THE UNIVERSITY of WISCONSIN MADISON

- Please complete the UNGRADED quiz on HW2 performance. It is called HW2 Review and Verilog

ECE/CS 552: Pipeline Hazards

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Pipeline Hazards

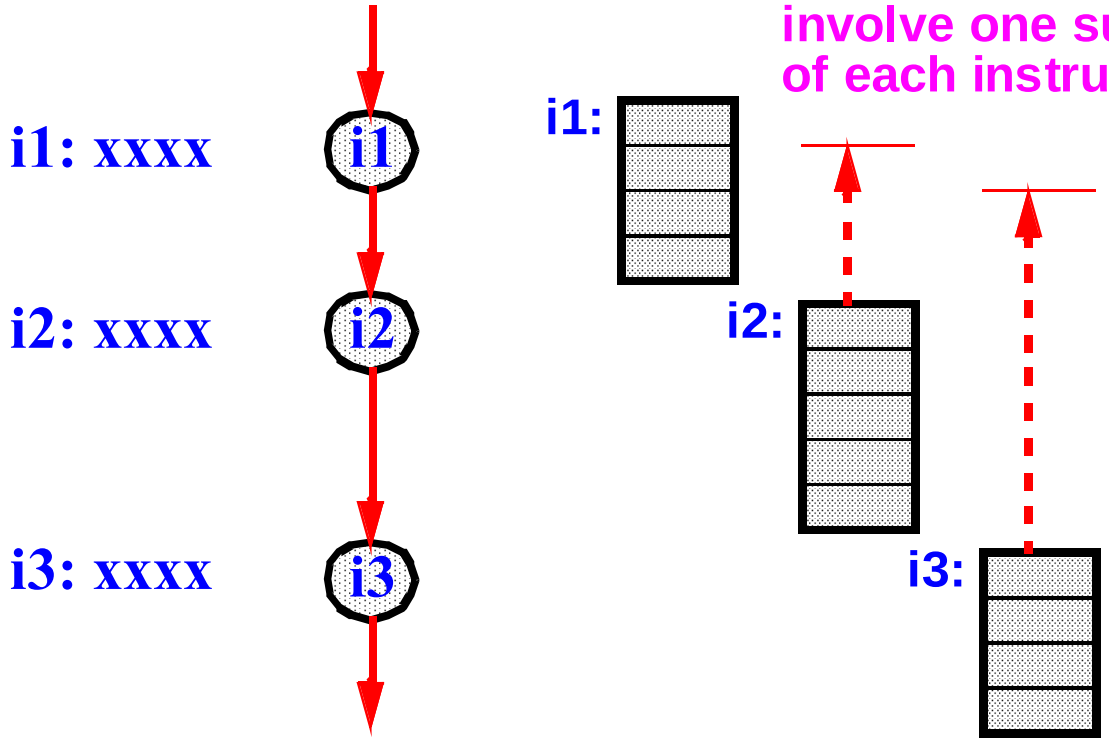
- Forecast
  - Program Dependences
  - Data Hazards
    - Stalls
    - Forwarding
  - Control Hazards
  - Exceptions

Sequential Execution Model

- MIPS ISA requires the appearance of *sequential execution*
  - *Precise exceptions*
  - True of most general purpose ISAs

Program Dependences

A true dependence between two instructions may only involve one subcomputation of each instruction.



The implied sequential precedences are an overspecification. It is sufficient but not necessary to ensure program correctness.

Program Data Dependences

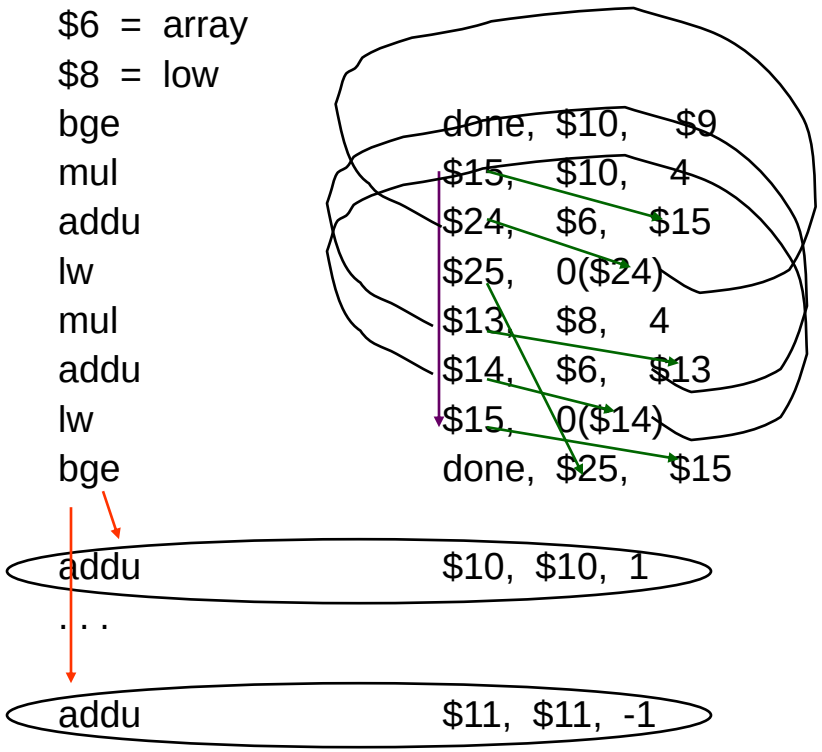
- True dependence (RAW) $D(i) \cap R(j) \neq \phi$
  - j cannot execute until i produces its result
- Anti-dependence (WAR) $R(i) \cap D(j) \neq \phi$
  - j cannot write its result until i has read its sources
- Output dependence (WAW) $D(i) \cap D(j) \neq \phi$
  - j cannot write its result until i has written its result

Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction

Example (quicksort/MIPS)

```
#       for (;  (j < high)  &&  (array[j] < array[low])  ;  ++j  );
#       $10  =  j
#       $9  =  high
#       $6  =  array
#       $8  =  low
        bge     done,  $10,    $9
        mul     $15,   $10,    4
        addu    $24,   $6,     $15
        lw      $25,   0($24)
        mul     $13,   $8,     4
        addu    $14,   $6,     $13
        lw      $15,   0($14)
        bge     done,  $25,    $15
cont:
        addu    $10,  $10,  1
        . . .
done:
        addu    $11,  $11,  -1
```



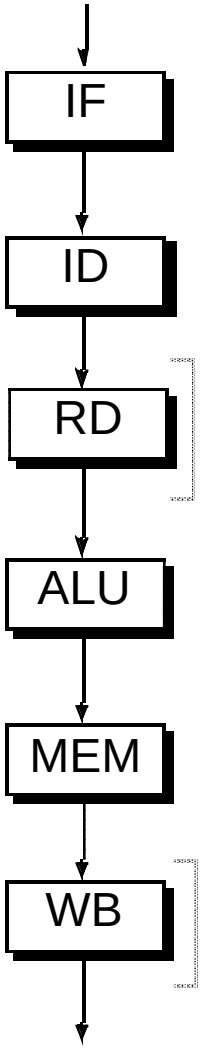
Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

Pipeline Hazards

- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline Hazard Analysis



- Memory hazards
  - RAW: Yes/No?
  - WAR: Yes/No?
  - WAW: Yes/No?
- Register hazards
  - RAW: Yes/No?
  - WAR: Yes/No?
  - WAW: Yes/No?

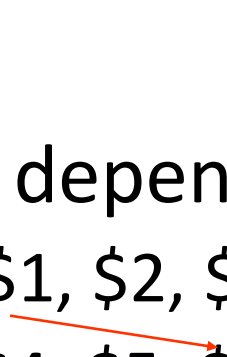
RAW Hazard

- Earlier instruction produces a value used by a later instruction:
  - add $1, $2, $3
  - sub $4, $5, $1

Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
add	F	D	X	M	W								
sub		F	D	X	M	W							

RAW Hazard - Stall

- Detect dependence and stall:
  - add $1, $2, $3
  - sub $4, $5, $1



Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
add	F	D	X	M	W								
sub						F	D	X	M	W			

Control Dependence

- One instruction affects which executes next
  - sw $4, 0($5)
  - bne $2, $3, loop
  - sub $6, $7, $8

Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
sw	F	D	X	M	W								
bne		F	D	X	M	W							
sub			F	D	X	M	W						

Control Dependence - Stall

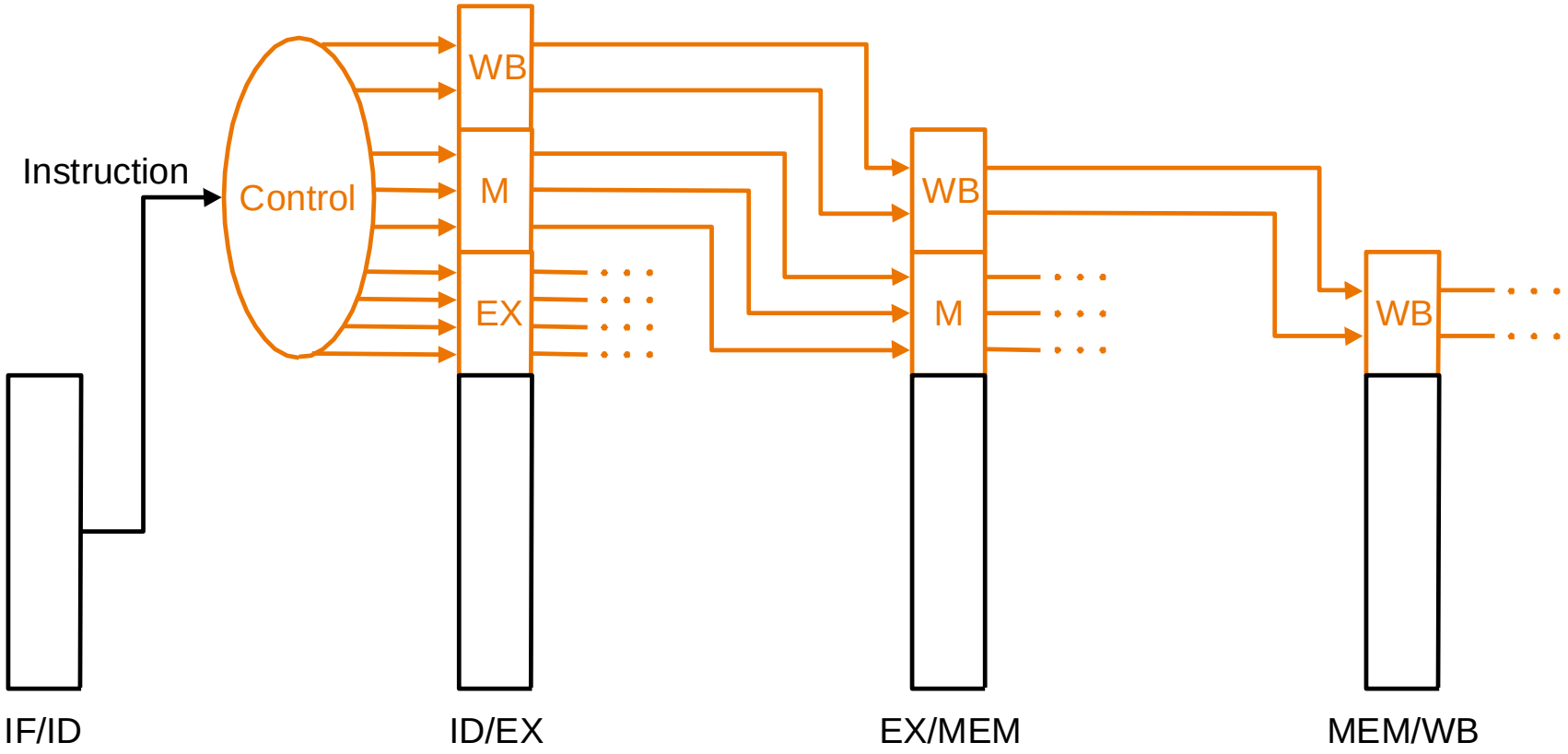
- Detect dependence and stall
  - sw $4, 0($5)
  - bne $2, $3, loop
  - sub $6, $7, $8

Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
sw	F	D	X	M	W								
bne		F	D	X	M	W							
sub					F	D	X	M	W				

Pipelined Control

- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline
  - No explicit FSM
  - Instead, distributed FSM

Pipelined Control



RAW Hazards

- Must first detect RAW hazards
  - Pipeline analysis proves that WAR/WAW don't occur

ID/EX.WriteRegister = IF/ID.ReadRegister1

ID/EX.WriteRegister = IF/ID.ReadRegister2

EX/MEM.WriteRegister = IF/ID.ReadRegister1

EX/MEM.WriteRegister = IF/ID.ReadRegister2

MEM/WB.WriteRegister = IF/ID.ReadRegister1

MEM/WB.WriteRegister = IF/ID.ReadRegister2

RAW Hazards

- Not all hazards because
  - WriteRegister not used (e.g. sw)
  - ReadRegister not used (e.g. addi, jump)
  - Do something only if necessary

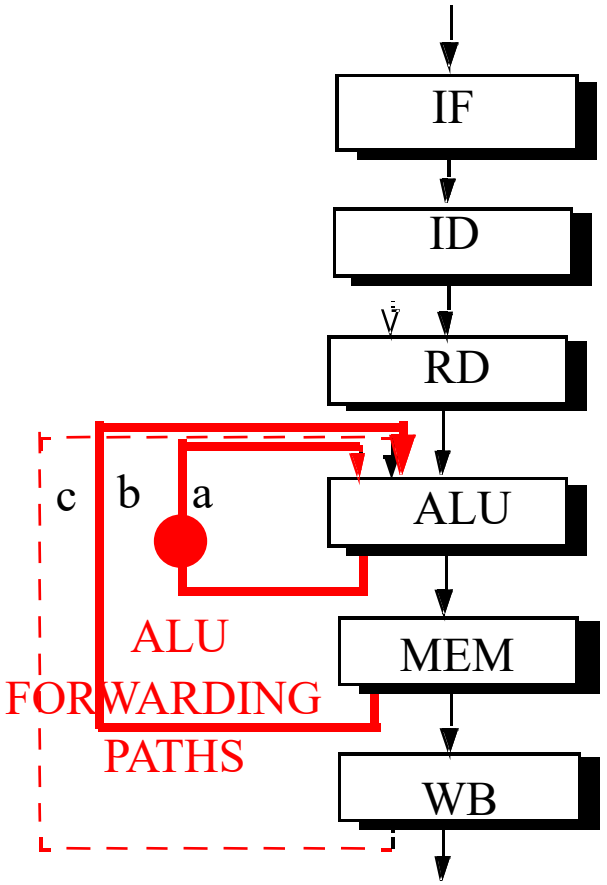
RAW Hazards

- Hazard Detection Unit
  - Several 5-bit comparators
- Response? Stall pipeline
  - Instructions in IF and ID stay
  - IF/ID pipeline latch not updated
  - Send ‘nop’ down pipeline (called a bubble)
  - PCWrite, IF/IDWrite, and nop mux

RAW Hazard Forwarding

- A better response – forwarding
  - Also called bypassing
- Comparators ensure register is read after it is written
- Instead of stalling until write occurs
  - Use mux to select forwarded value rather than register value
  - Control mux with hazard detection logic

Forwarding Paths (ALU instructions)



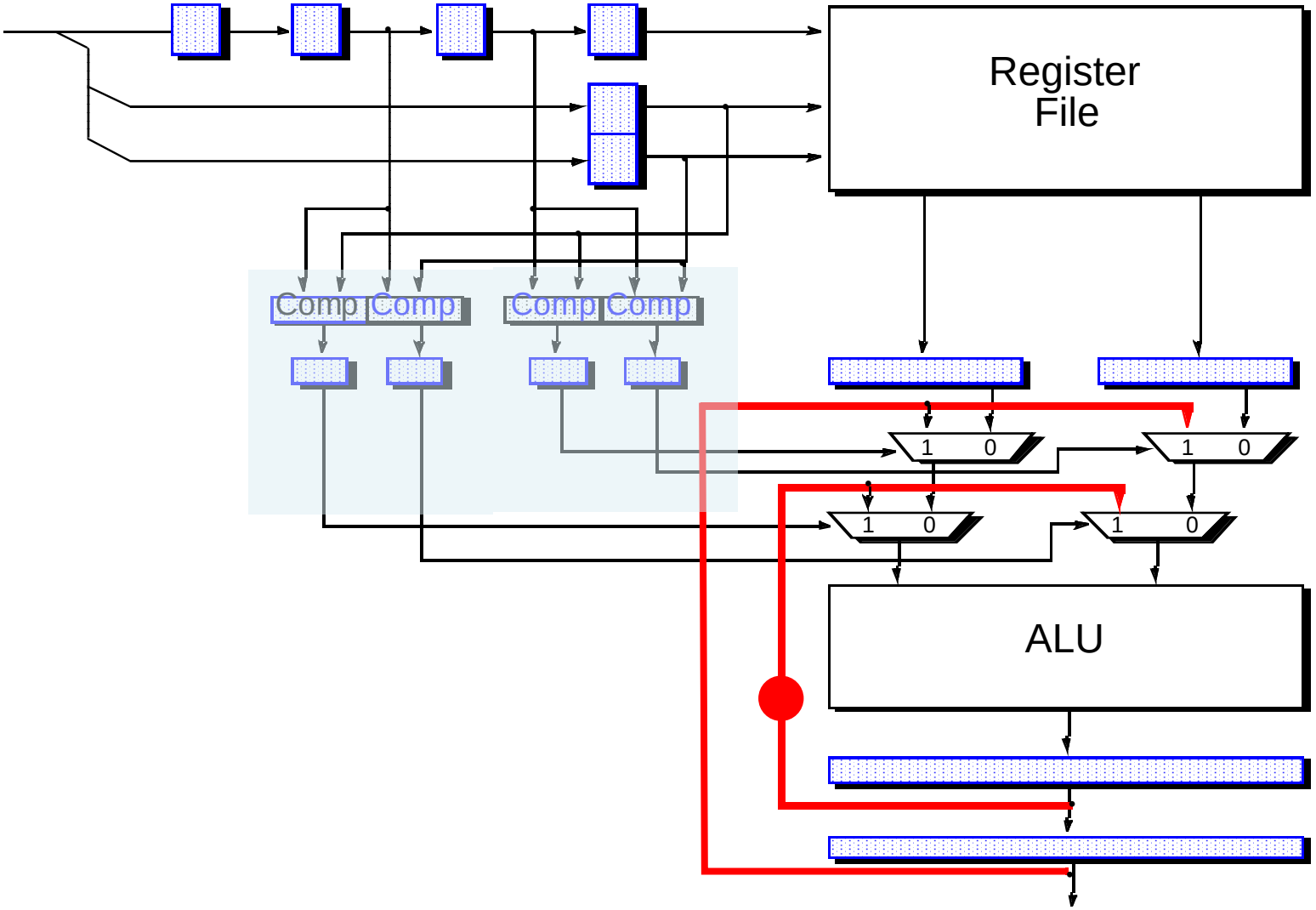
i+1: ← R1	i+2: ← R1	i+3: ← R1
i: R1 ←	i+1: ← R1	i+2: ← R1
	i: R1 ←	i+1: ←
		i: R1 ←
(i → i+1) Forwarding via Path a	**(i → i+2) Forwarding via Path b**	**(i → i+3) i writes R1 before i+3 reads R1**

