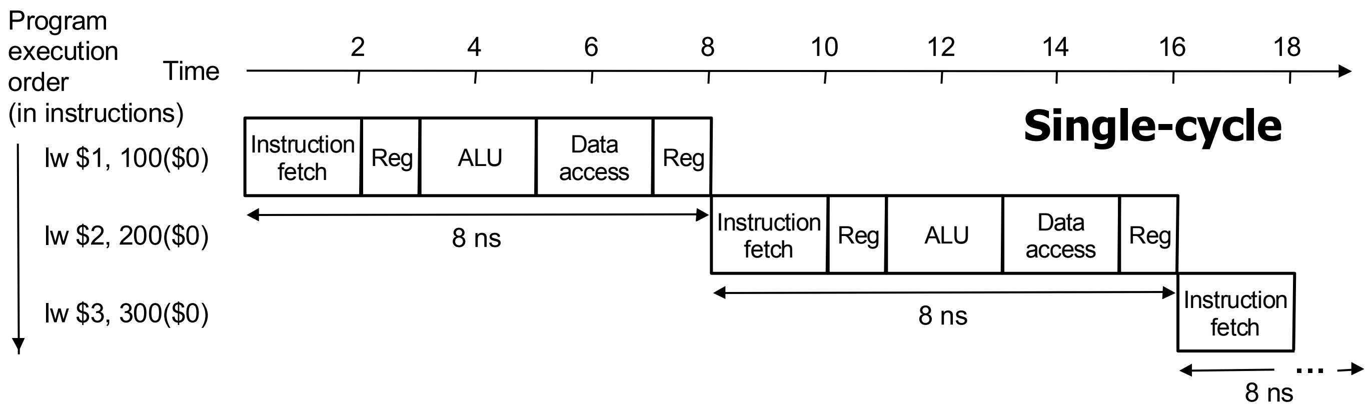
Enhancing Performance with PIPELINING

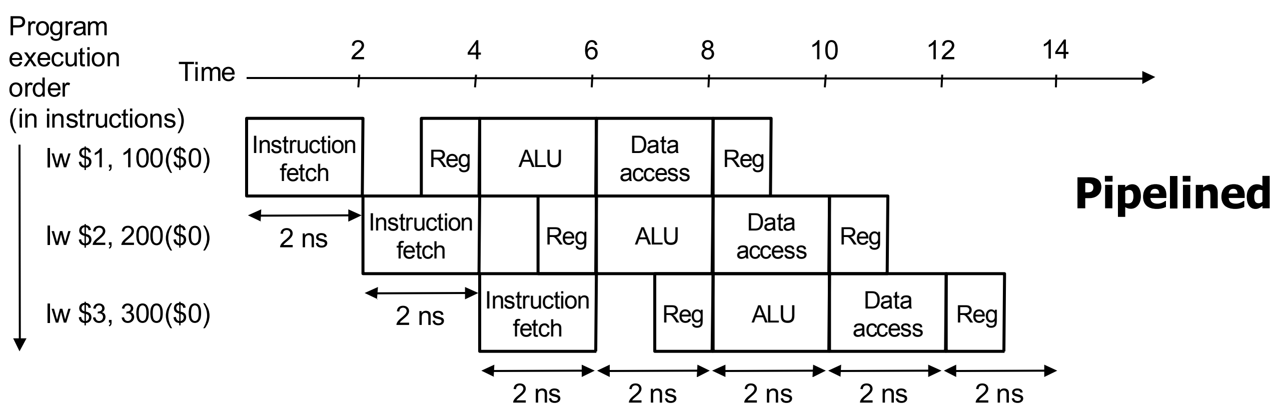
Pipelining

- Pipeline concepts
- Hazards
- Example

Pipelined vs. Single-Cycle Instruction Execution



Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Pipeline Implementation

- Idea:
  - Goal of MIPS: CPI <= 1
  - Some instructions take longer to execute than others
  - Don’t want cycle time to depend on slowest instruction
  - Want 100% hardware utilization
  - Split execution of each instruction into several, balanced “stages”
  - Each stage is a block of combinational logic
  - Latency of each stage fits within 1 clock cycle
  - Insert registers between each pipeline stage to hold intermediate results
  - **Execute each of these steps in parallel for a sequence of instructions**

- This is called pipelining

Pipelining MIPS

- MIPS characteristics make pipelining easy
  - All *instructions are approx. same length*
    - so fetch and decode stages are similar for all instructions
  - Just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *Memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *Operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

