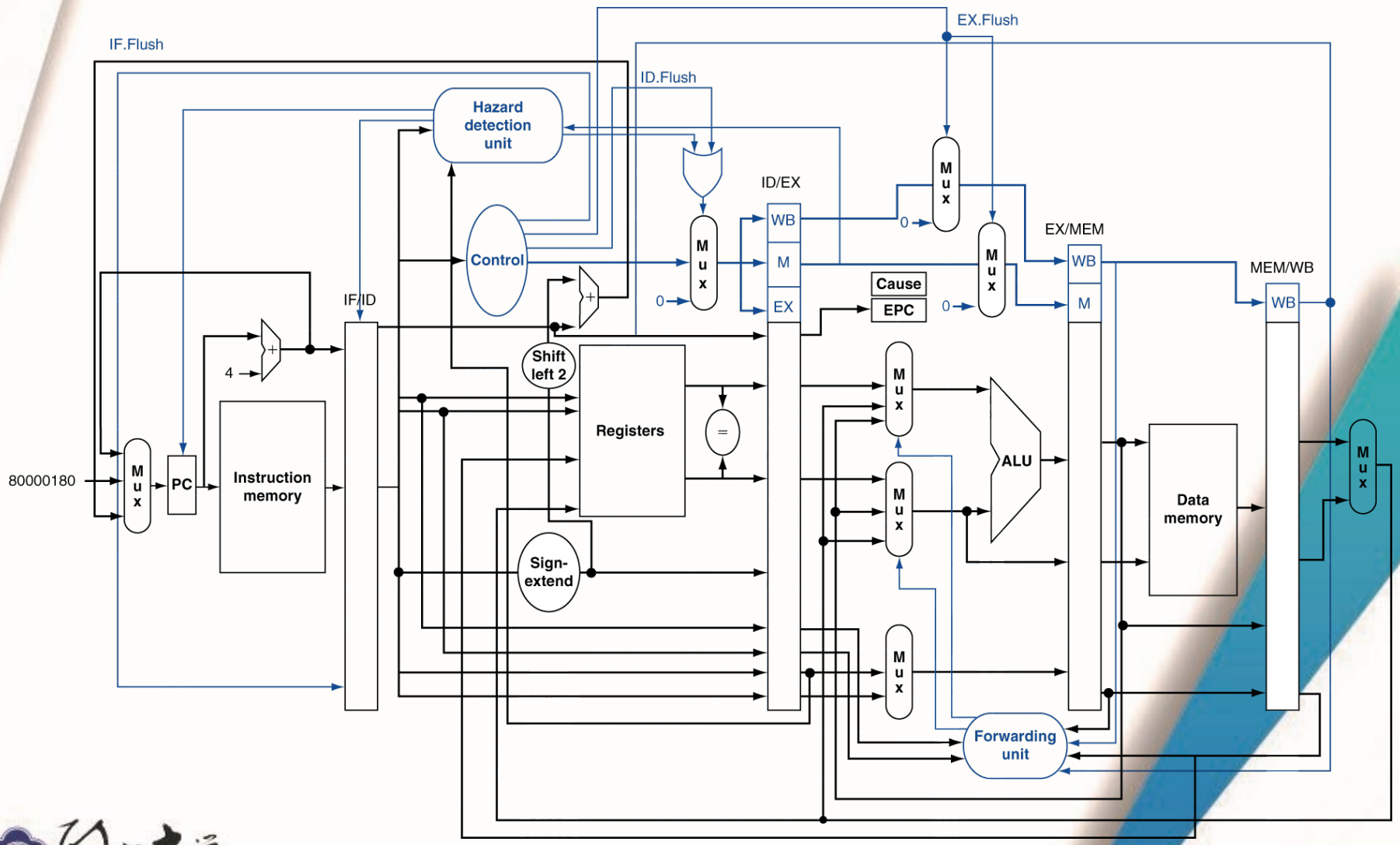
# Conclusion

- Pipelining improves instruction throughput instead of latency
  - More instructions executed using parallelism
- Three Hazards
  - structural, data, control
  - Stall, Forwarding