© 2005 Mikko Lipasti

Write before Read RF

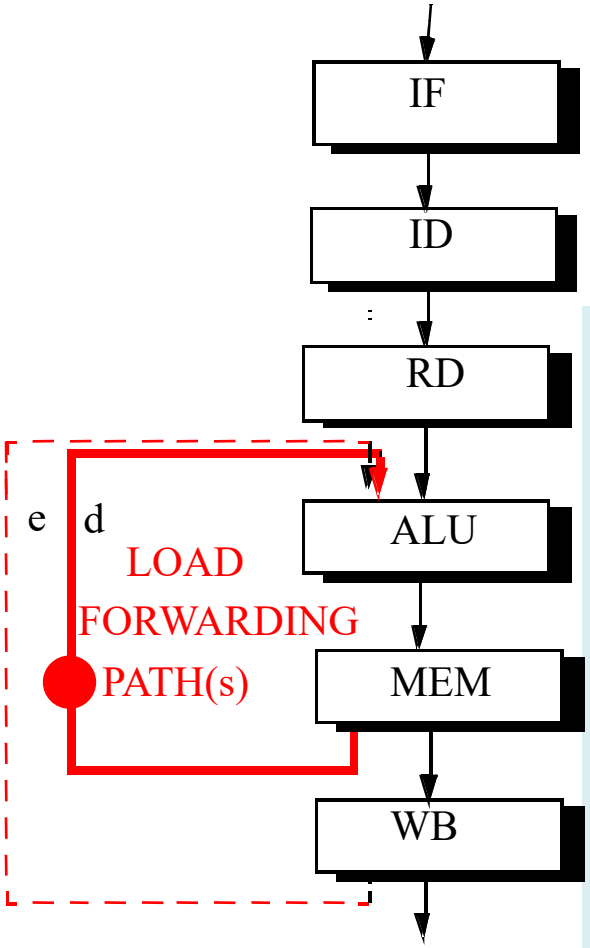
- Register file design
  - 2-phase clocks common
  - Write RF on first phase
  - Read RF on second phase
- Hence, same cycle:
  - Write $1
  - Read $1
- No bypass needed
  - If read before write or DFF-based, need bypass

ALU Forwarding



© 2005 Mikko Lipasti

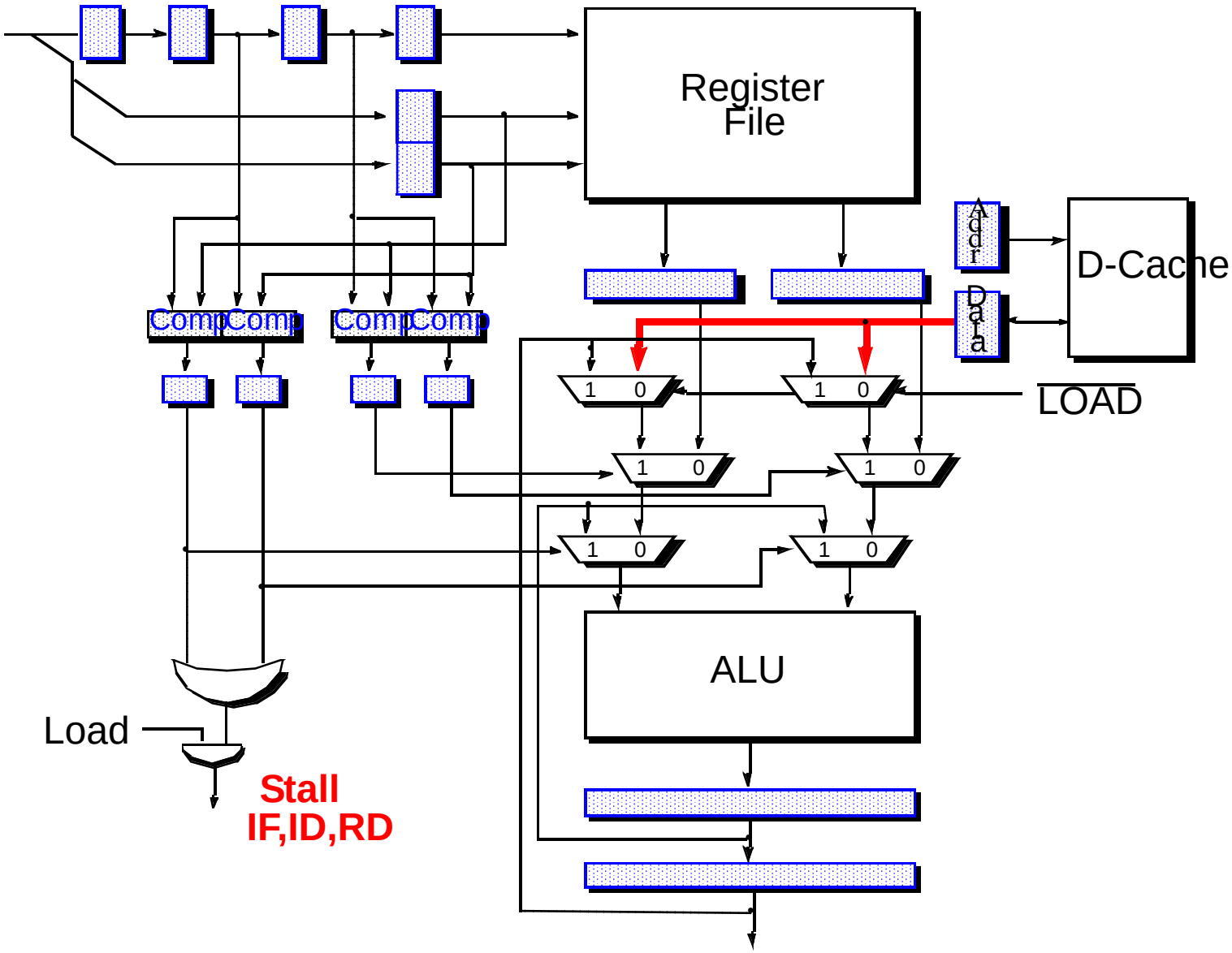
Forwarding Paths (Load instructions)



i+1: ← R1	i+1: ← R1	i+2: ← R1
i:R1 ← MEM[]	——	i+1: ← R1
	i:R1 ← MEM[]	——
		i:R1 ← MEM[]
(i → i+1) Stall i+1	(i → i+1) Forwarding via Path d	(i → i+2) i writes R1 before i+2 reads R1

© 2005 Mikko Lipasti

Implementation of Load Forwarding



Pipeline with Forwarding

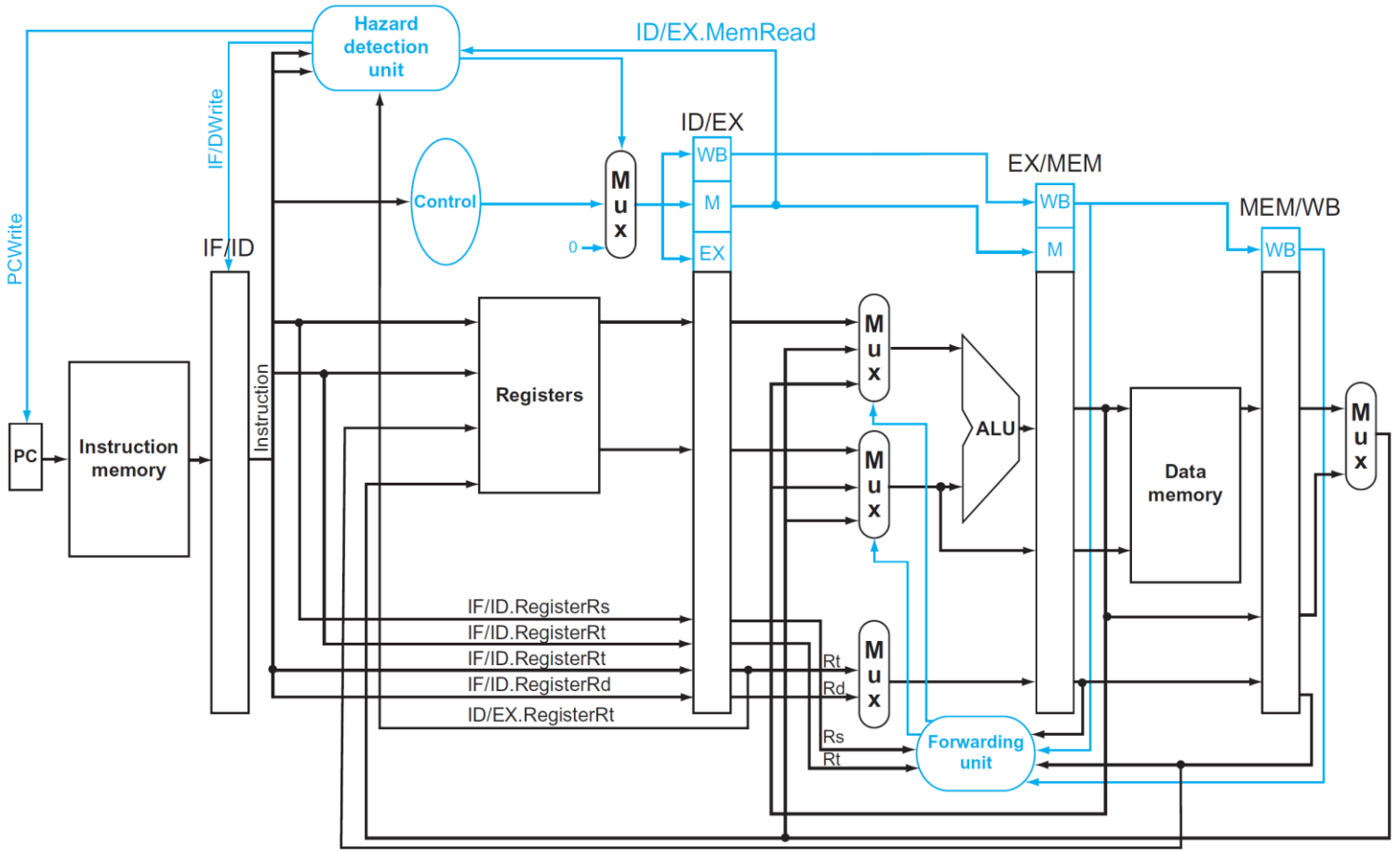


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Control Flow Hazards

- Control flow instructions
  - branches, jumps, jals, returns
  - Can’t fetch until branch outcome known
  - Too late for next IF

Control Flow Hazards

- What to do?
  - Always stall
  - Easy to implement
  - Performs poorly
  - $1/6^{th}$ instructions are branches
    - each branch takes 3 cycles
  - CPI = 1 + 3 x 1/6 = 1.5 (lower bound)

Control Flow Hazards

- Predict branch not taken
- Send sequential instructions down pipeline
- Kill instructions later if incorrect
- Must stop memory accesses and RF writes
- Late flush of instructions on misprediction
  - Complex
  - Global signal (wire delay)

Control Flow Hazards

- Even better but more complex
  - Predict taken
  - Predict both (eager execution)
  - Predict one or the other dynamically
    - Adapt to program branch patterns
    - Lots of chip real estate these days
      - Core i7, ARM A15
    - Current research topic
  - More later, covered in detail in ECE752

Control Flow Hazards

- Another option: delayed branches
  - Always execute following instruction
  - “delay slot” (later example on MIPS pipeline)
  - Put useful instruction there, otherwise ‘nop’
- A mistake to cement this into ISA
  - Just a stopgap (one cycle, one instruction)
  - Superscalar processors (later)
    - Delay slot just gets in the way

Pipelined Datapath and Control path

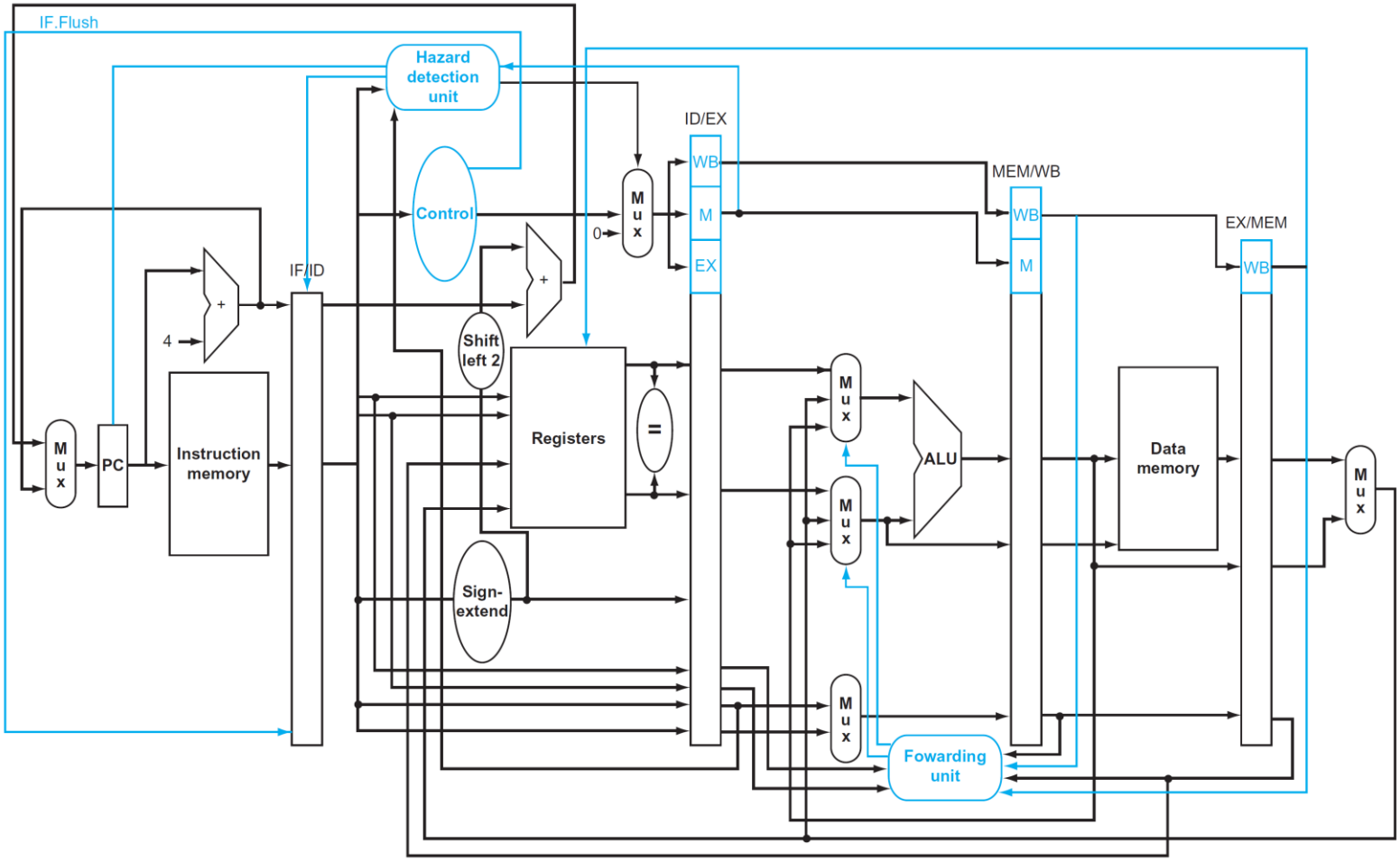


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

Pipeline Walkthrough Examples

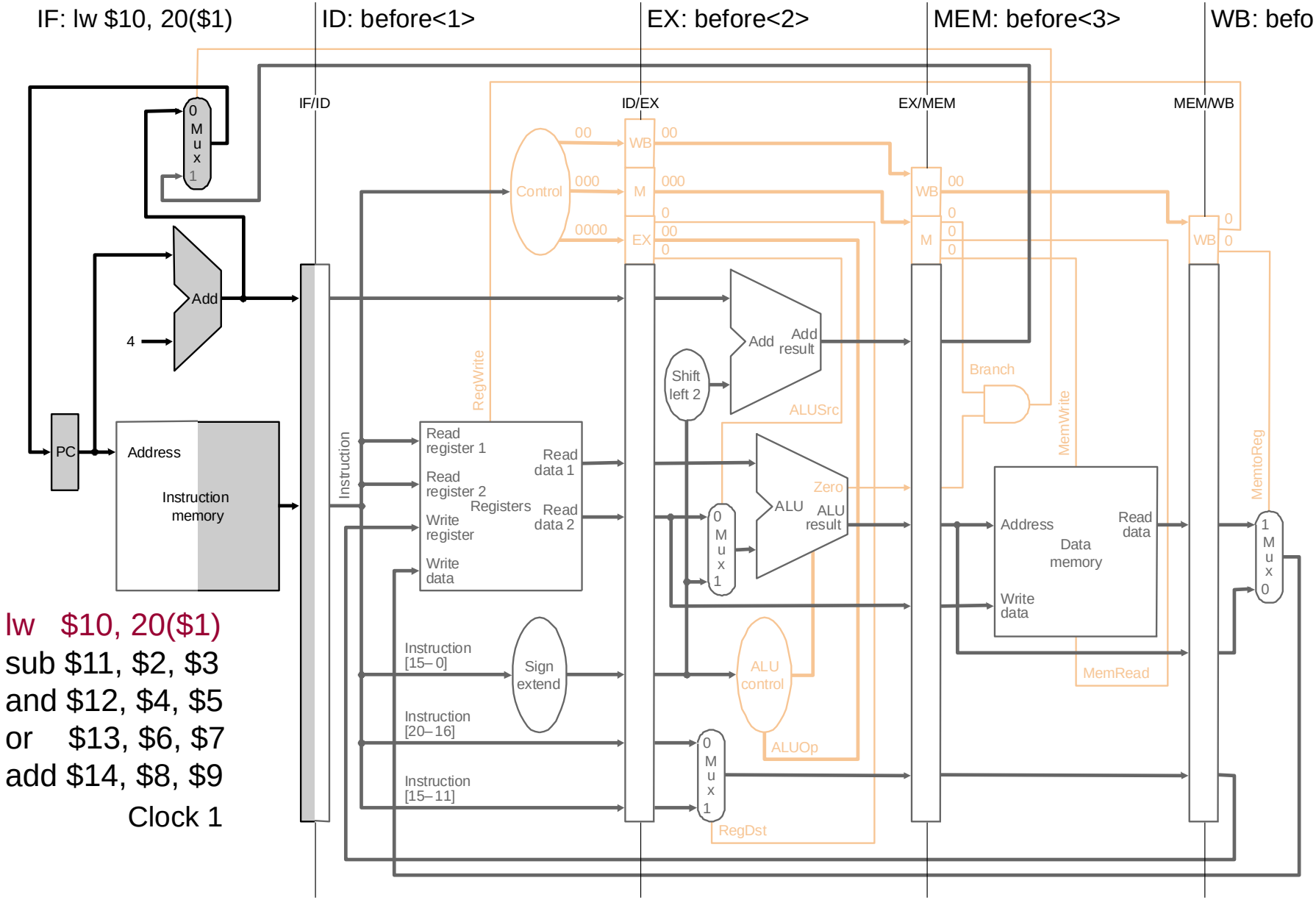
- Example complex sequence
- Detailed walkthrough
  - No dependences
  - Load dependence
  - Branch instruction