MIPS pipeline stages

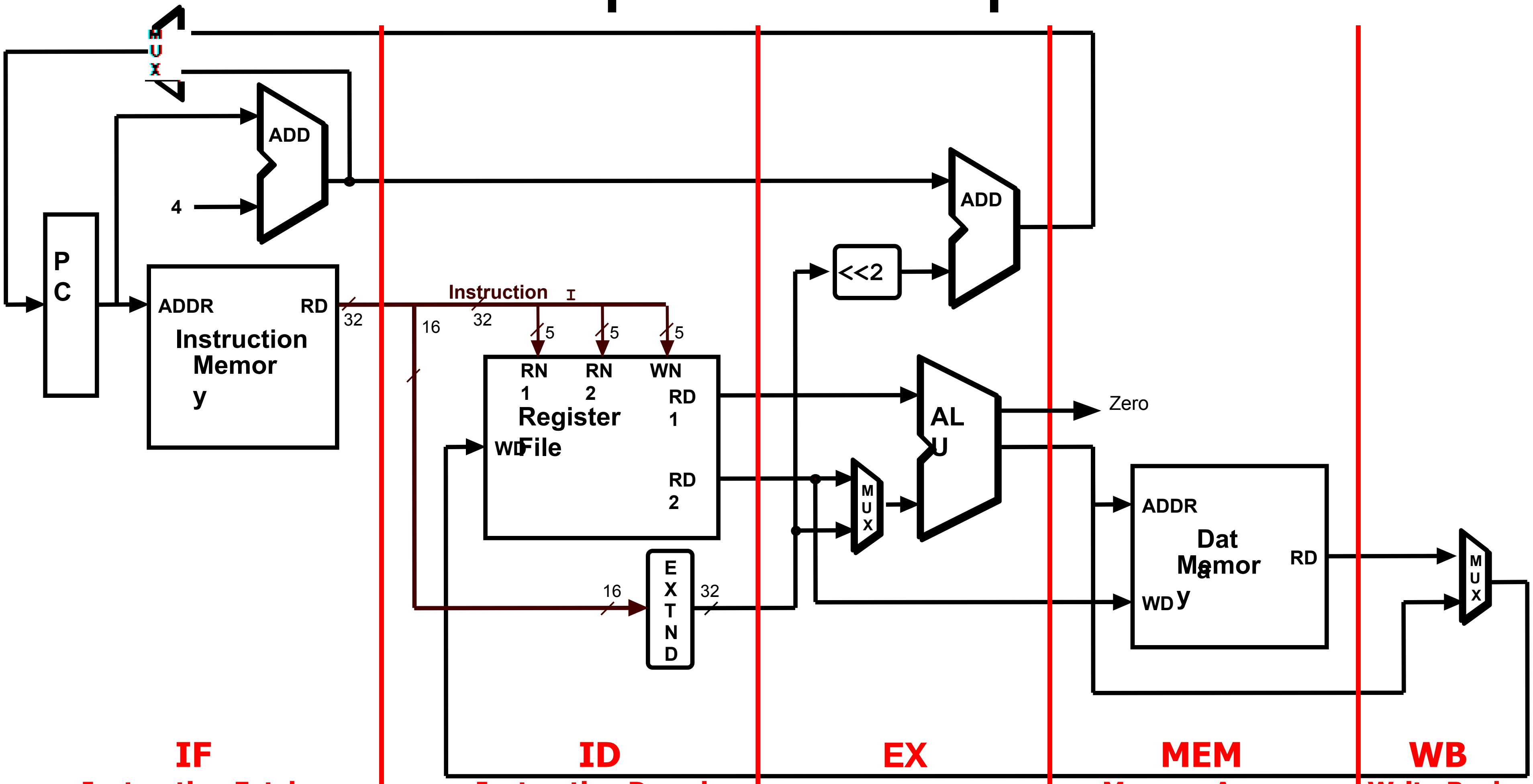
- **Fetch (IF)**
  - Read next instruction from memory
  - Increment address counter
- **Decode (ID)**
  - Read register operands,
  - Resolve instruction in control signals
  - Compute branch target
- **Execute (EX)**
  - Execute arithmetic/resolve branches
- **Memory (MEM)**
  - Perform load/store accesses to memory
  - Take branches
- **Write back (WB)**
  - Write arithmetic results to register file

Pipelined Datapath

Recall the 5 steps in instruction execution

1. Instruction Fetch & PC Increment (IF)
2. Instruction Decode and Register Read (ID)
3. Execution or calculate address (EX)
4. Memory access (MEM)
5. Write result into register (WB)

Review - Single-Cycle Datapath "Steps"