# MISC.
# Exceptions and Interrupts

- Exceptions raised when unexpected events occur
- Exceptions and interrupts in general
  - Exceptions raised in CPU
    - undefined opcode, overflow, …
  - Interrupt raised from an external I/O controller
- Exceptions in pipeline is considered a control hazard
- Dealing with them in a pipelined architecture requires
  - Complete previous instructions
  - Flush following instructions
  - Set Cause and EPC register values
  - Transfer control to handler (handler at 8000 00180)

# Pipeline with Exceptions

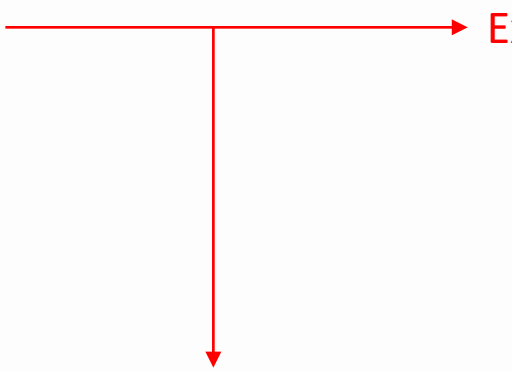


# Exceptions

- Require the pipeline flush the instruction
  - Set control signals to 0
- Executing Handler, then returning to the previous instruction
- PC (PC+4) saved in EPC register
  - Identifies causing instruction
    - Exceptions: PC-4 (resume the current instruction)
    - Interrupt: PC+4 (go to the next instruction)