© Hill, Shen, Lipasti

Pipeline Walkthrough with controls

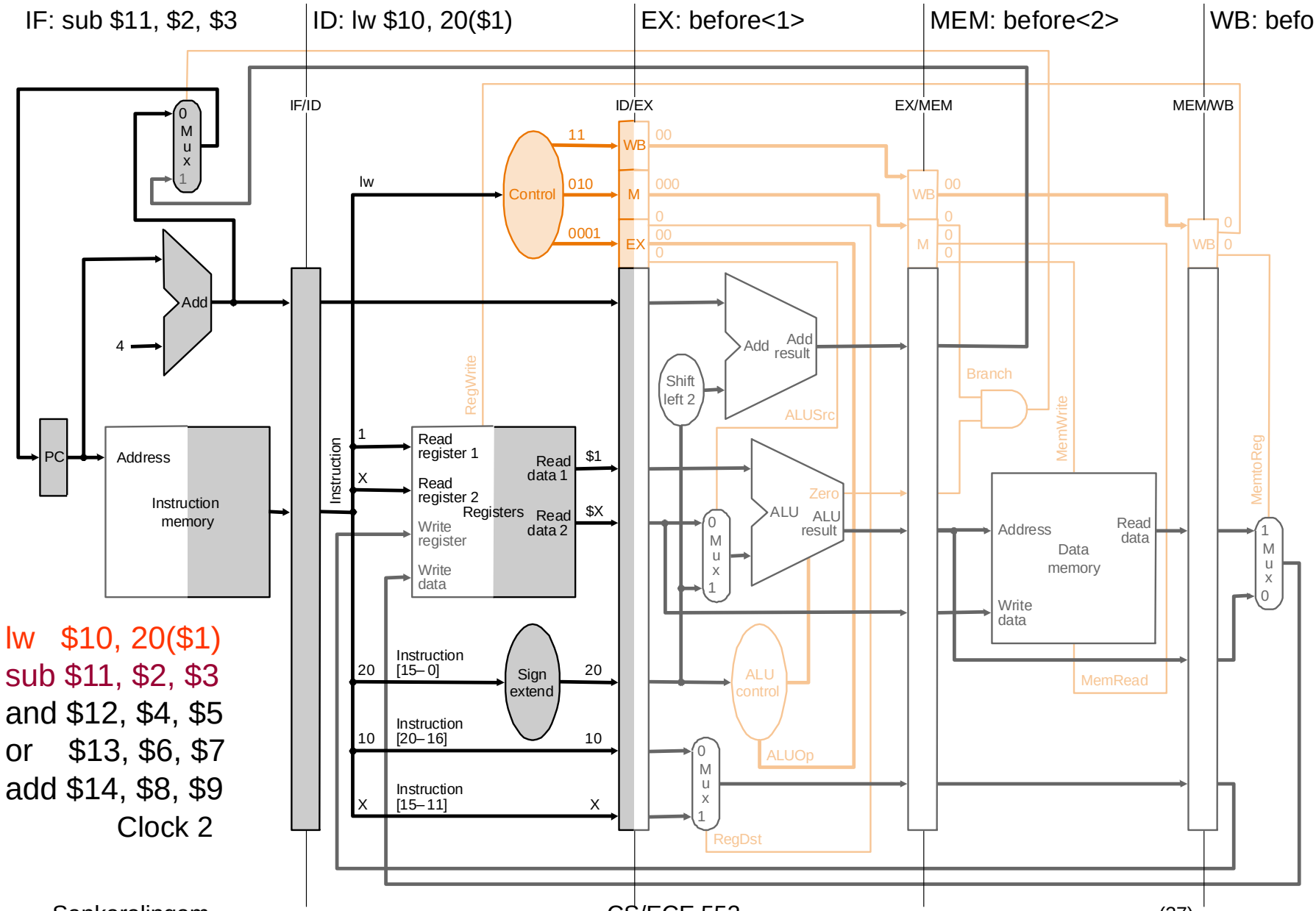
- Use walkthrough worksheets
  - Use code segment shown
  - Fill in controls
  - Interesting stages
    - Controls generated in Decode stage
    - Controls consumed in subsequent stages
  - NOTE: THERE ARE INTENTIONALLY NO DATA DEPENDENCES OR BRANCH INSTRUCTIONS IN SNIPPET

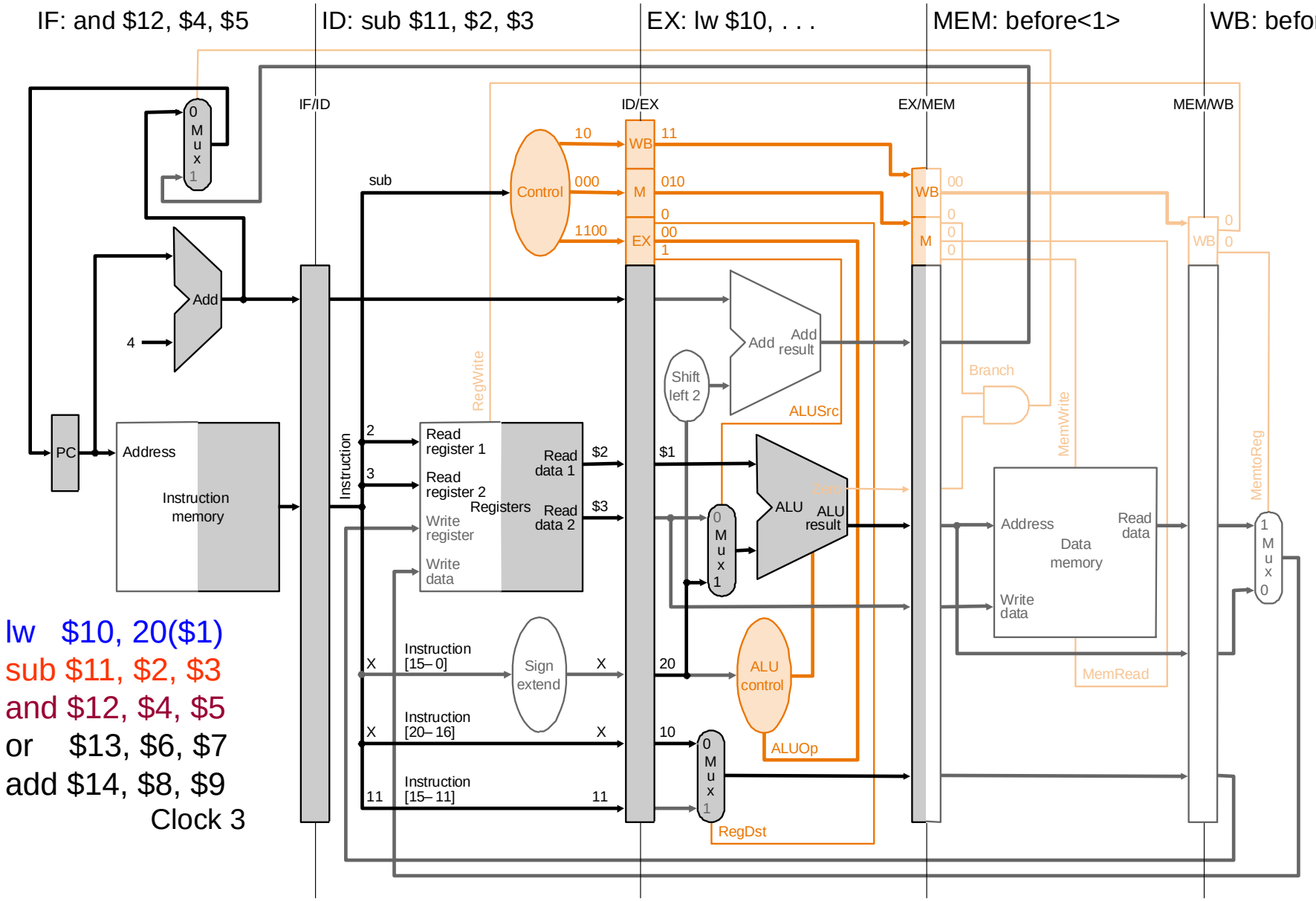
```
lw   $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or    $13, $6, $7
add $14, $8, $9
```

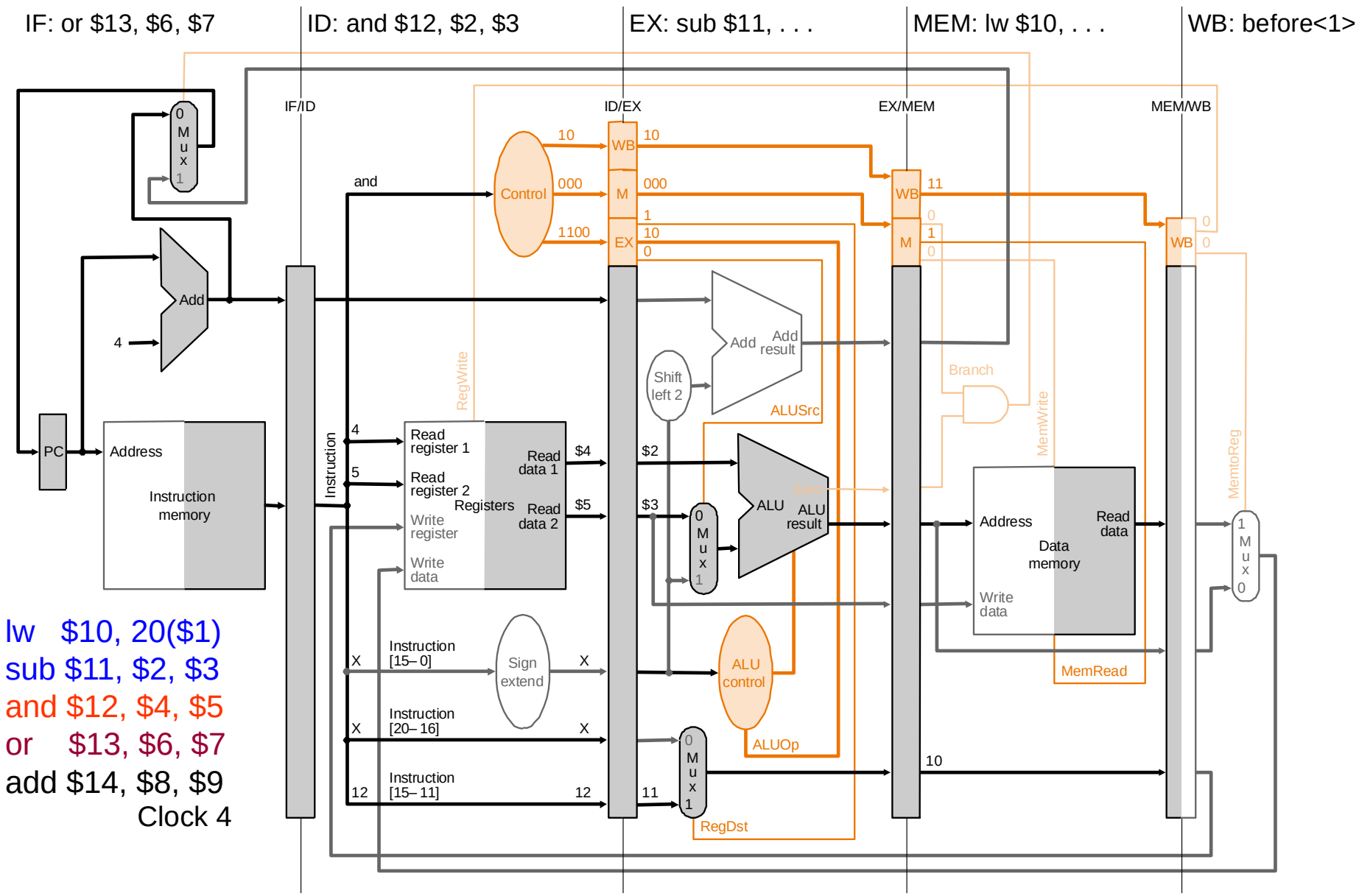


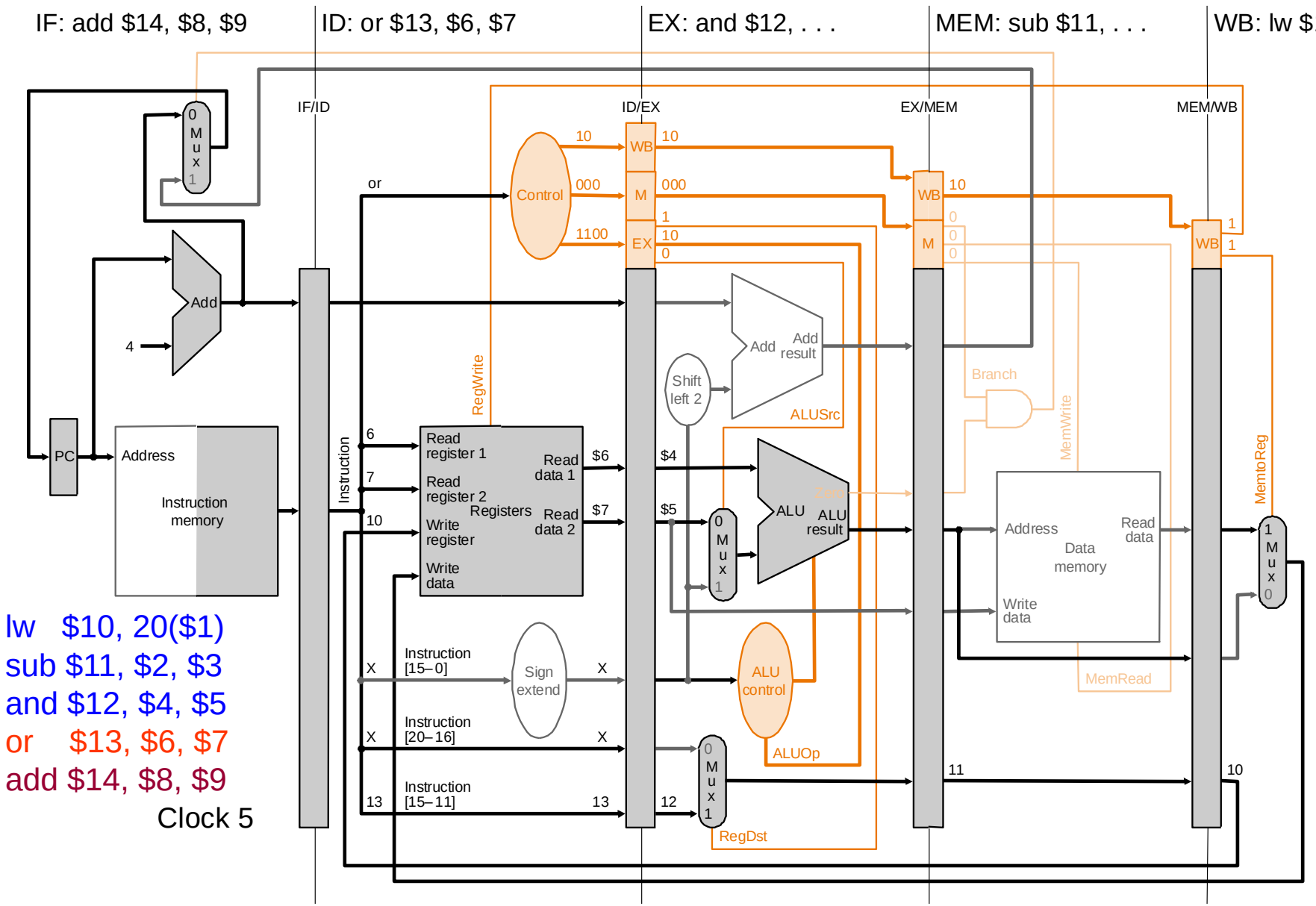
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

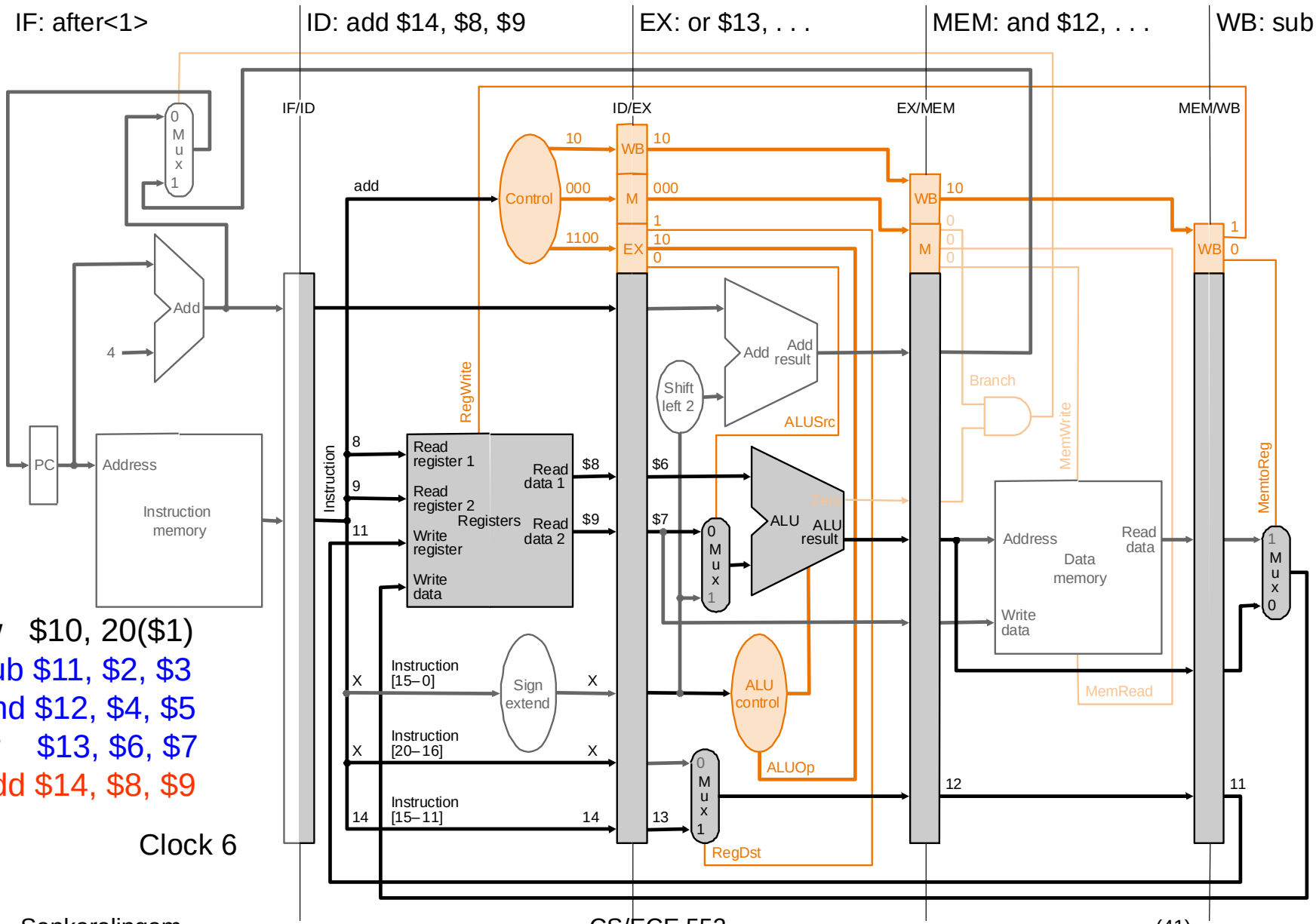
Clock 1

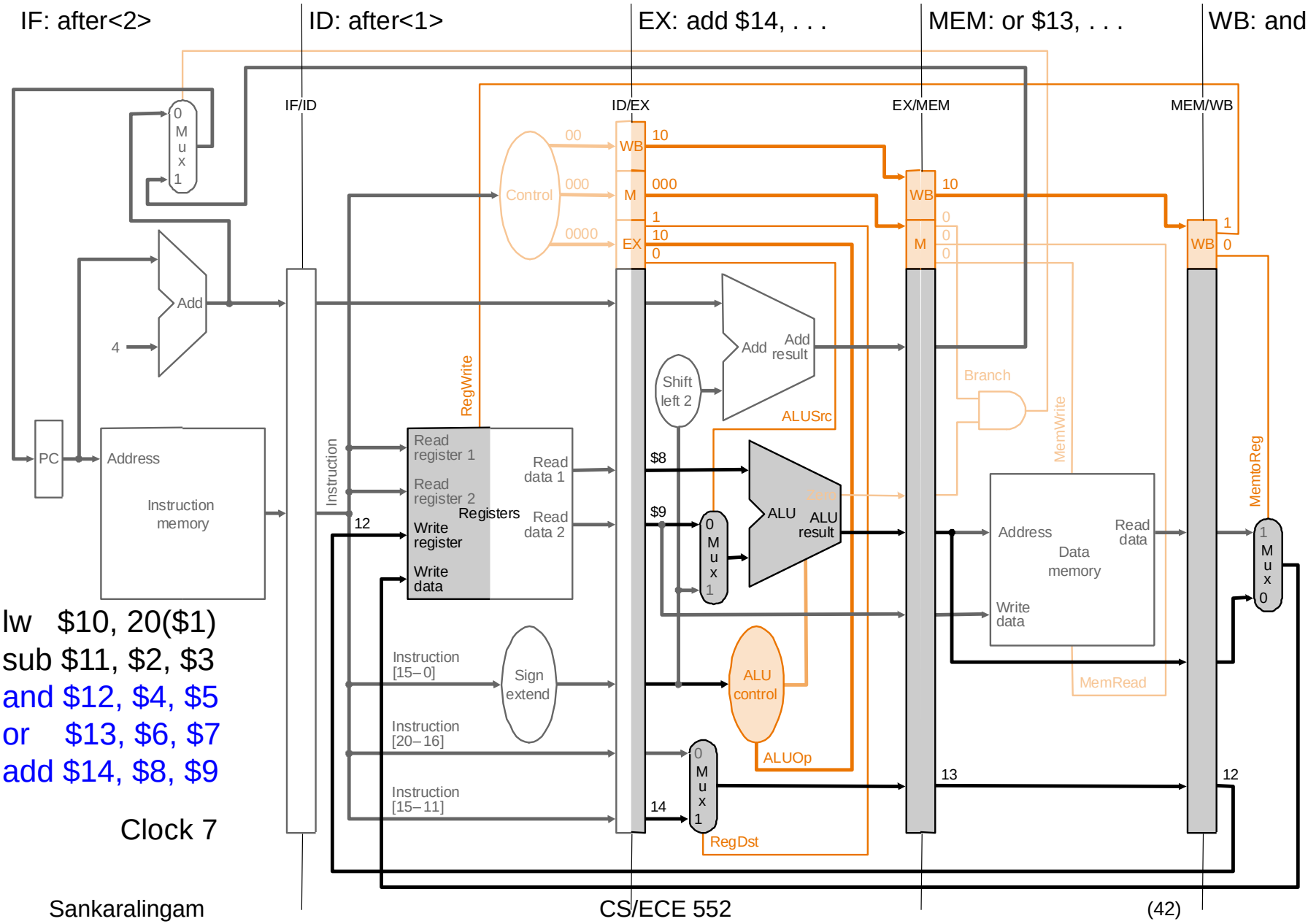

IF: sub $11, $2, $3
ID: lw $10, 20($1)
EX: before<1>
MEM: before<2>
WB: before<3>
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 2
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Control
11
010
0001
WB
M
EX
Shift left 2
Add result
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
ALU control
ALUOp
RegDst
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Address
Data memory
Read data
Write data
MemRead
Sankaralingam
CS/ECE 552
(37)