IF
Instruction Fetch

ID
Instruction Decode

EX
Execute/ Address Calc

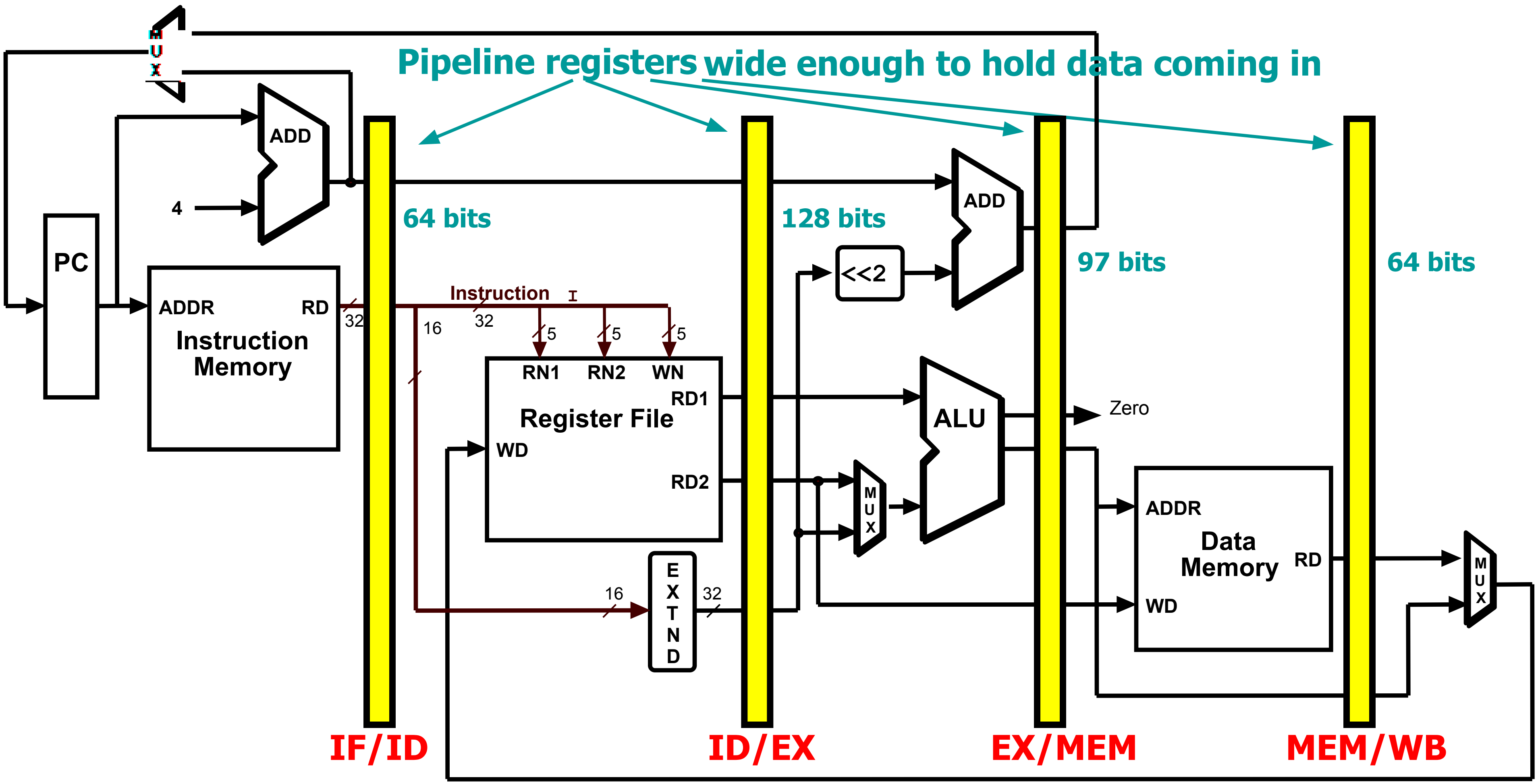
MEM
Memory Access

WB
Write Back

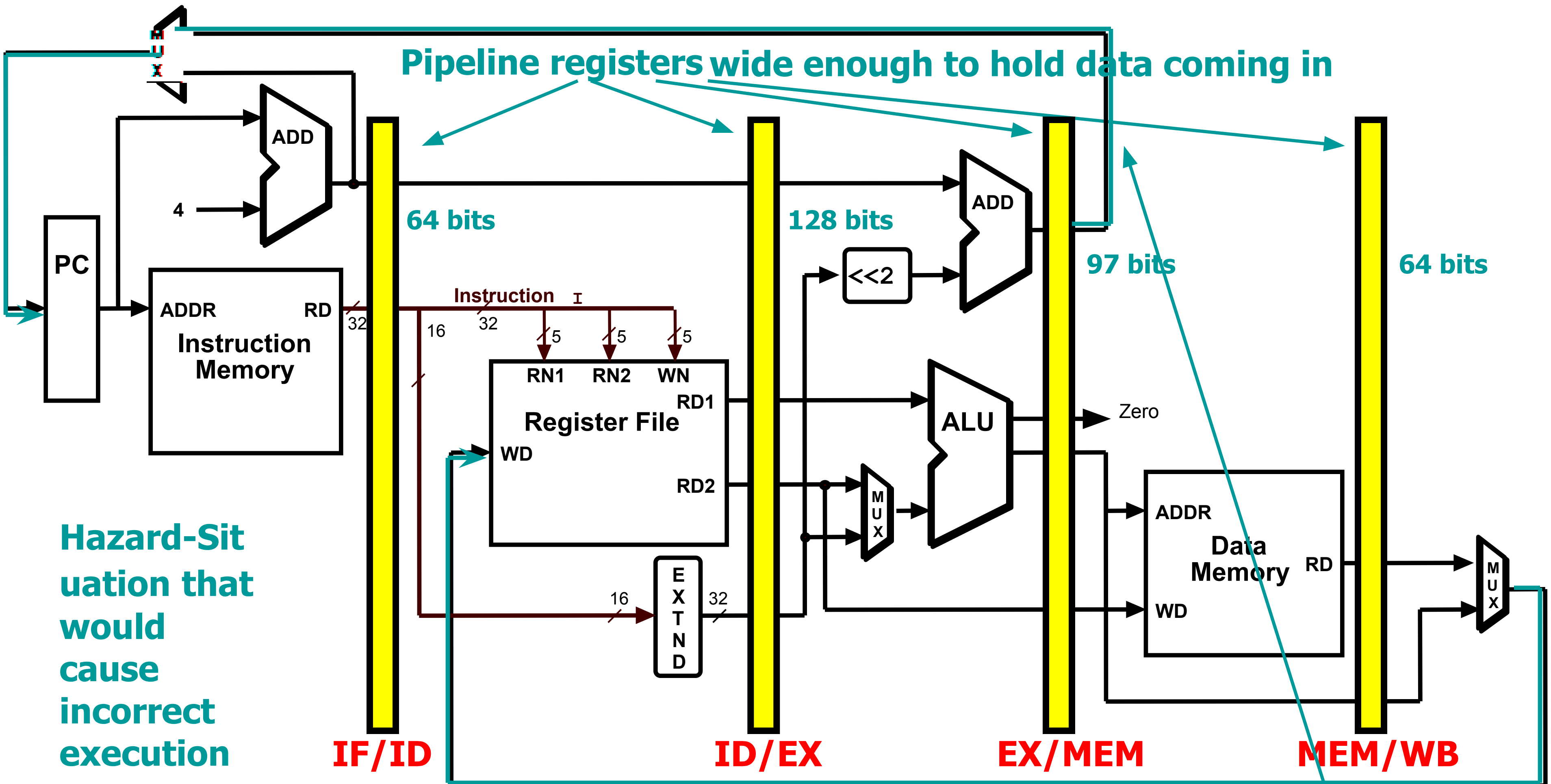
Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - Answer: *We may be able to start executing a new instruction at each clock cycle -* pipelining
- …but we shall need *extra* registers to hold data between cycles – *pipeline registers*