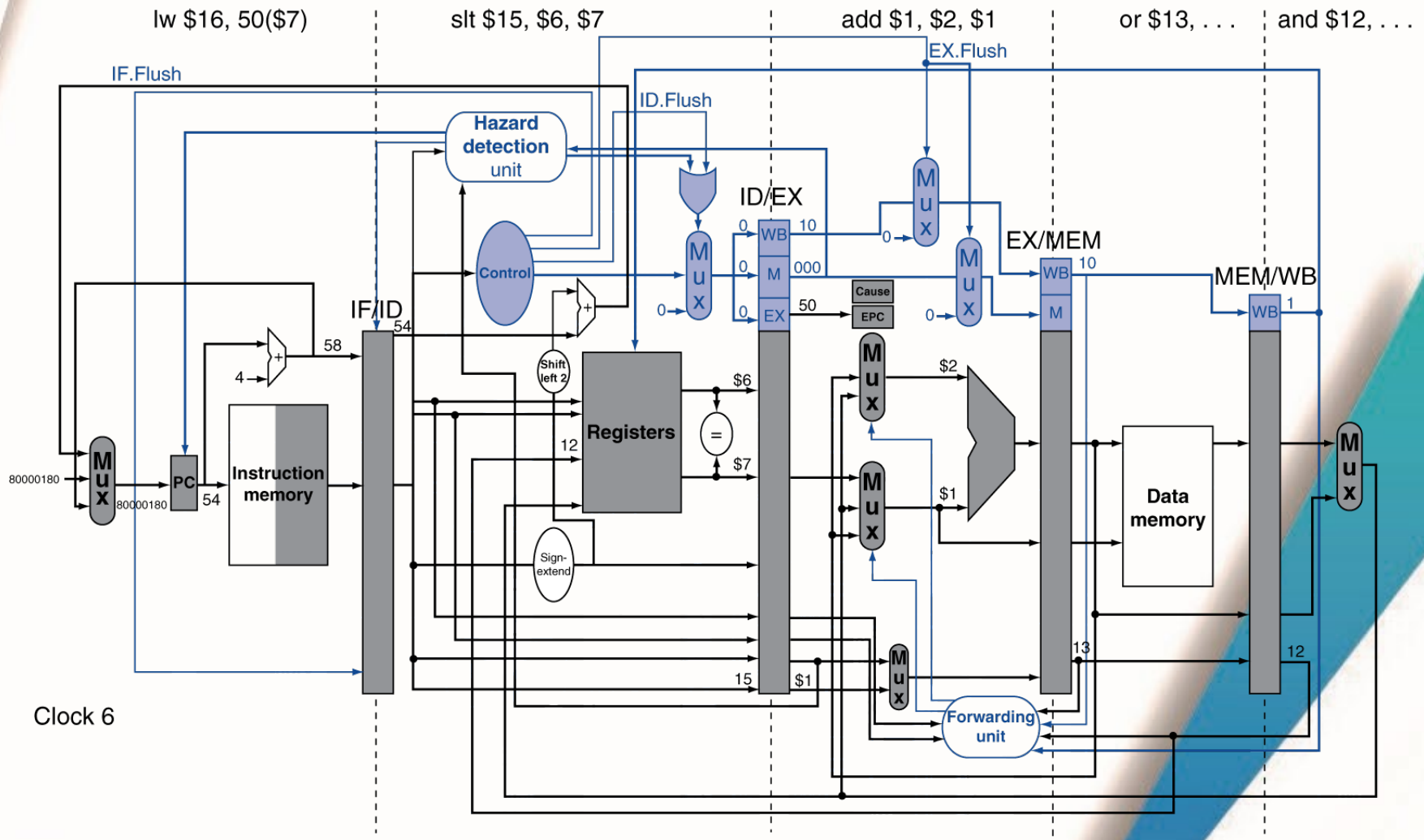
# Exception Example

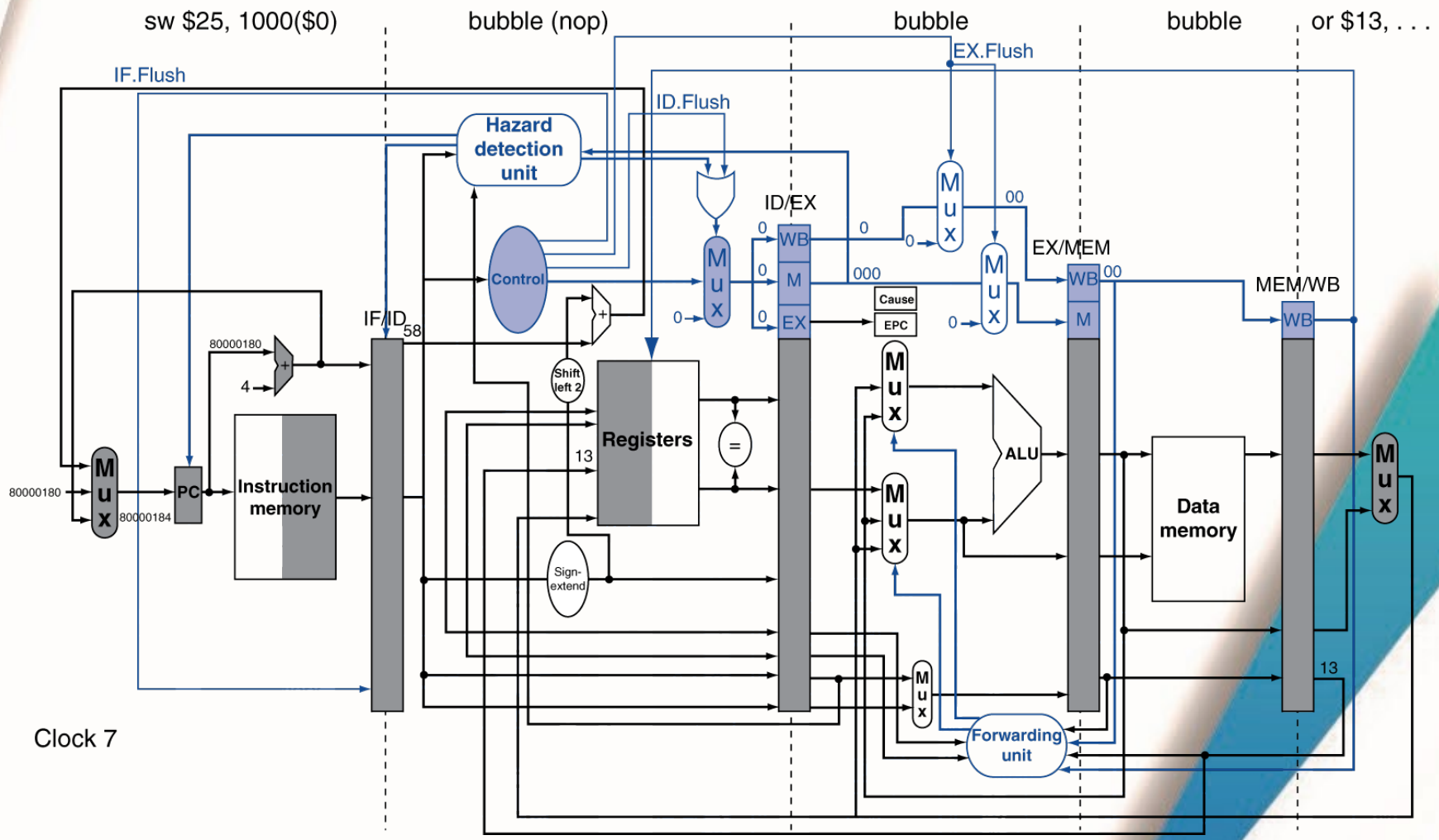
```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4C      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
```