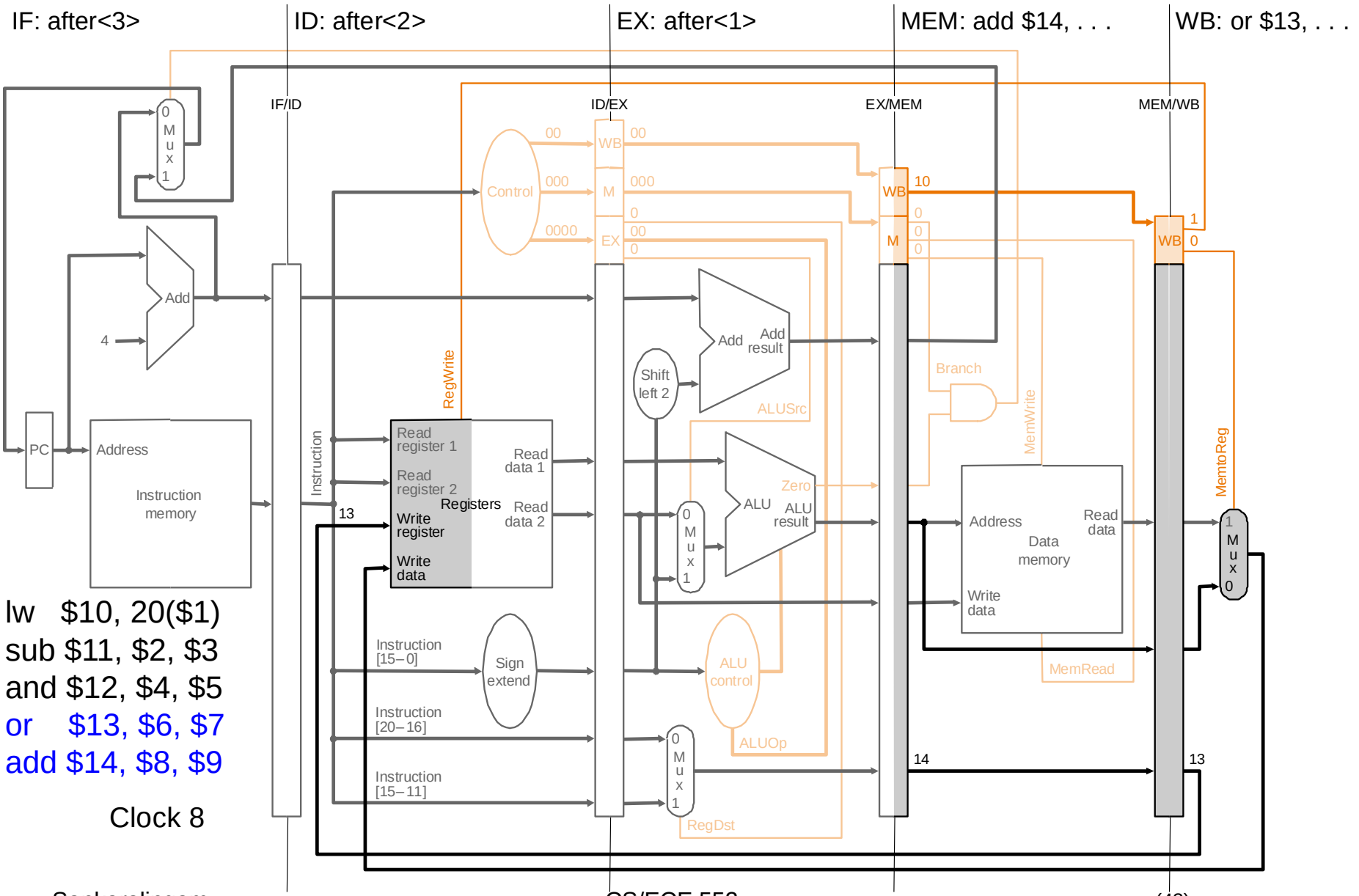

IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
IF/ID
ID/EX
EX/MEM
MEM/WB
sub
Control
10
000
1100
WB
M
EX
11
010
0
00
1
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
2
3
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$2
$3
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]
Sign extend
X
20
ALU control
ALUOp
Instruction [20– 16]
10
Instruction [15– 11]
11
RegDst
Mux
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 3

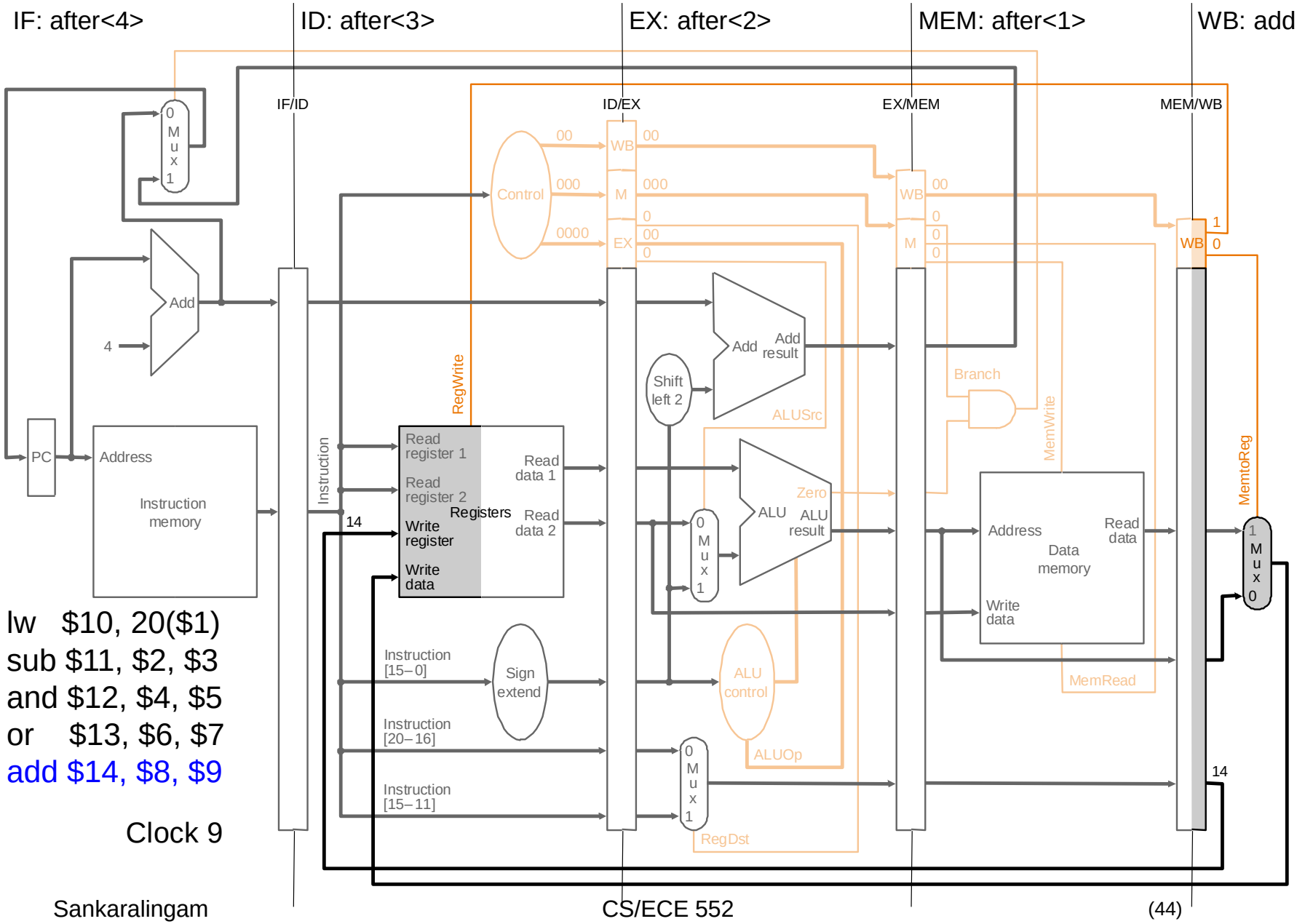

IF: or $13, $6, $7
ID: and $12, $2, $3
EX: sub $11, . . .
MEM: lw $10, . . .
WB: before<1>
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 4
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
and
Control
10
000
1100
WB
M
EX
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
ALU control
ALUOp
RegDst
Instruction [15– 0]
Instruction [20– 16]
Instruction [15– 11]
Sign extend
$4
$5
$2
$3
11
12
10


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
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 5
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
Control
WB
M
EX
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
MemRead
Data memory
Read data
ALU control
ALUOp
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
RegDst
Mux

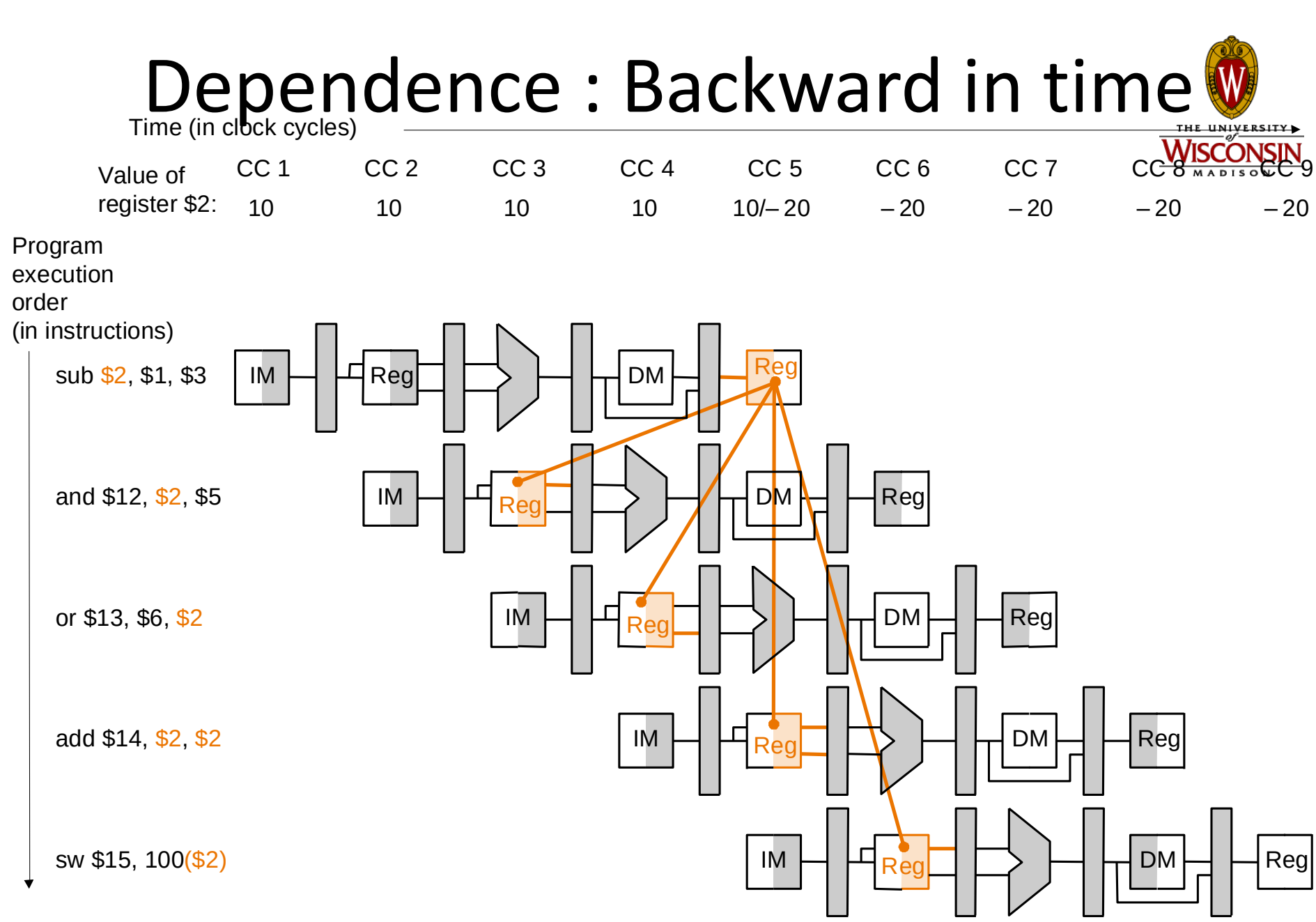

IF: after<1>
ID: add $14, $8, $9
EX: or $13, . . .
MEM: and $12, . . .
WB: sub $11, .
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 6
PC
Address
Instruction memory
Add
4
add
Instruction
RegWrite
Control
WB
M
EX
Shift left 2
Add result
ALUSrc
Branch
MemWrite
MemtoReg
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
ALU result
Zero
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign extend
ALU control
ALUOp
RegDst
Mux


IF: after<2>
ID: after<1>
EX: add $14, . . .
MEM: or $13, . . .
WB: and $12,
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 7
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
12
Control
00
000
0000
WB
M
EX
10
000
1
10
0
Shift left 2
Add
Add result
ALUSrc
$8
$9
ALU
ALU result
Zero
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
Instruction [15–0]
Sign extend
Instruction [20–16]
Instruction [15–11]
ALU control
ALUOp
14
RegDst
13
12
Mux


IF: after<3>
ID: after<2>
EX: after<1>
MEM: add $14, . . .
WB: or $13, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 8
PC
Address
Instruction memory
Add
4
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
13
Control
WB
M
EX
00
000
0000
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
MemRead
Address
Data memory
Read data
Write data
MemtoReg
Instruction [15–0]
Sign extend
Instruction [20–16]
Instruction [15–11]
ALU control
ALUOp
RegDst
14
13
Sankaralingam
CS/ECE 552
(43)


IF: after<4>
ID: after<3>
EX: after<2>
MEM: after<1>
WB: add $14, .
IF/ID
ID/EX
EX/MEM
MEM/WB
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 9
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
00
000
0000
WB
M
EX
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
Sign extend
Instruction [20–16]
Instruction [15–11]
ALU control
ALUOp
RegDst
Mux

Dependence : Backward in time

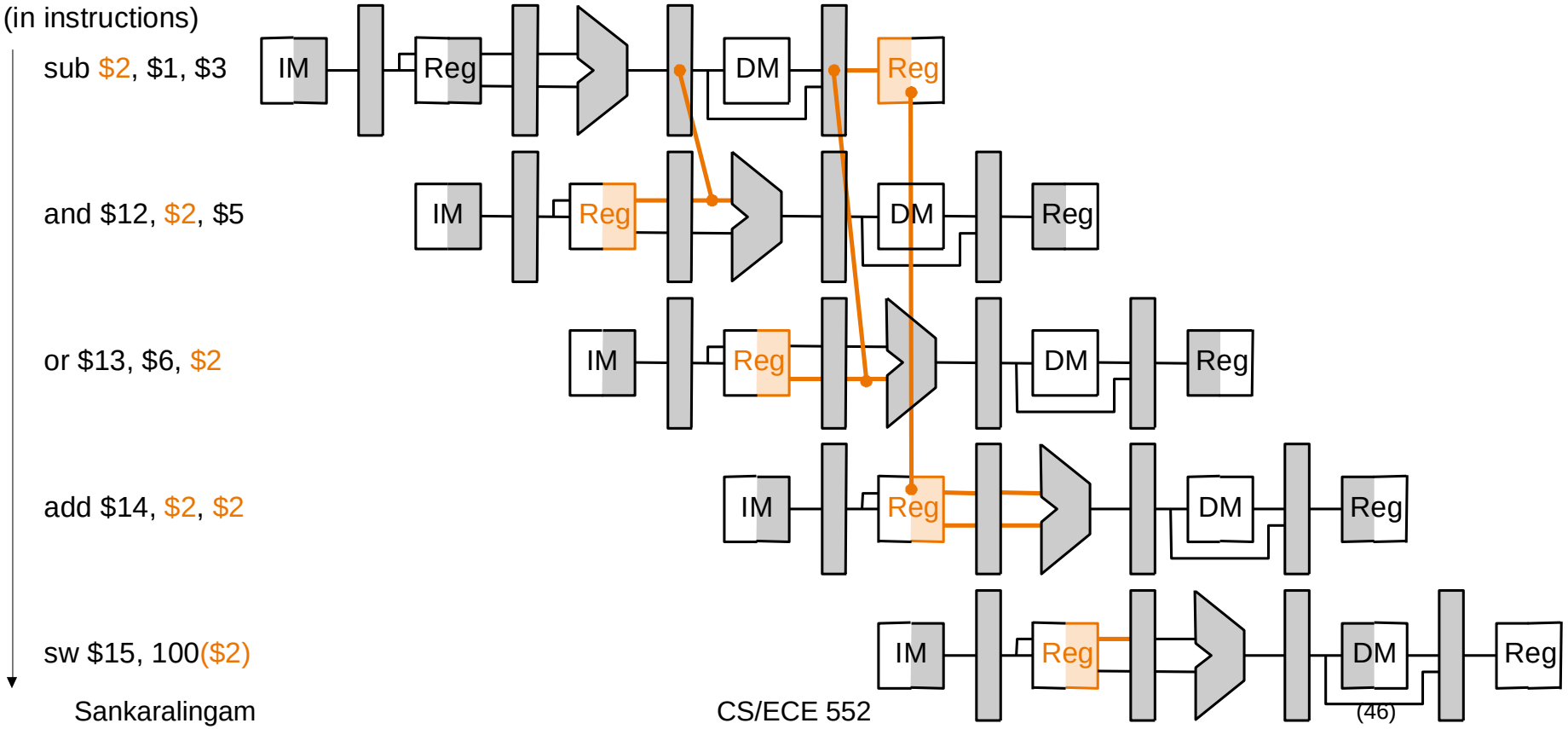


True dependence : Forward in time

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)