Pipelined Datapath



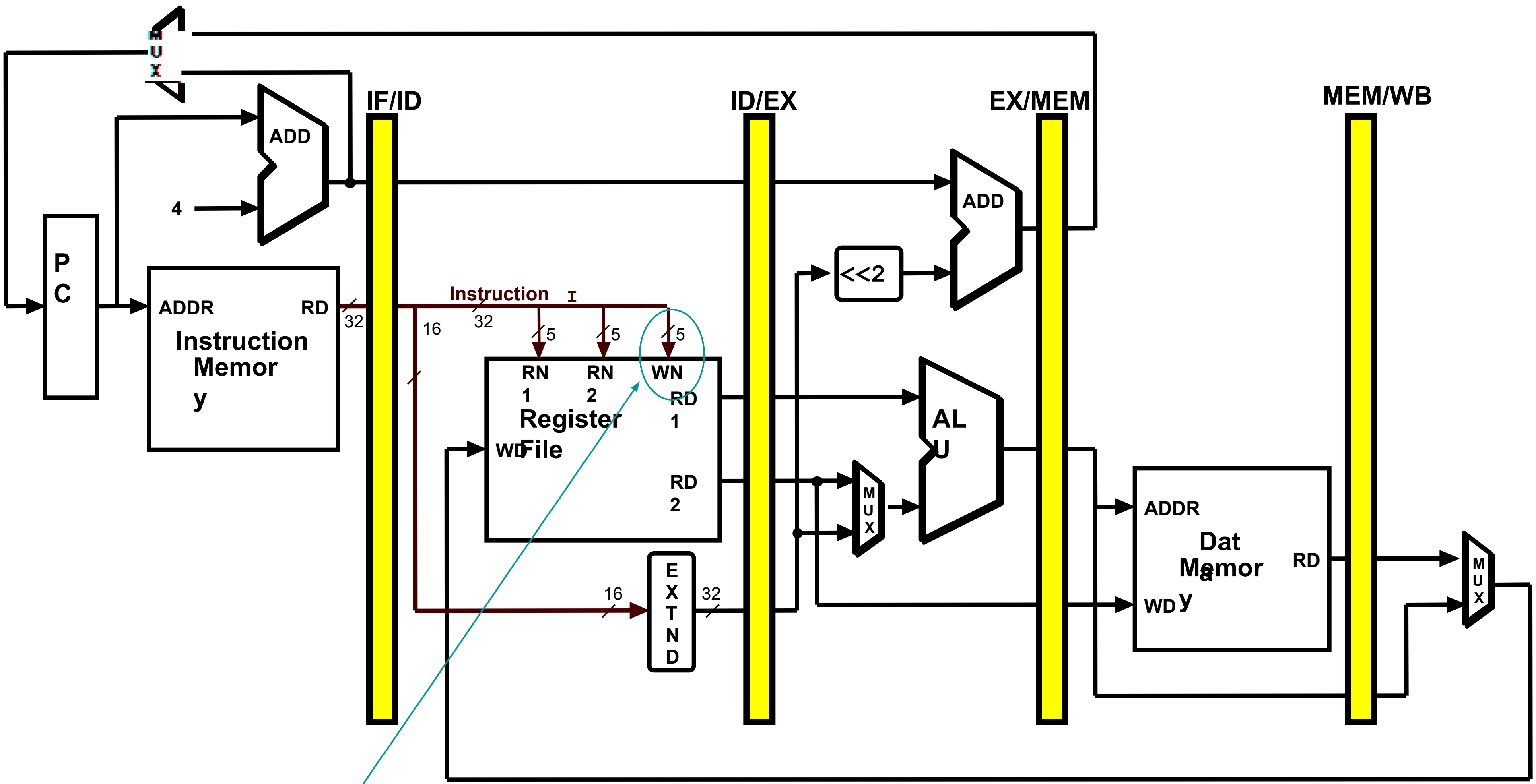
Pipelined Datapath



Hazard-Situation that would cause incorrect execution

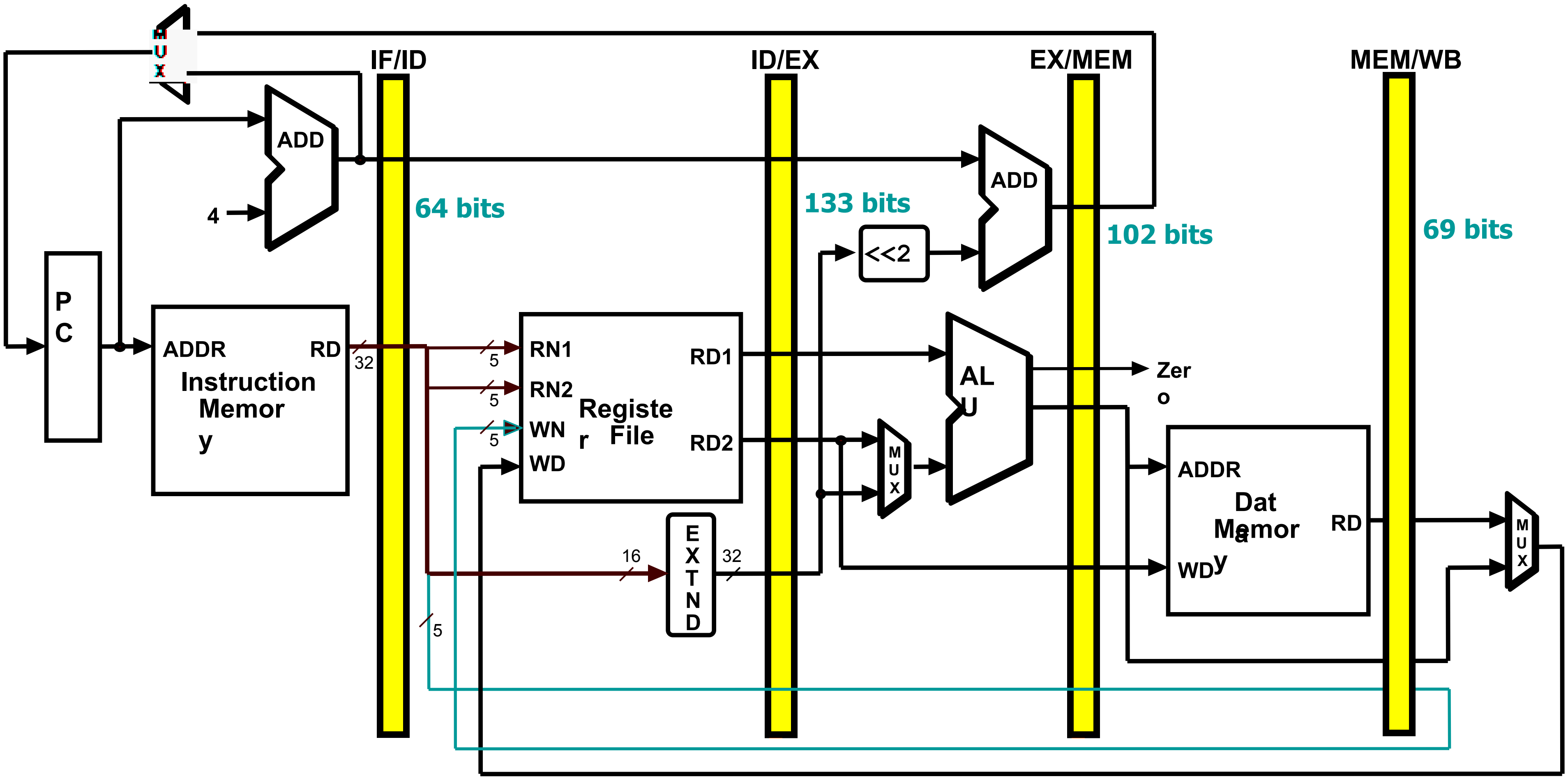
Only data flowing right to left may cause hazard…, why?