Handler

```
80000180      sw    $25, 1000($0)
80000184      sw    $26, 1004($0)
…
```


lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, . . .
and $12, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Cause
EPC
Shift left 2
Registers
Instruction memory
Data memory
Sign-extend
Forwarding unit
PC
Mux
80000180
$1
$2
$6
$7
Clock 6


sw $25, 1000($0)
bubble (nop)
bubble
bubble
or $13, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Cause
EPC
IF/ID
80000180
4
PC
80000184
Instruction memory
Shift left 2
Registers
Sign-extend
ALU
Data memory
Forwarding unit
13
Clock 7

# Multiple Exceptions Occur

- Deal with multiple exceptions
  - Handle the exception that comes first
    - Flush the following instructions
    - Cannot deal with out-of-order execution
  - Stop pipeline
    - Save all the related states and exception causes
    - Let the software (OS) to handle

# Instruction-Level Parallelism (ILP)

- Pipeline
  - Parallel execution for multiple instructions
- To increase ILP
  - Increase the number of stages (Depth ↑)
    - Running less tasks per stage, leading to a shorter clock cycle
  - Increase the number of pipeline (Width ↑)
    - Adding hardware to fetch more instructions at once
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 2 GHz 4-way multiple instruction parallelism
      - Peak CPI = 0.25, peak IPC = 4, $4 \times 2 \times 10^9 = 8$ billion of instructions per second
      - However, instruction dependencies affect the actual CPI (more than 0.25)

# Multiple Instruction Issue

- Static multiple issue
  - Instructions are grouped to issue by the compiler
  - Compiler deals with hazards
- Dynamic multiple issue
  - CPU chooses which instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU deals with hazards

Ref: Prof. T.-T. Huang, CS4100 course slides

# Speculation

- In static and dynamic multiple issues
- Guess on
  - branch target
    - Roll back if path taken is different
  - load
    - Since data not loaded could affect the following instructions that need it, load needs to be executed earlier
      - Roll back if location is updated

Ref: Prof. T.-T. Huang, CS4100 course slides

# More Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Ref: Prof. T.-T. Huang, CS4100 course slides

# Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before store
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Ref: Prof. T.-T. Huang, CS4100 course slides

# Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Ref: Prof. T.-T. Huang, CS4100 course slides

# Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

| Microprocessor | Year | Clock Rate | Pipeline Stages | Issue width | Out-of-order/ Speculation | Cores | Power |
|---|---|---|---|---|---|---|---|
| i486 | 1989 | 25MHz | 5 | 1 | No | 1 | 5W |
| Pentium | 1993 | 66MHz | 5 | 2 | No | 1 | 10W |
| Pentium Pro | 1997 | 200MHz | 10 | 3 | Yes | 1 | 29W |
| P4 Willamette | 2001 | 2000MHz | 22 | 3 | Yes | 1 | 75W |
| P4 Prescott | 2004 | 3600MHz | 31 | 3 | Yes | 1 | 103W |
| Core | 2006 | 2930MHz | 14 | 4 | Yes | 2 | 75W |
| UltraSparc III | 2003 | 1950MHz | 14 | 4 | No | 1 | 90W |
| UltraSparc T1 | 2005 | 1200MHz | 6 | 1 | No | 8 | 70W |

Ref: Prof. T.-T. Huang, CS4100 course slides

# Conclusions

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Ref: Prof. T.-T. Huang, CS4100 course slides