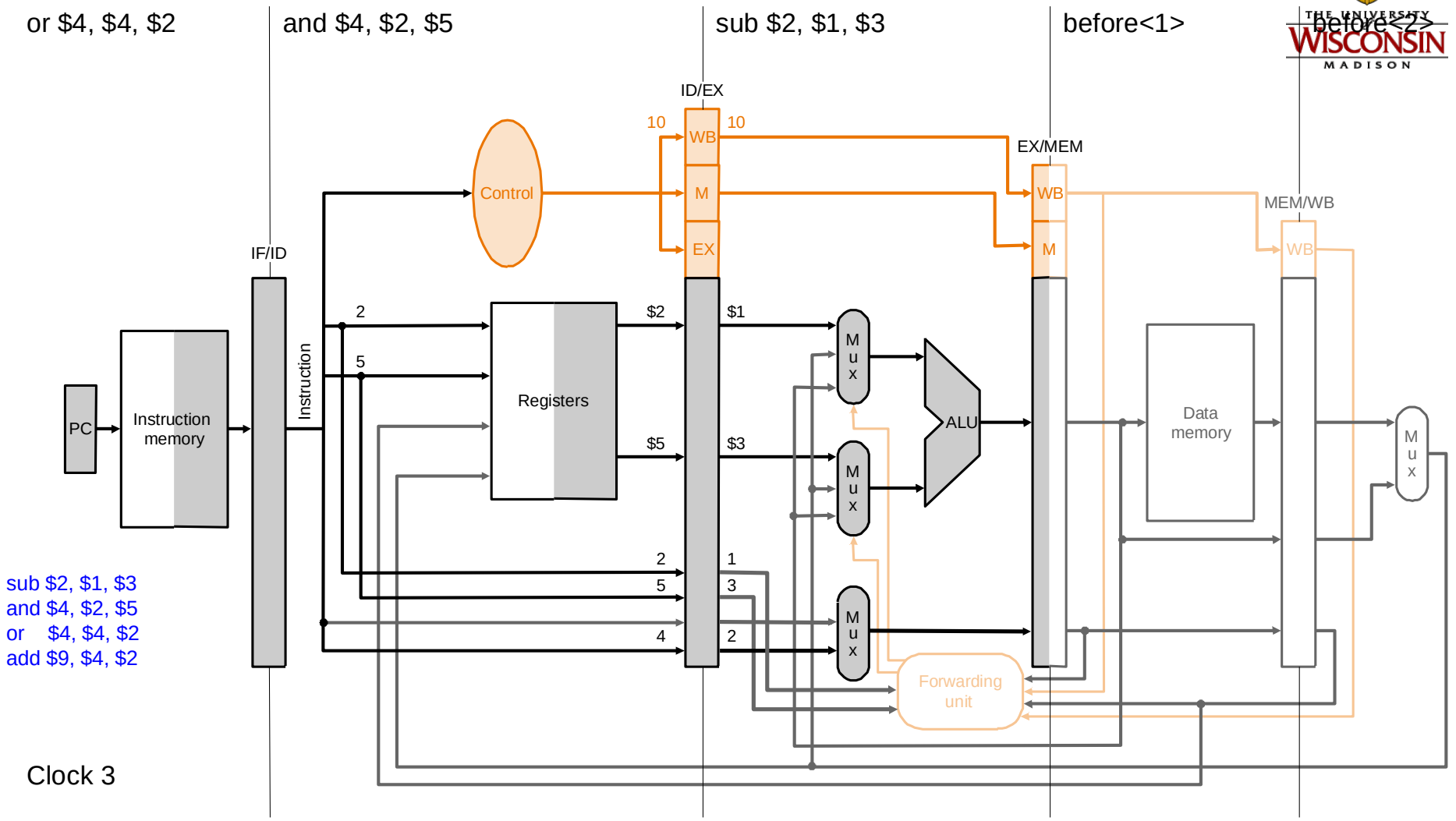


Data Hazards and Forwarding: Walkthrough

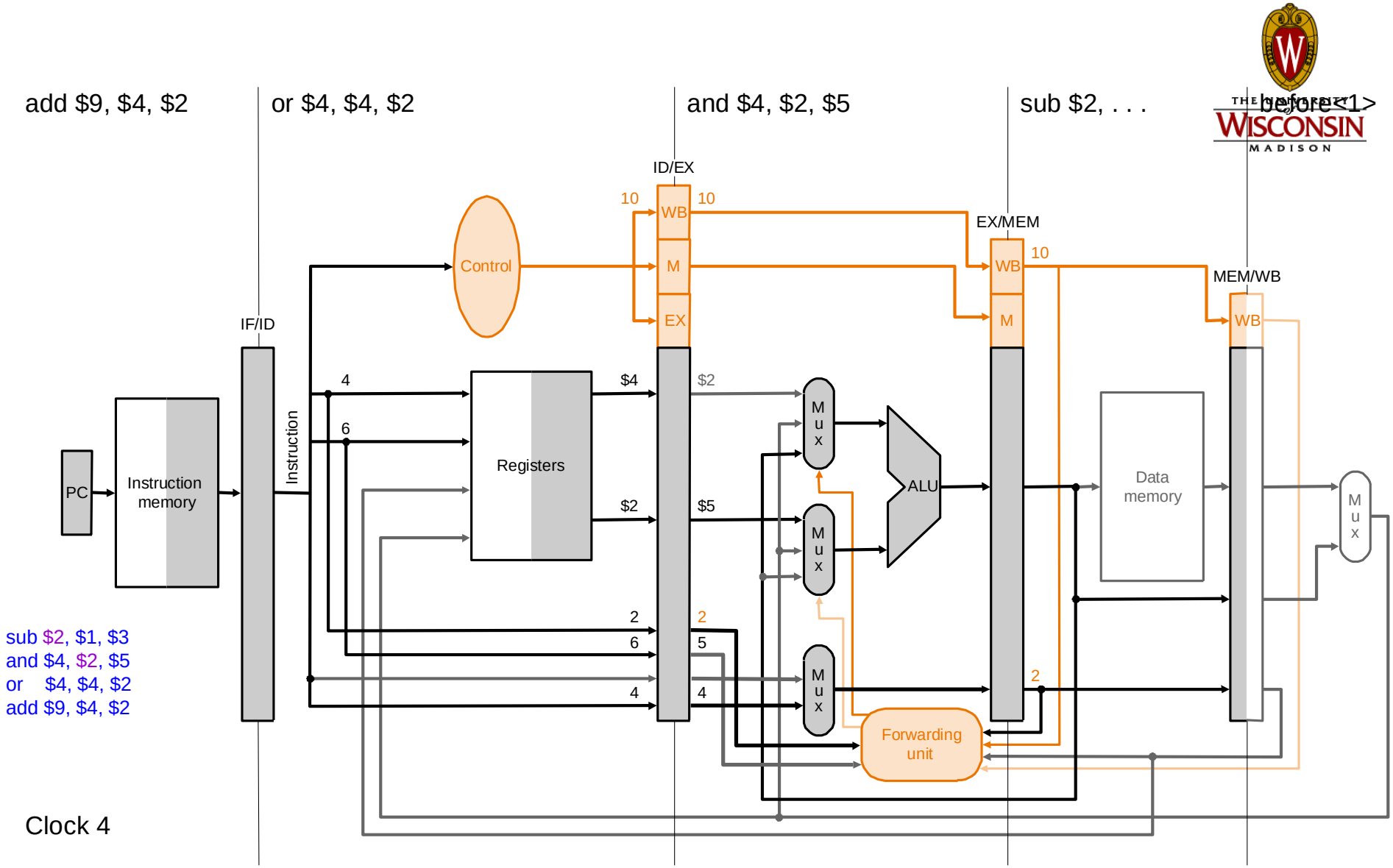
- Code snippet
  - identify hazards
  - identify forwarding paths
- NOTE:
  - DATA DEPENDENCES ON REG $2, REG $4
  - ALSO TWO INSTRUCTIONS WRITE TO REG $4

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

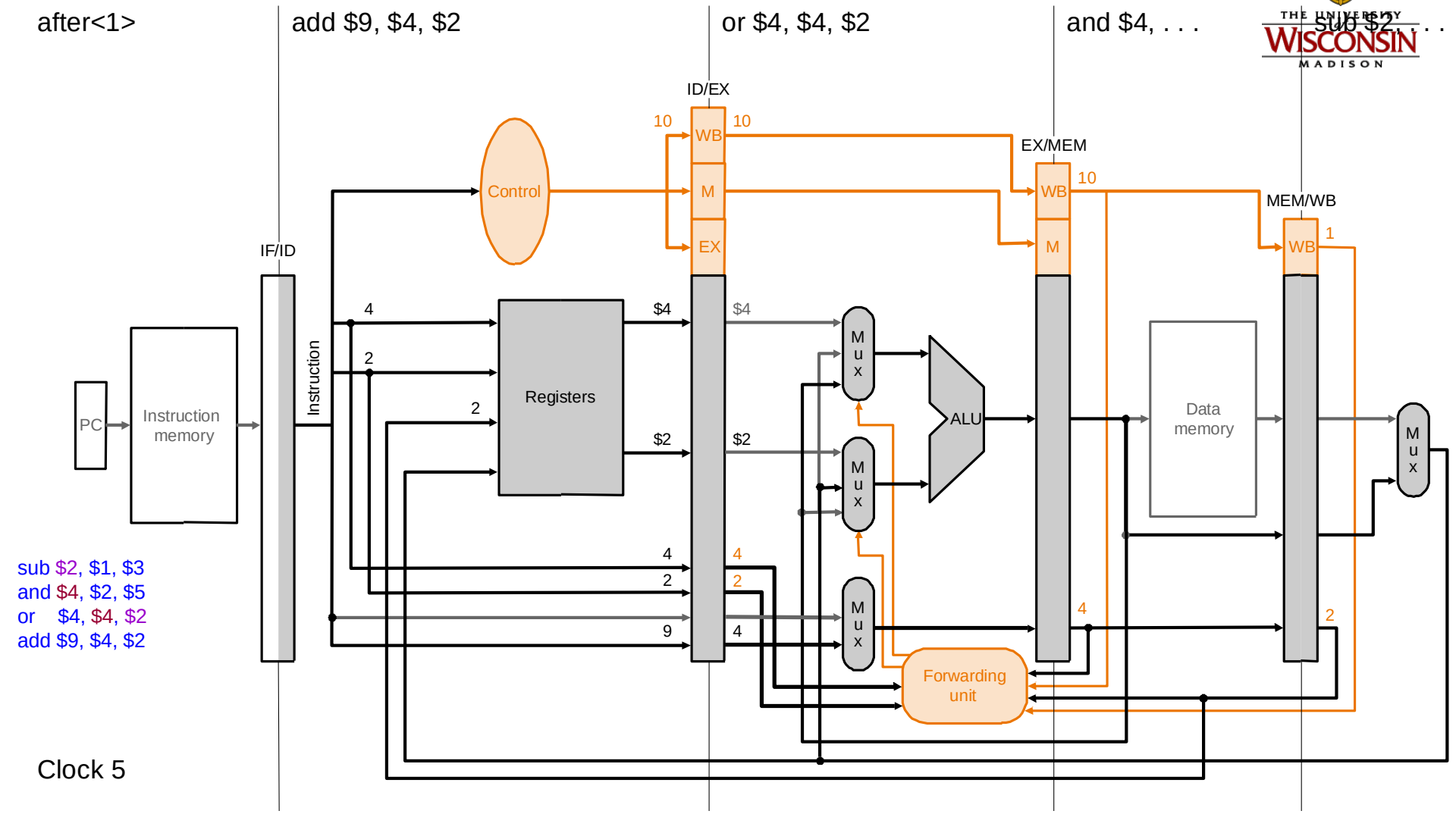
Walkthrough



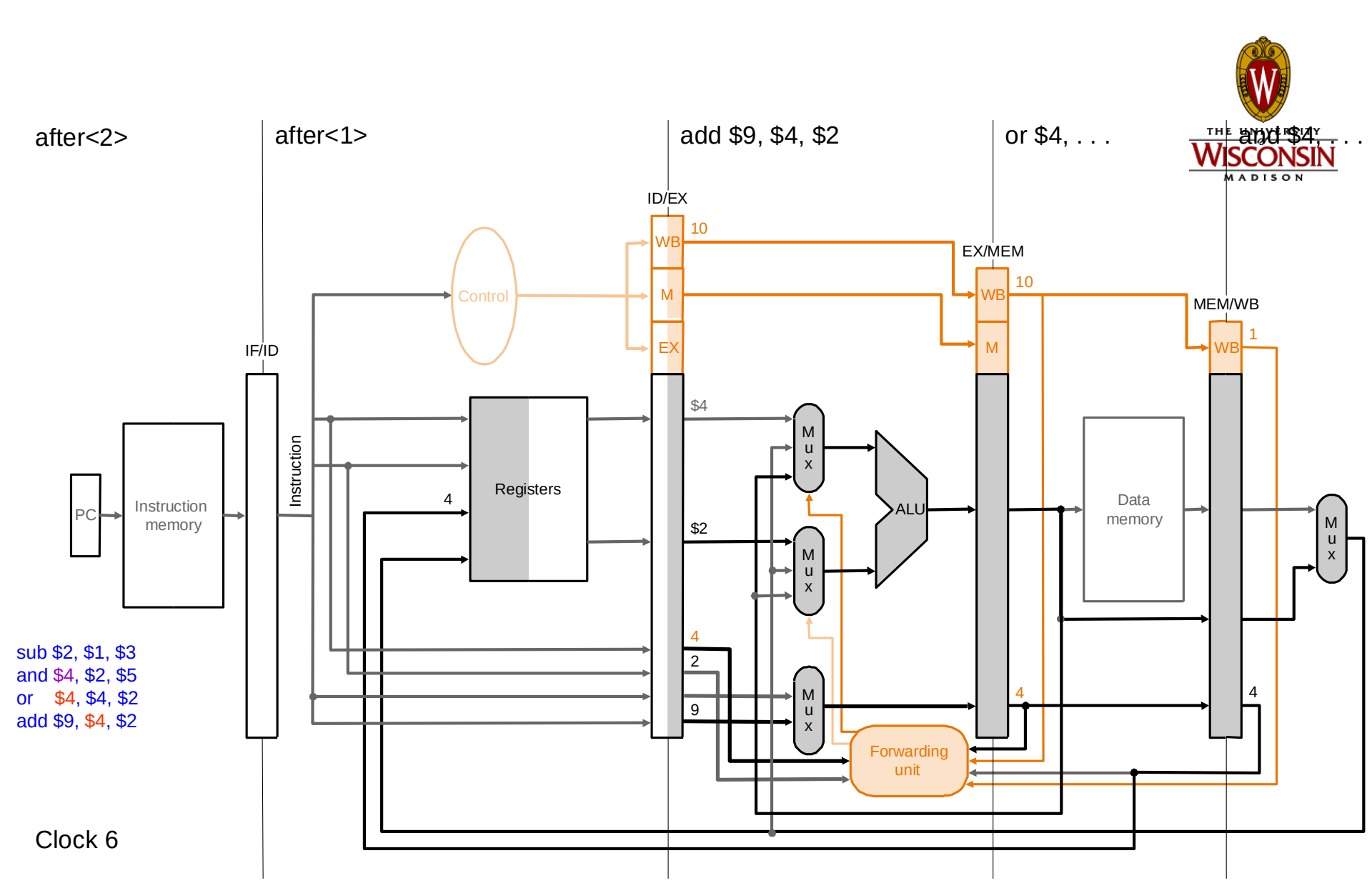
- Skip the boring stuff, jump to cycle 3



- Forward ALUOut to Operand 1

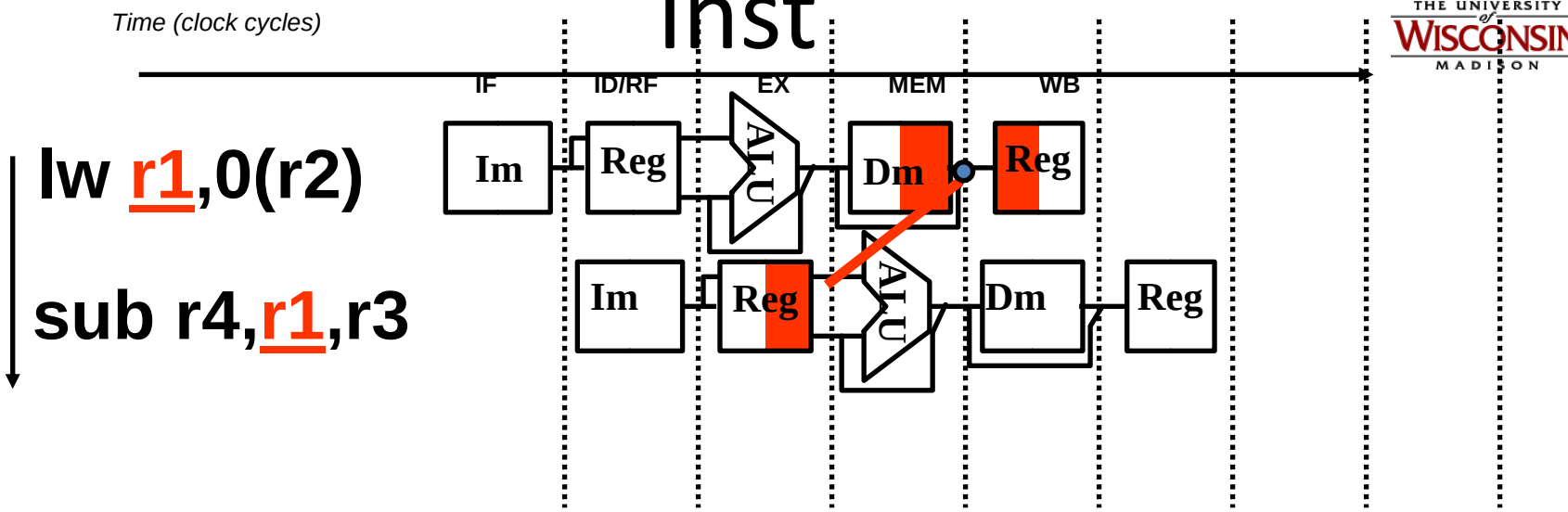


- Forward ALUout to Op1, Mem to Op2



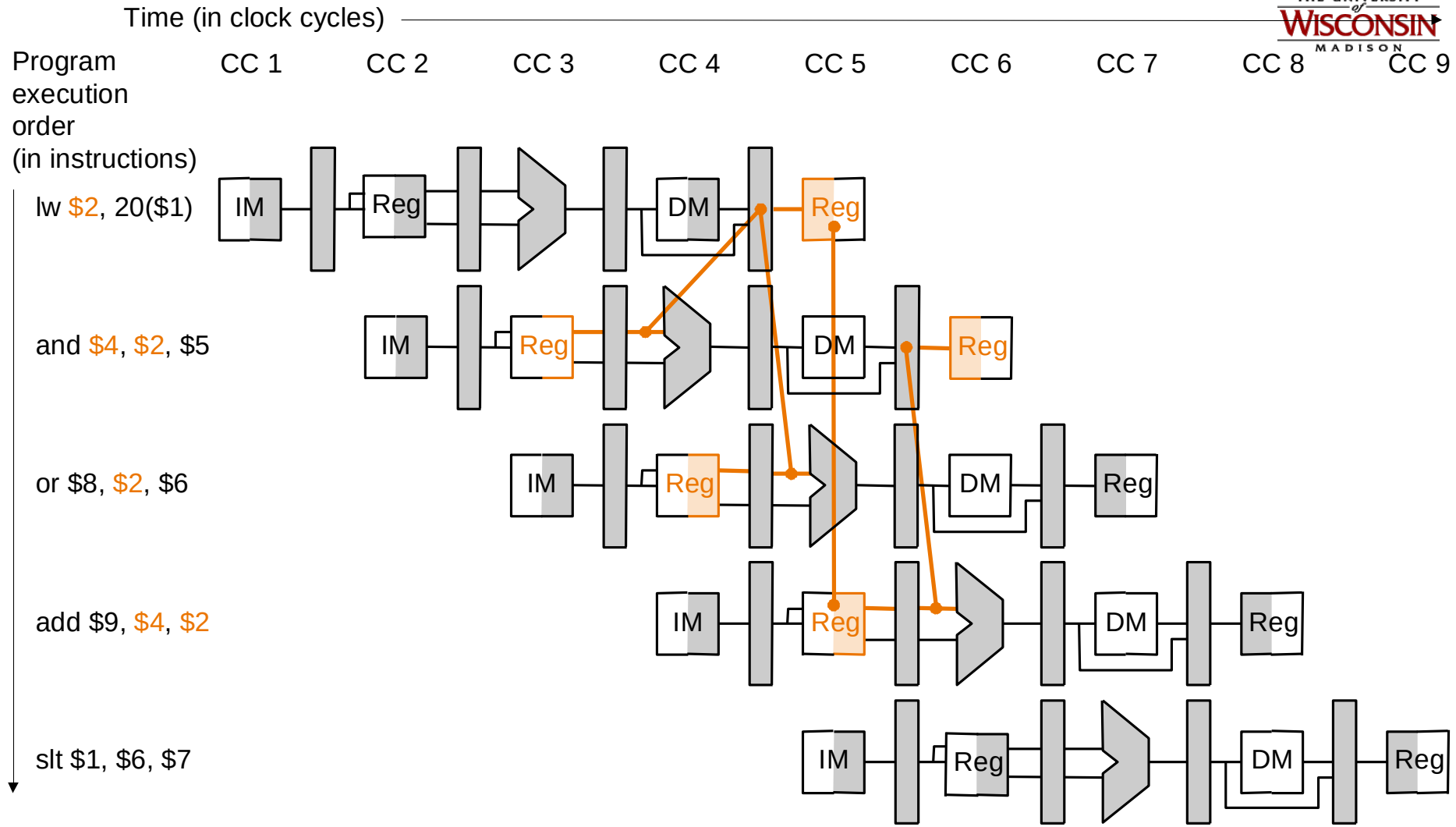
- Two candidates match, forward the latest

Lookahead: RAW hazard with load inst



- Forwarding as solution to RAW hazard
  - possible if no (true) dependence going backwards in time
  - True for R-type instructions
    - Data available after EX stage (i.e., at ALUOut)
  - Not true for load instruction

Load instruction



- Replaced "sub" with "lw" in previous code-example

Hazards with load instruction

- True dependencies: backward in time
- Stall the pipeline

- Minor change in terminology
  - If forwarding can solve it, it is not a hazard!
  - "Hazard" refers only to true backward dependencies in time.