Bug in the Datapath



Write register number comes from another *later* instruction!

Corrected Datapath



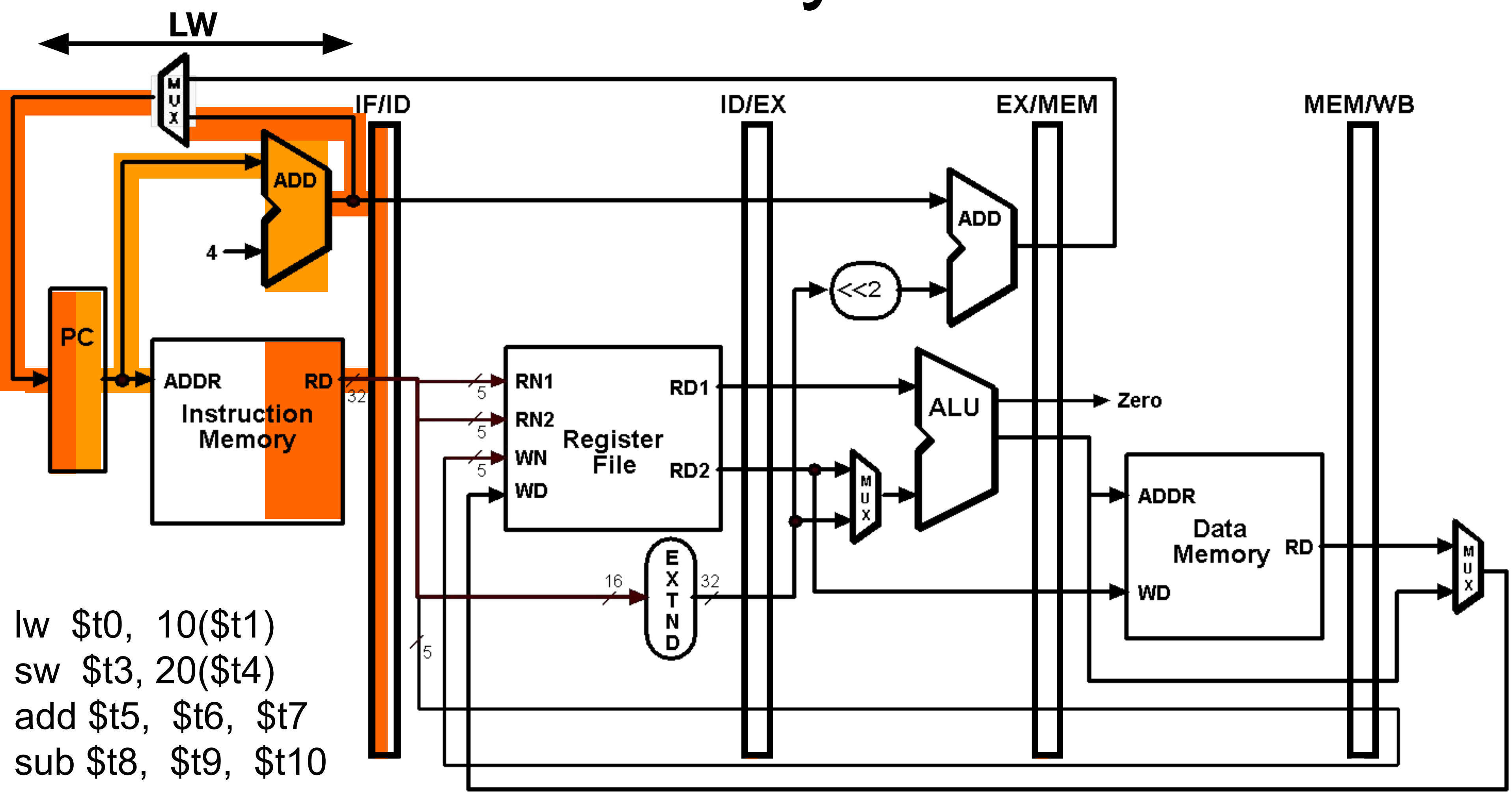
Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

- Consider the following instruction sequence:

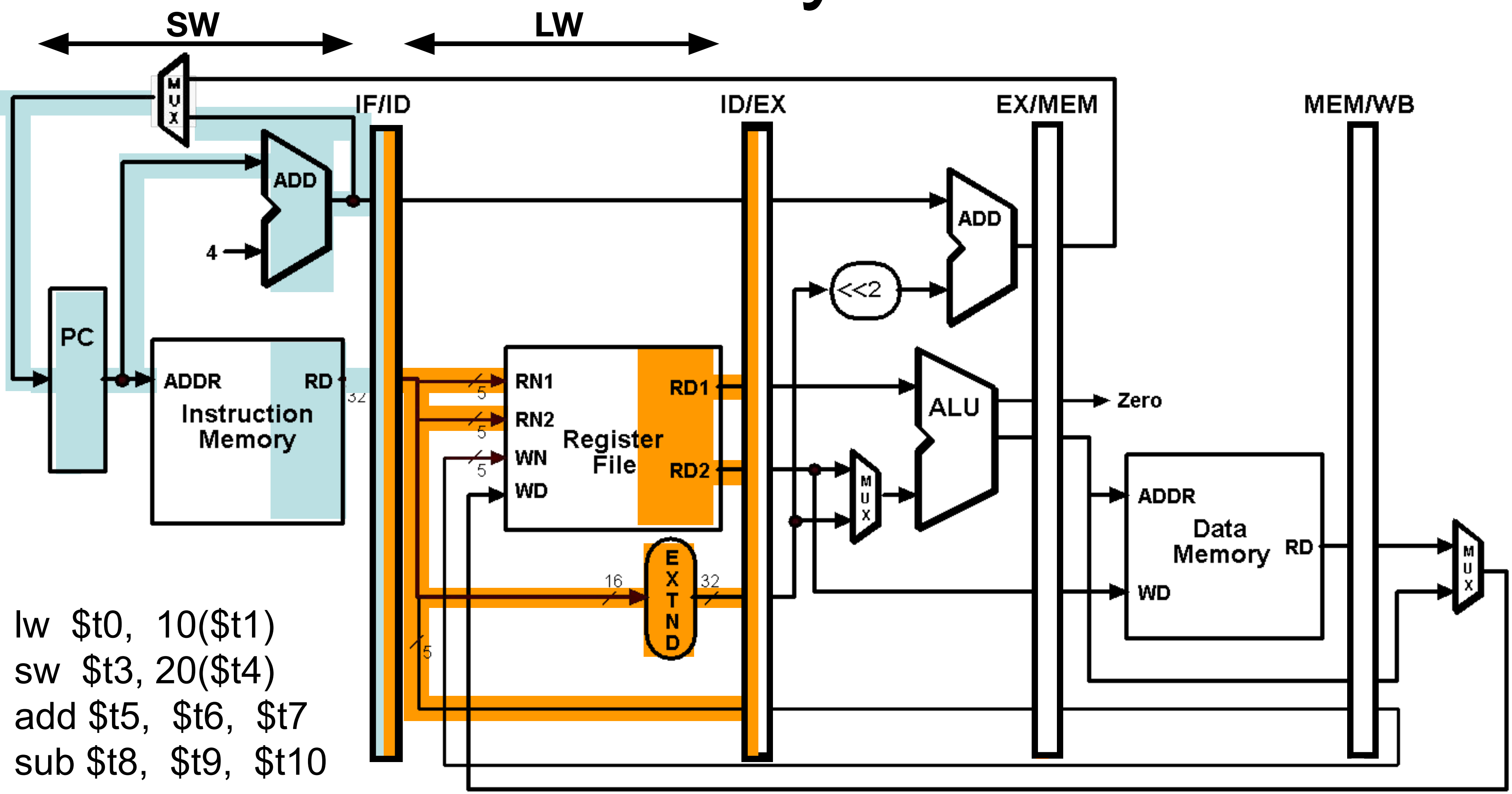
```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

Single-Clock-Cycle Diagram: Clock Cycle 1