Detection

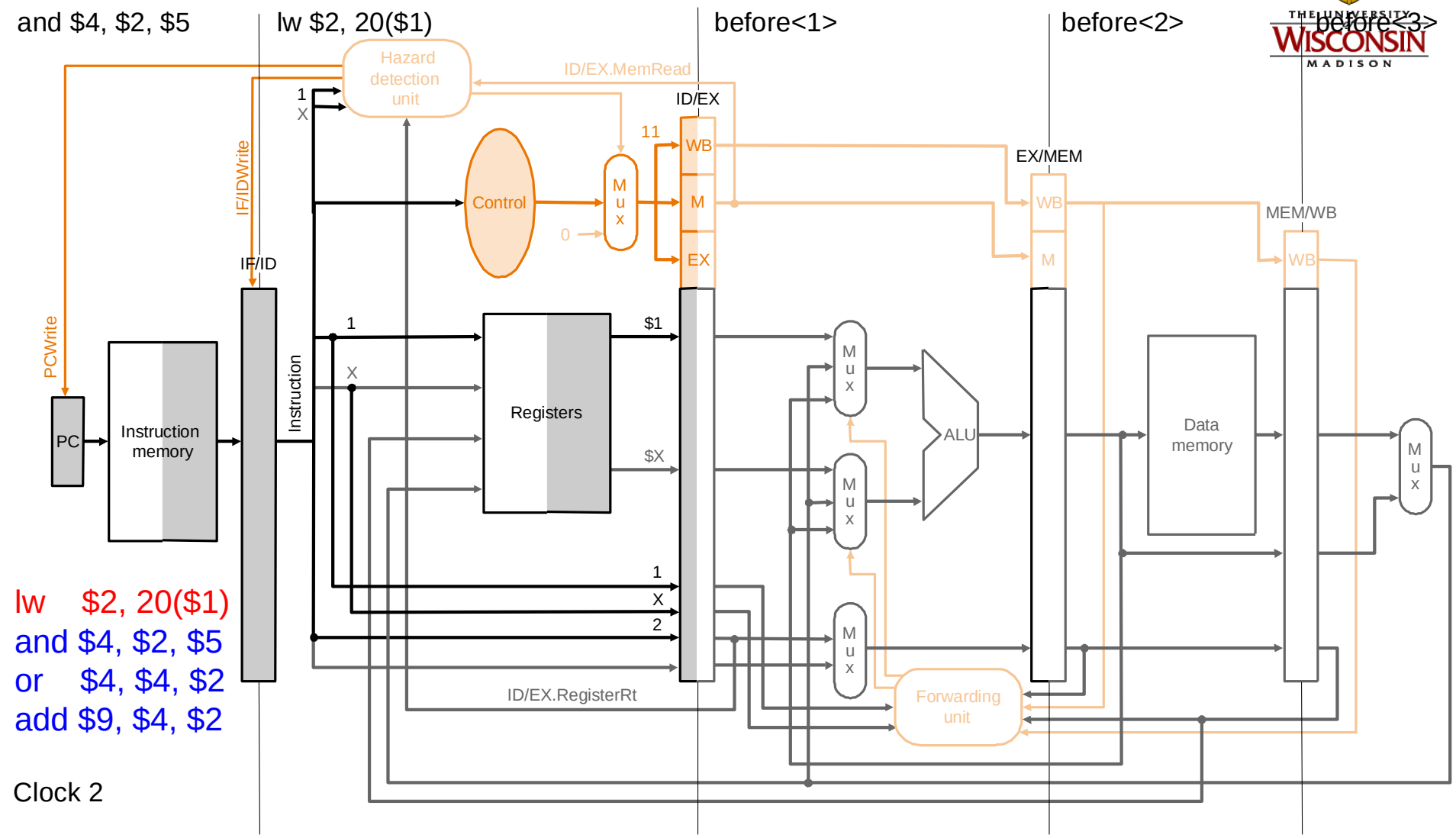
- Conditions
  - Preceding instruction must read memory
    - MemRead must be asserted
  - Destination of preceding instruction (rt) must be one of operands of current instruction
- Logic equations– restate above conditions formally
  - If(ID/EX.MemRead AND
    ((ID/EX.RegRt = IF/ID.RegRs) OR
    (ID/EX.RegRt = IF/ID.RegRt)))
    STALL

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

Stalling the pipeline

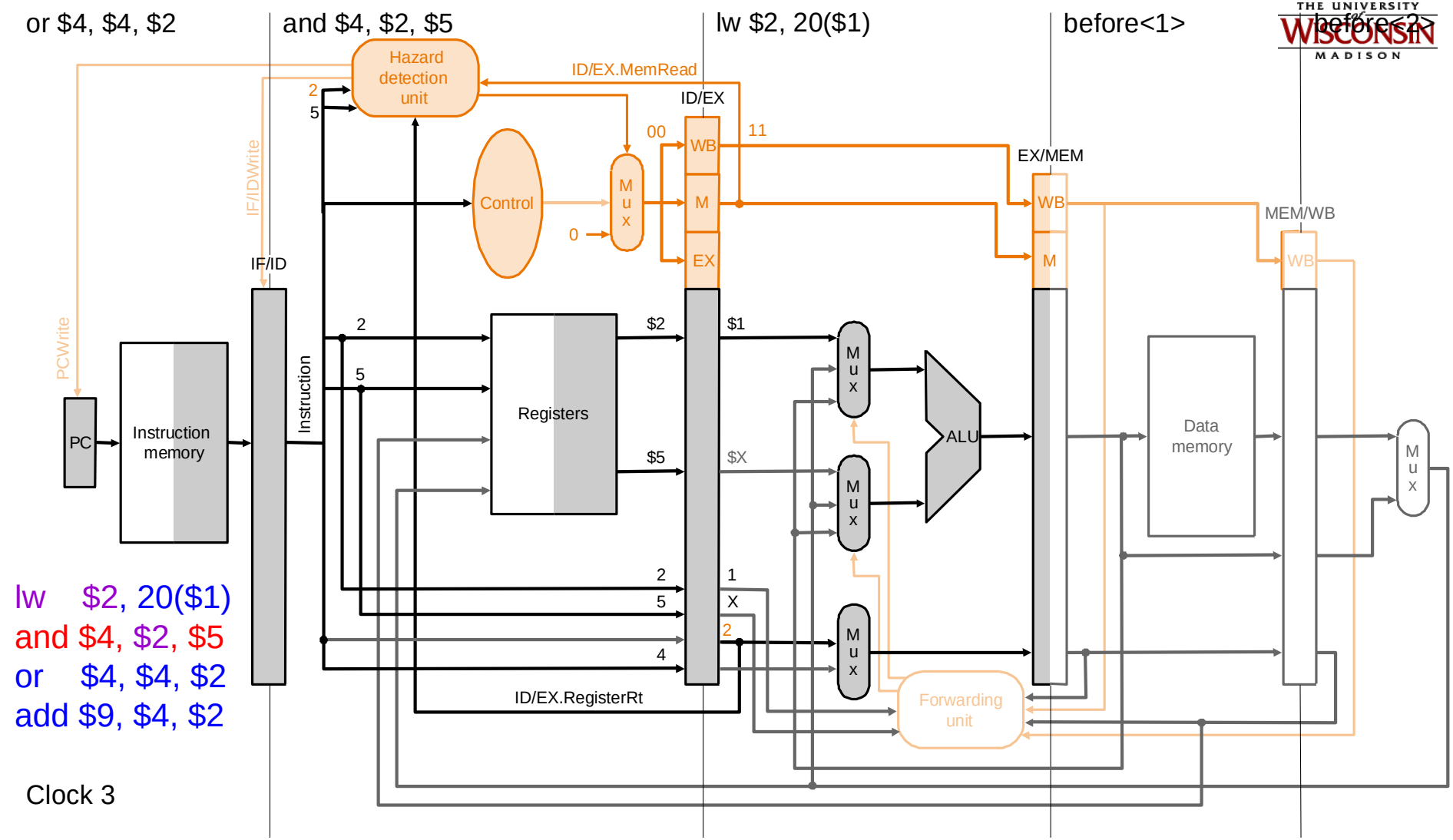
- Instruction cannot proceed
  - Following instruction must be stalled too.
  - Otherwise state in pipeline registers is overwritten
- Preceding instructions may proceed as usual
- Solution
  - inject NOP into EX/Mem pipeline
  - Prevent writes to PC to IF/ID register

Walk-through (1 of 6)



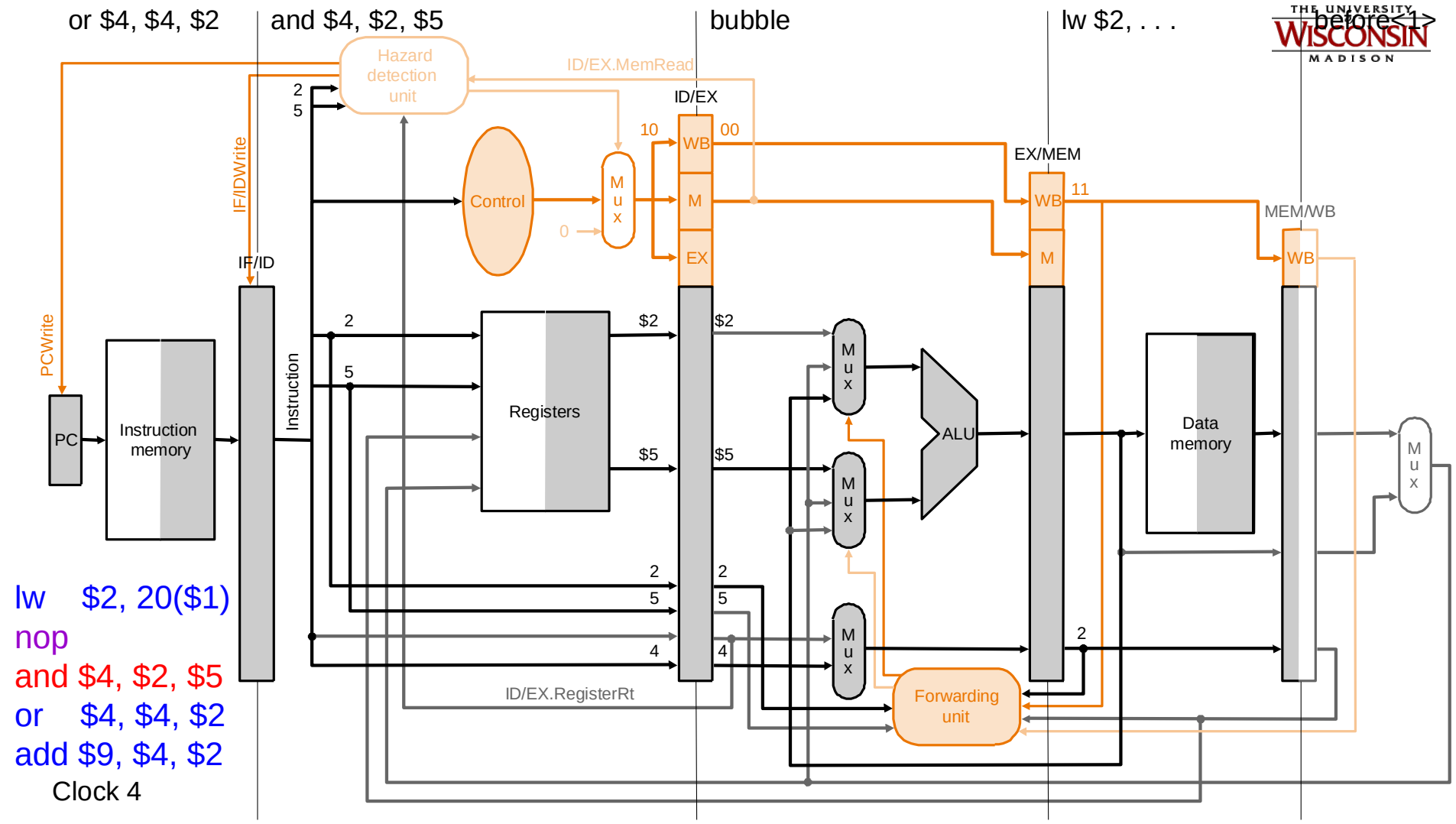
- Skip to cycle 2

Walk-through (2 of 6)

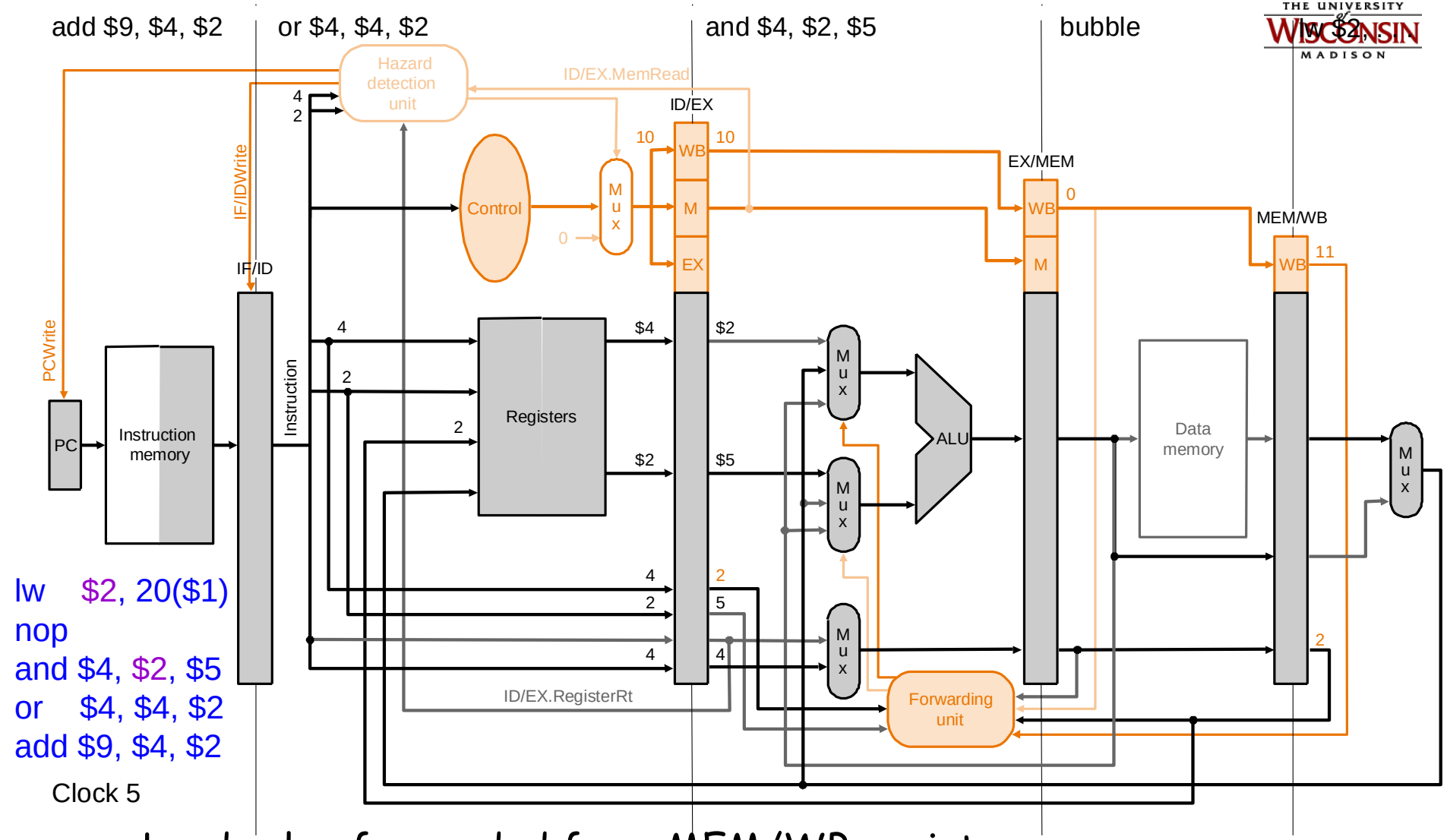


- All '0's => NOP (MemWr, RegWr, deasserted)

Walk-through (3 of 6)

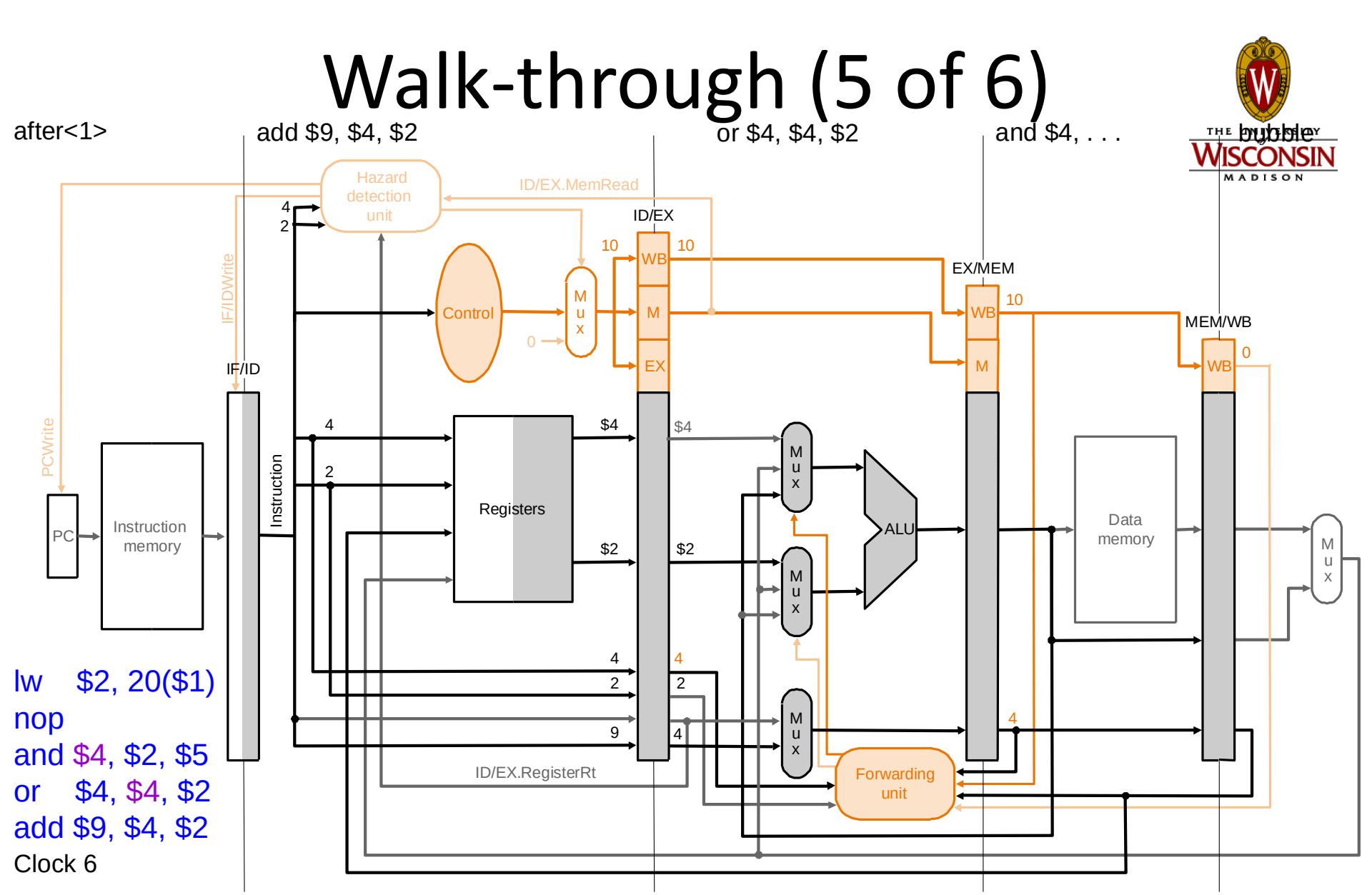


Walk-through (4 of 6)