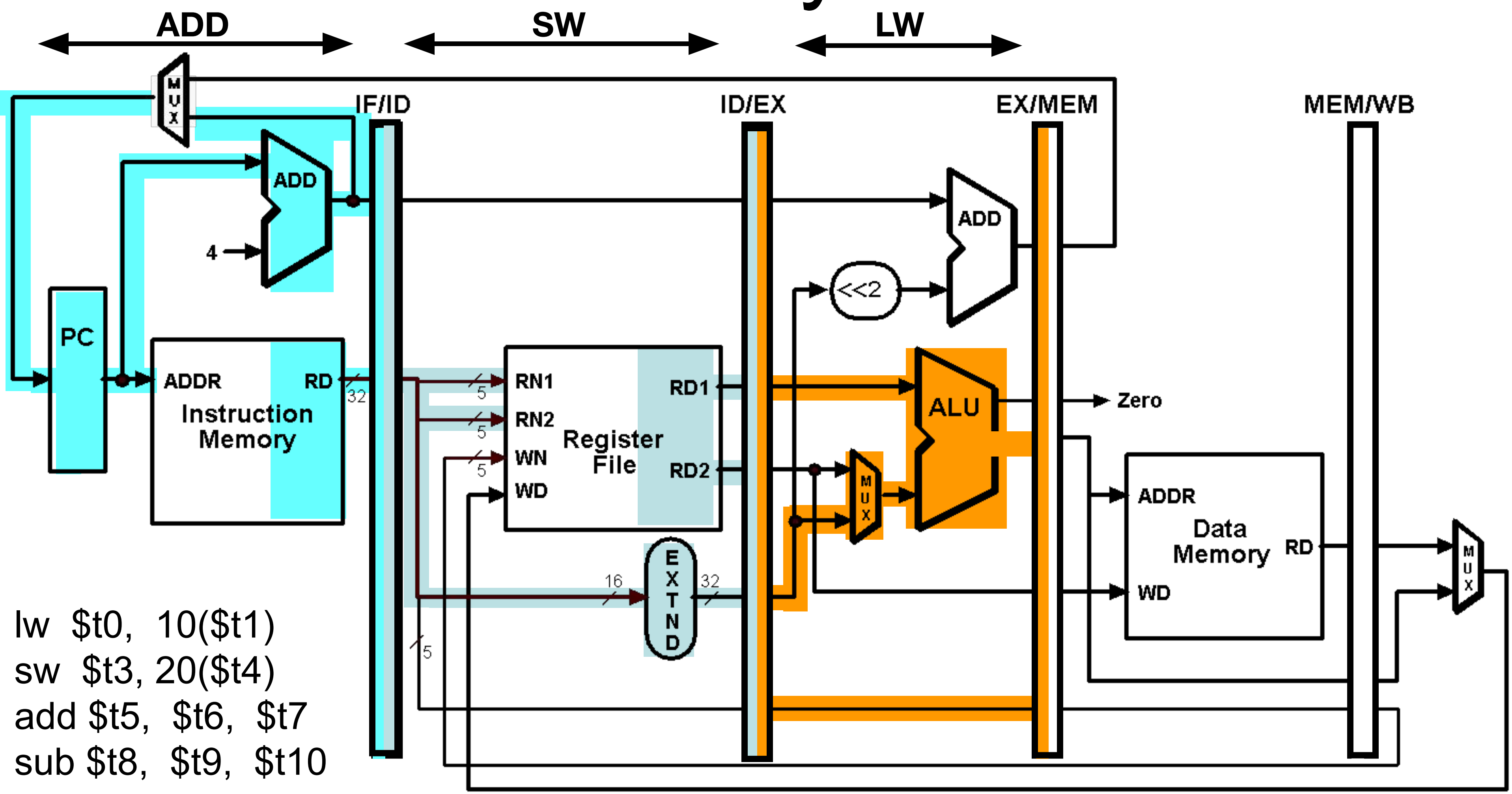
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 2



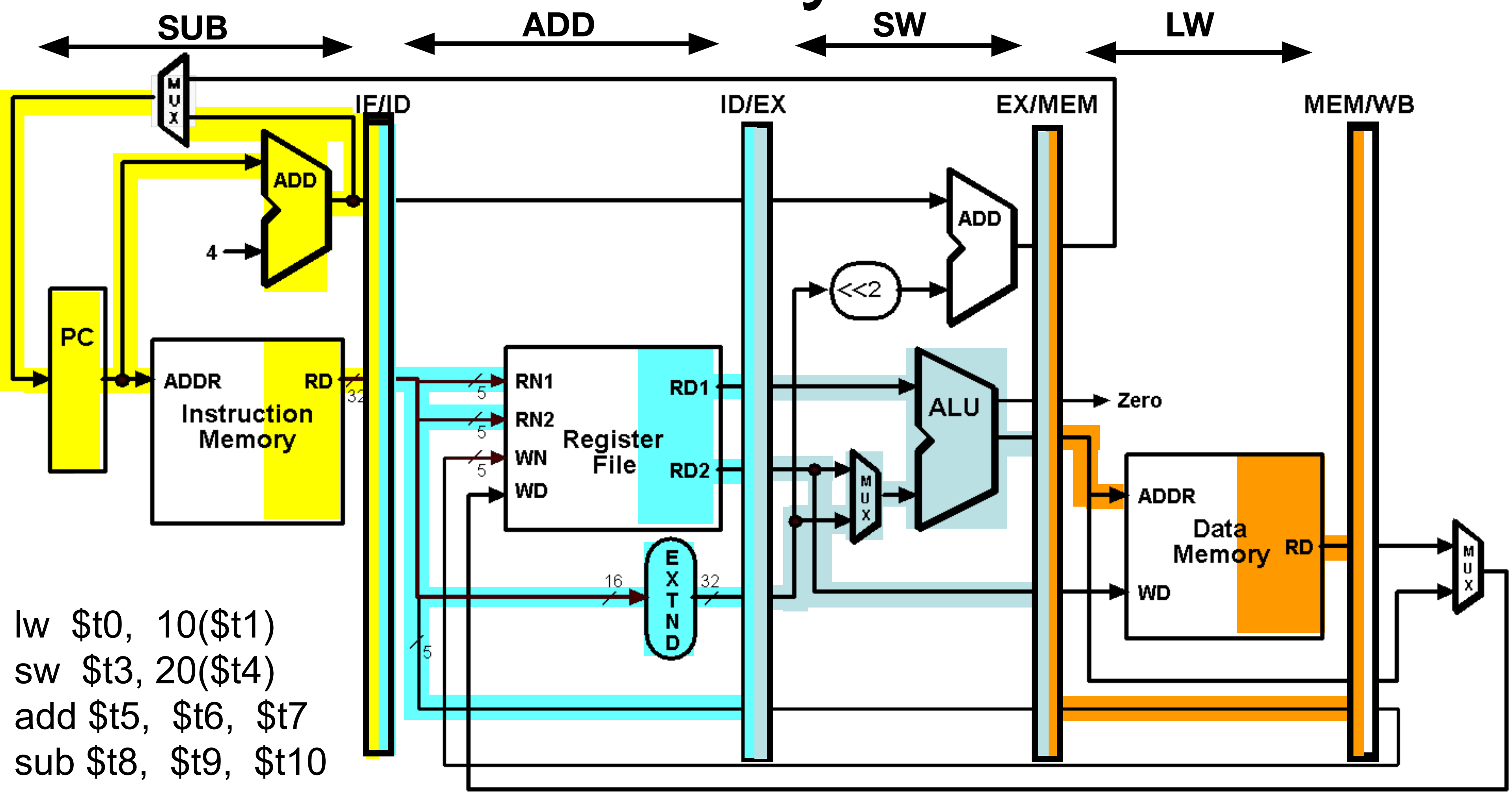
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 3