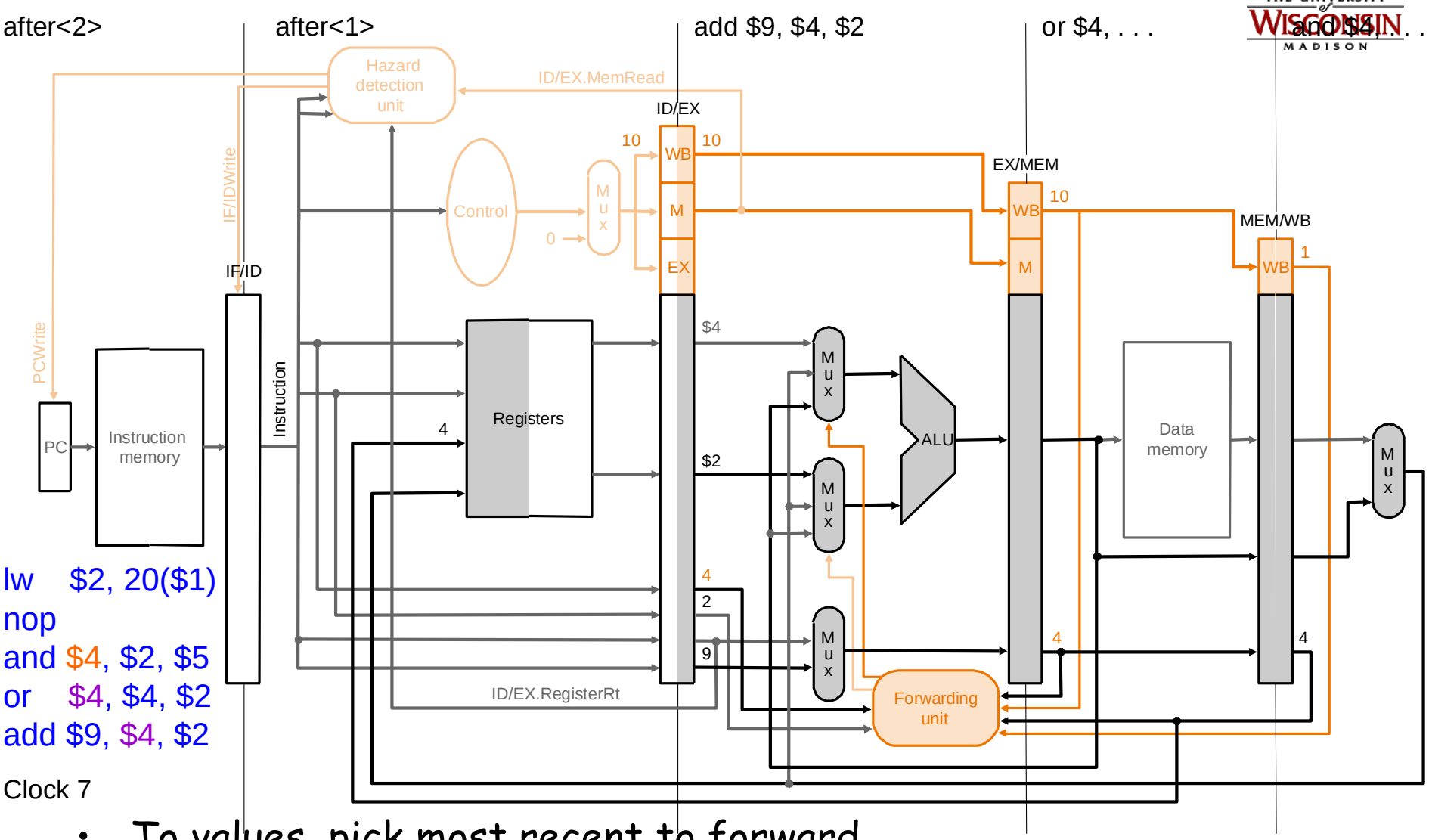
- Load value forwarded from MEM/WB register

Walk-through (5 of 6)



- $4 value forwarded from EX/MEM register

Walk-through (6 of 6)



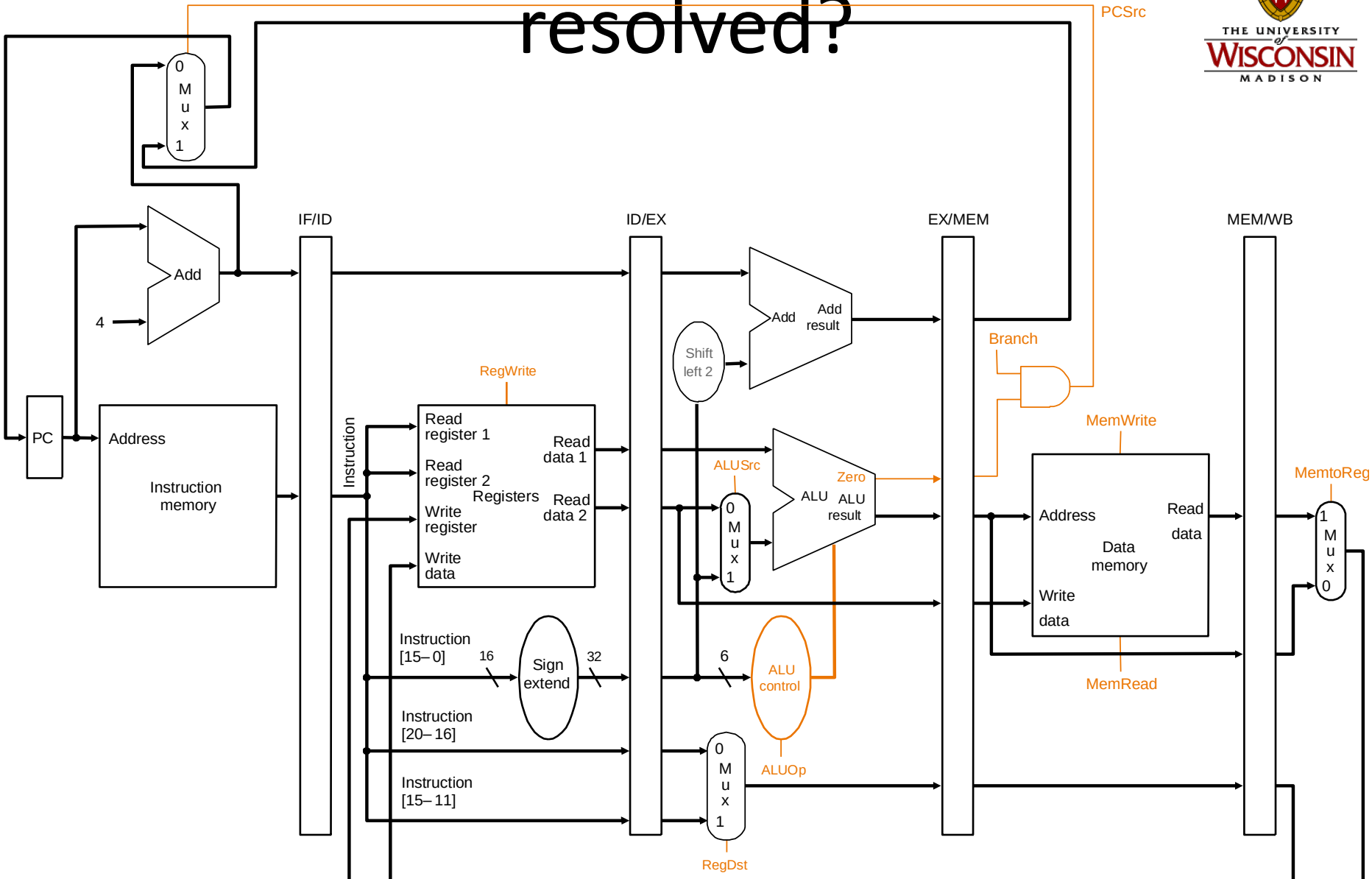
- To values, pick most recent to forward

RAW Hazard with Loads: Summary

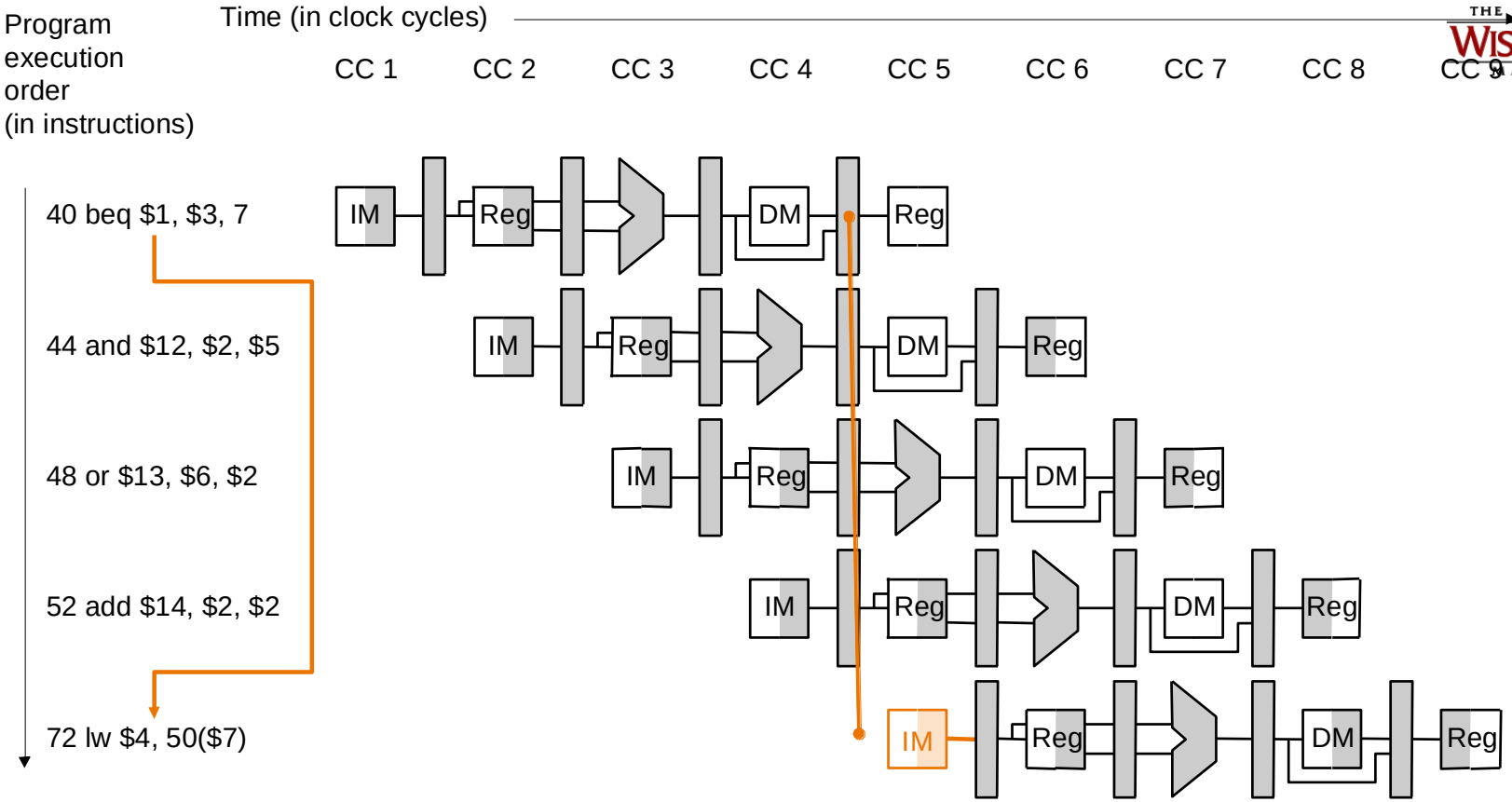
- True backward dependencies in time
  - Need to stall

- Stall achieved by
  - Detecting hazard (remember logic equation)
  - Inserting NOP (all EX/MEM/WB controls set to 0)
  - Preventing IF/ID register and PC from being overwritten
- Next Branch/Control Hazards

When conditional branches resolved?



Branch Hazards

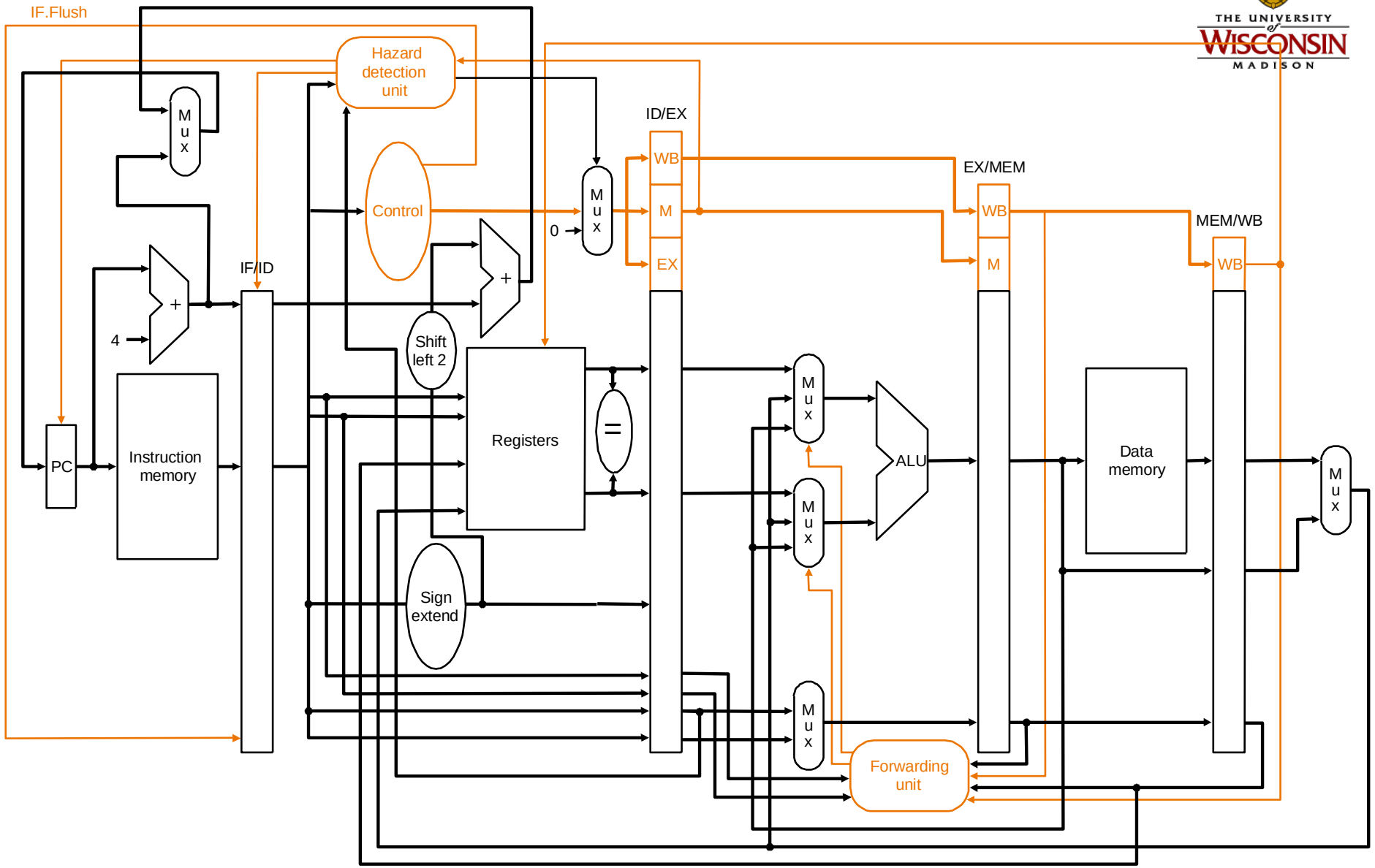


- Branch resolved in the MEM stage
- If taken,
  - PC<- PC + 4 + SX(Imm*4)
  - 40 + 4 + 7*4 = 72

Control/Branch Hazards

- Branch resolved in the MEM stage
  - But next instruction has to fetched in the next cycle
  - Reduce the penalty by moving decision earlier in pipeline
    - Need additional comparator (r1=r2?) and adder (PC+4+SX(IMM)*4)
  - Reduced penalty from 3 cycles to 1 cycle

Datapath for branch hazards

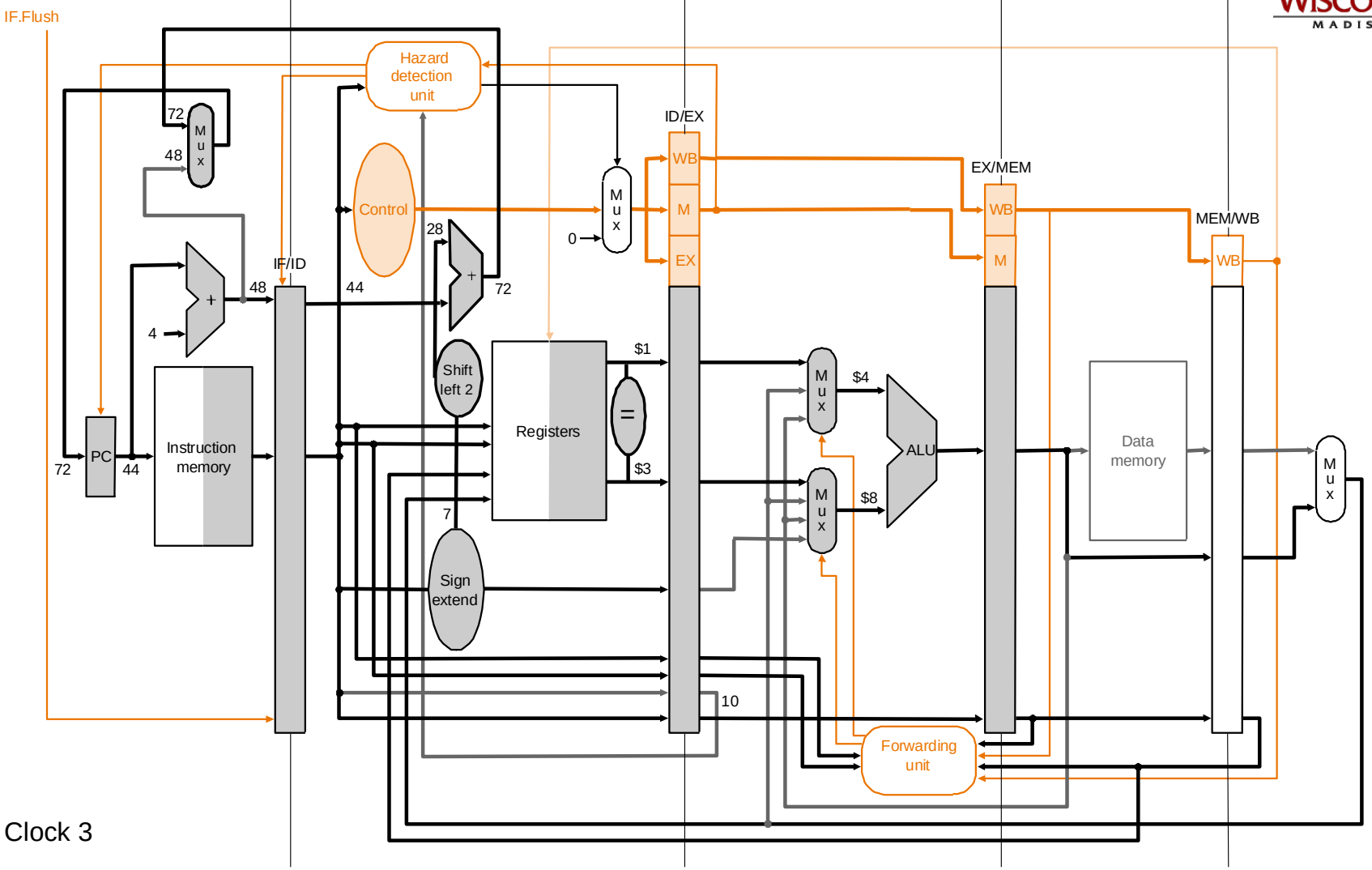


Eliminate 1-cycle stall?

- Two solutions
  - Predict branch is always not taken
    - More sophisticated prediction schemes
  - Delay slots
    - Compiler's problem
- Walkthrough example for solution #1
  - Predict not taken

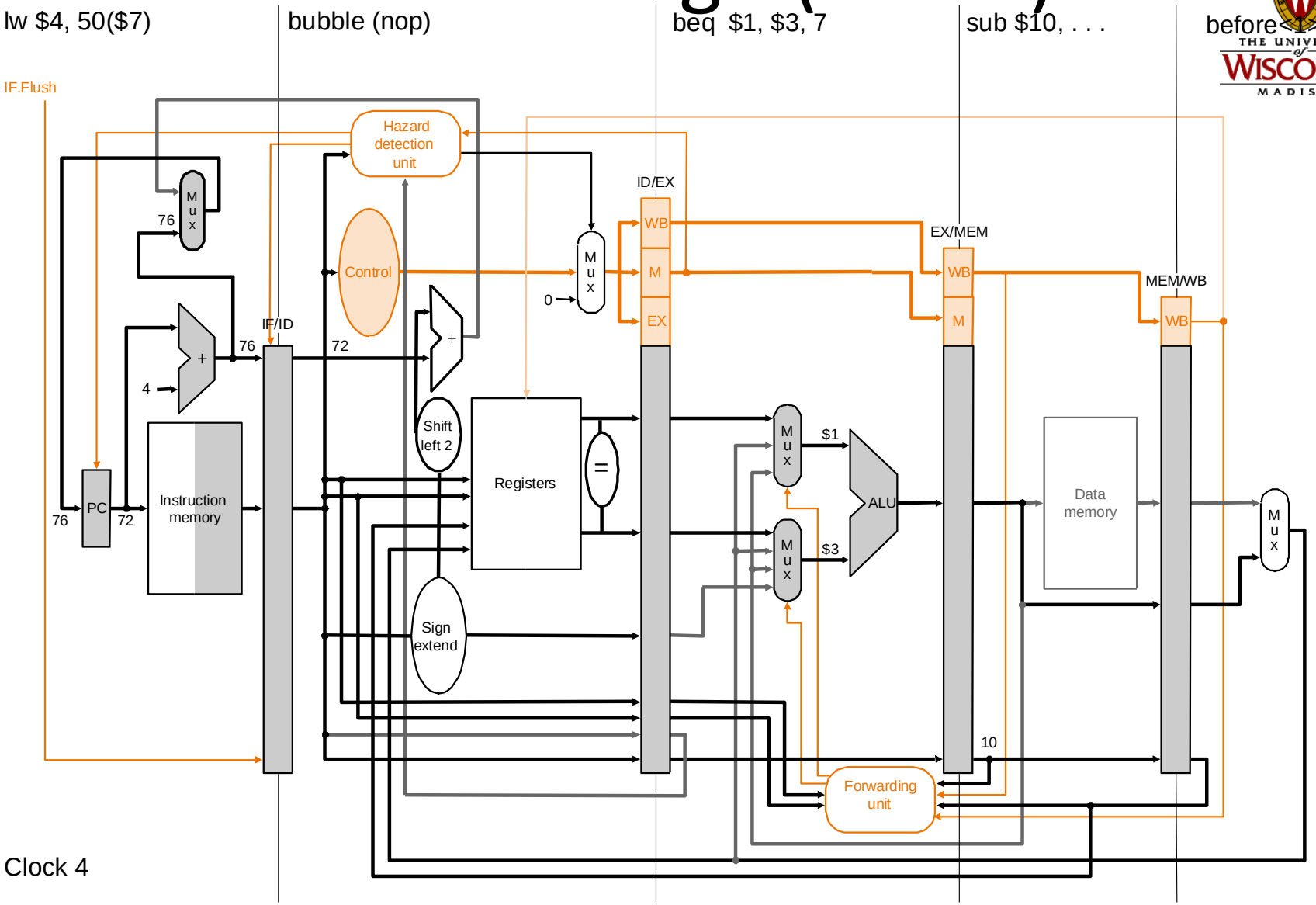
Walkthrough (1 of 2)

and $12, $2, $5 | beq $1, $3, 7 | sub $10, $4, $8 | before<1> | before<2>



Clock 3

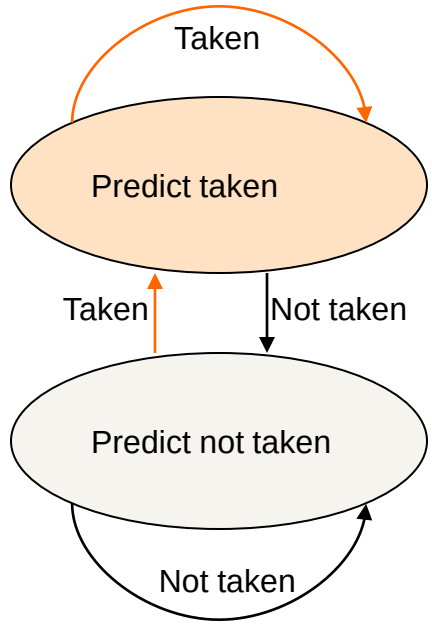
Walkthrough (2 of 2)



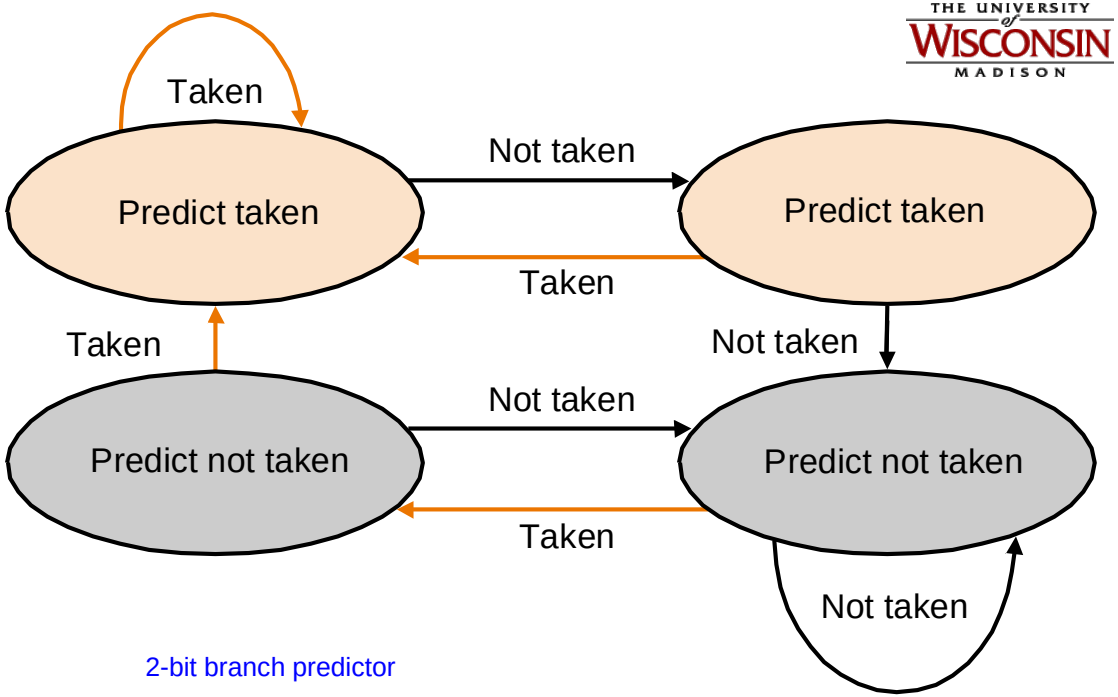
Dynamic Branch Prediction

- Better than static prediction
  - Branches are predictable
  - ~90% of program execution time is spent in ~10% of code (inner loops)
  - Think of a program loop of N iterations
    - Taken N-1 times
    - Not taken last time

Dynamic Branch Prediction



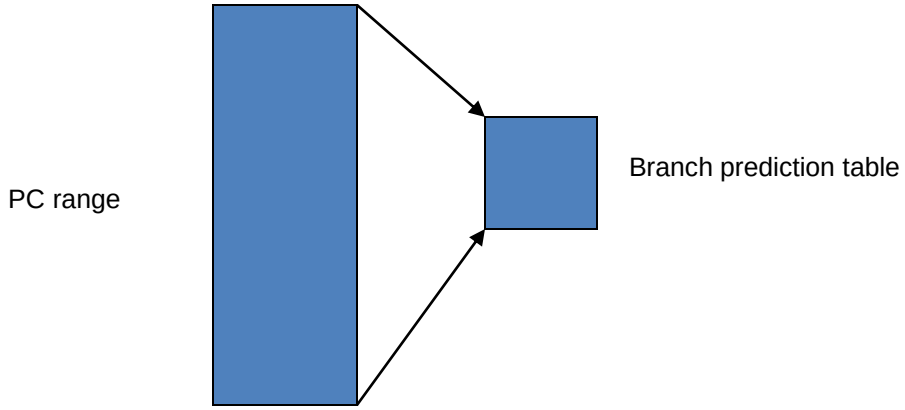
1-bit branch predictor



2-bit branch predictor

- How does hardware "learn" branch behavior?
- Store each branch instruction's history ***
  - If a branch was taken "recently", predict taken
    - One bit saturating counter
    - Two bit counters

Branch Prediction



- Store each branch's history ***
  - Not really
- Keep a small table indexed by program counter
- PC is large (32 bit number)
- Mapping to number of table entries
  - E.g. 16-entry branch prediction table
  - Mapping: use last 4 bits of PC
- Problem: Multiple branches may map to same entry in table -- Aliasing

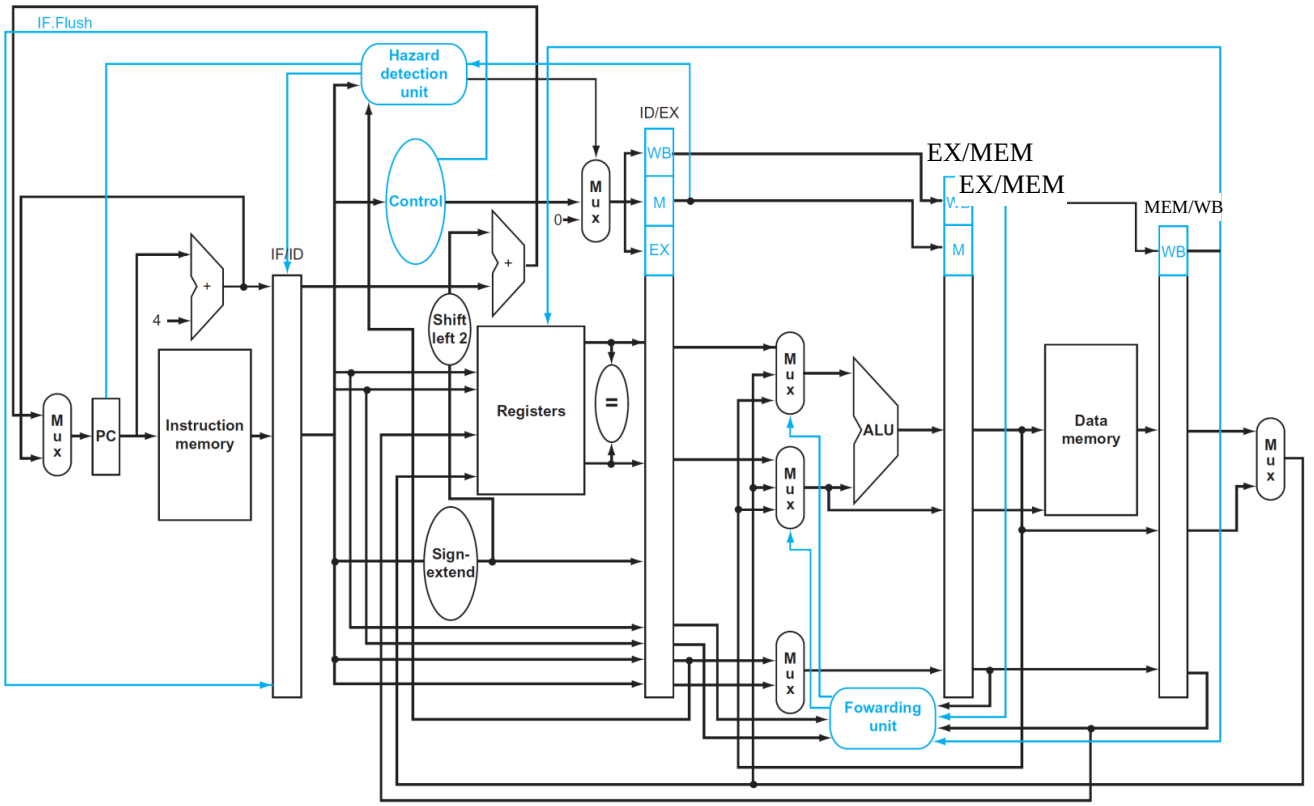
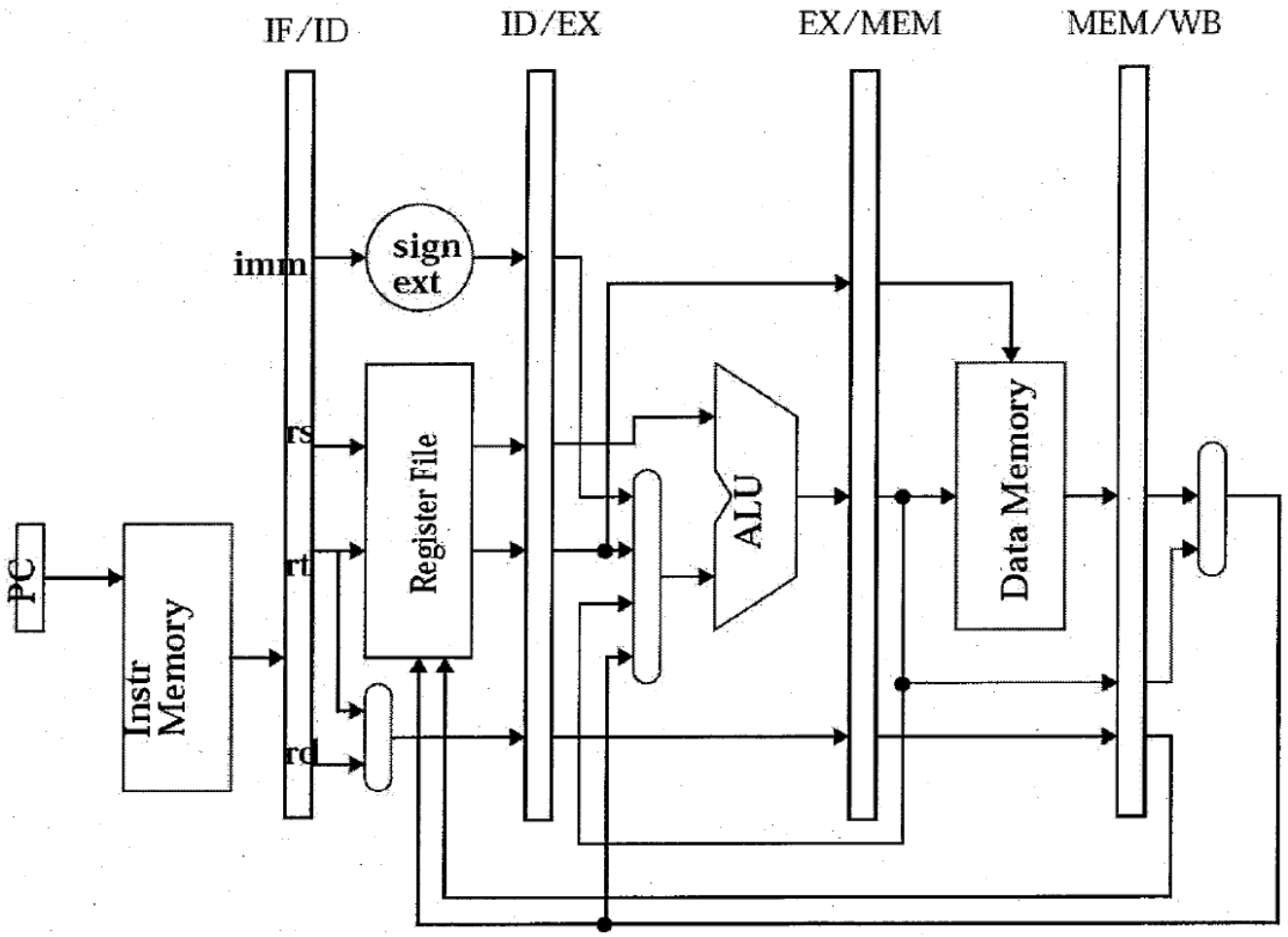


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
i0: add $1, $2, $3	F	D	X	M	W								
i1: sub $4, $1, $5		F	D	X x-x fwd	M	W							
i2: or $6, $1, $4			F	D	X x-x fwd($4) m-x fwd($1)	M	W						
i3: and $7, $4, $8				F	D	X w-x fwd	M	W					
i4: lw $9, 4($7)					F	D	X x-x fwd	M	W				
i5: add $1, $9, $2						F	D	D	X m-x fwd	M	W		
i6: sw $1, 4($7)							F	F	D	X	M	W	



Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
i0: add $1, $2, $3	F	D	X	M	W								
i1: sub $4, $1, $5		F	D	D	D	X	M	W					
i2: or $6, $1, $4			F	F	F	D	X	M	W				
i3: and $7, $4, $8					F	D	D	X	M	W			
i4: lw $9, 4($7)						F	F	D	D	D	X	M	W
i5: add $1, $9, $2								F	F	F	D	D	D
i6: sw $1, 4($7)													

Exceptions and Pipelining

- add $1, $2, $3 overflows
- A surprise branch
  - Earlier instructions flow to completion
  - Kill later instructions
  - Save PC in EPC, set PC to EX handler, etc.
- Costs a lot of designer sanity
  - 554 teams that try this sometimes fail

Exceptions

- Even worse: in one cycle
  - I/O interrupt
  - User trap to OS (EX)
  - Illegal instruction (ID)
  - Arithmetic overflow
  - Hardware error
  - Etc.
- Interrupt priorities must be supported

Pipeline Hazards

THE UNIVERSITY of WISCONSIN MADISON

- Program Dependences
- Data Hazards
  - Stalls
  - Forwarding
- Control Hazards
- Exceptions