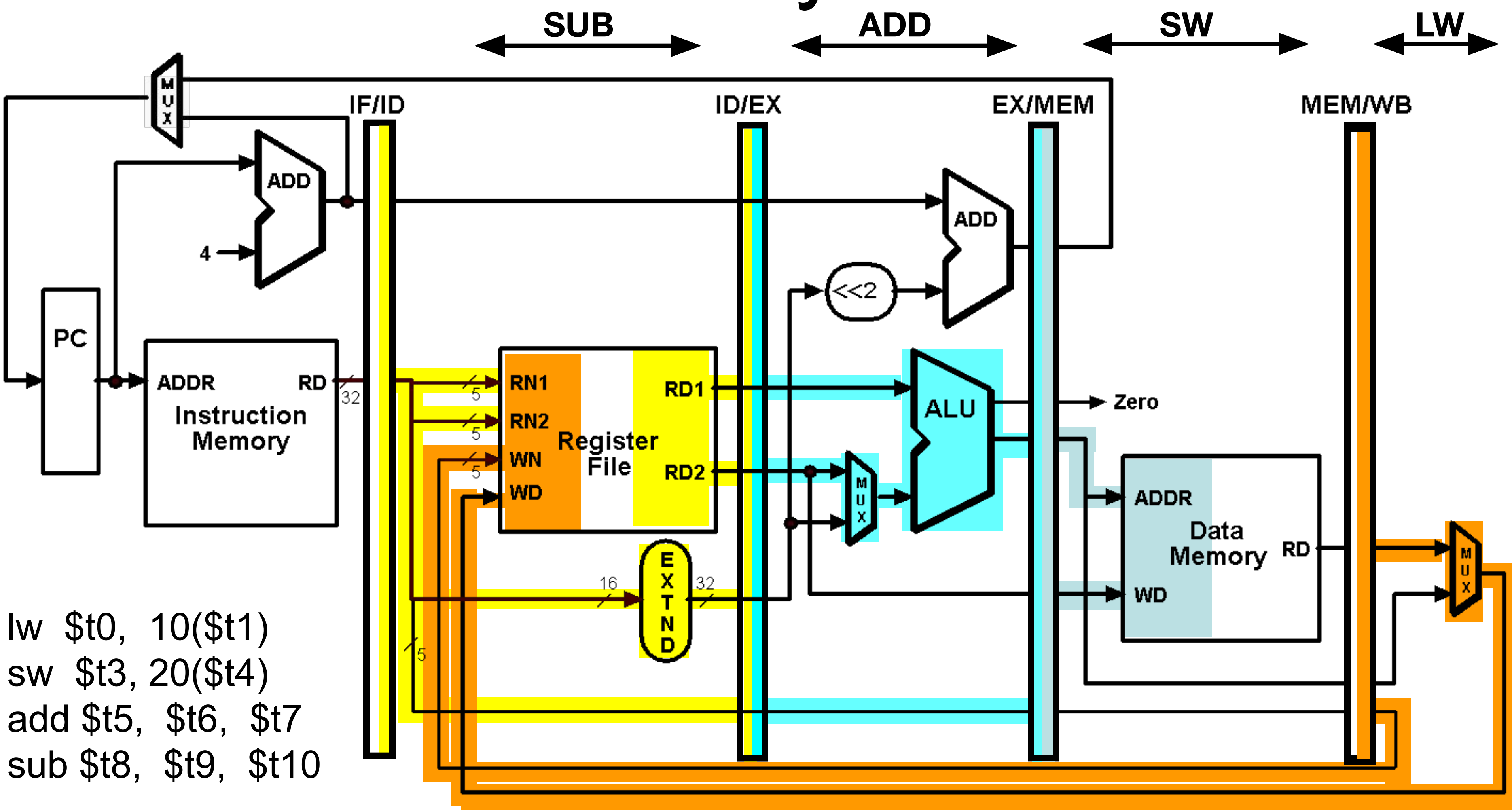
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 4



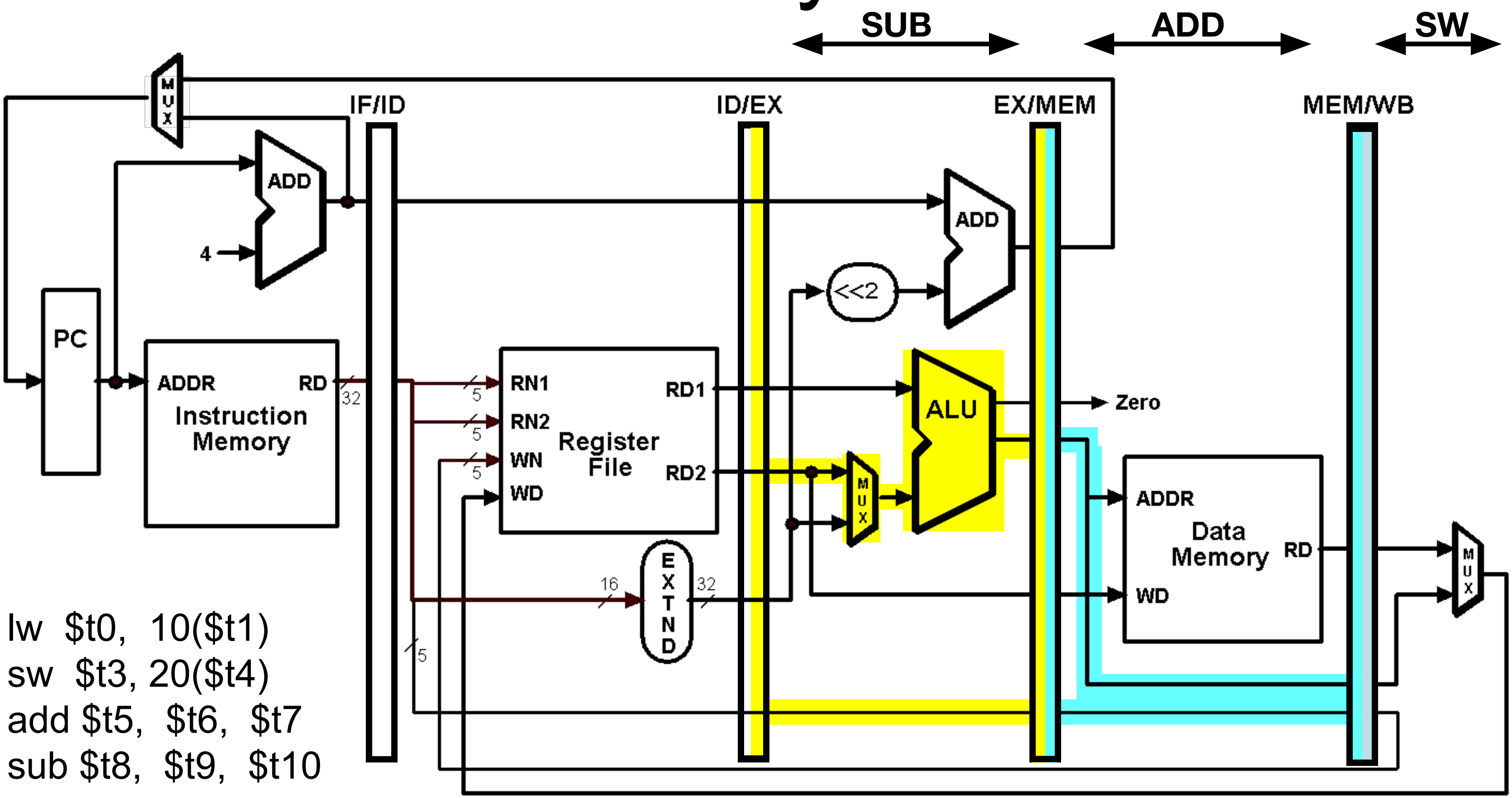
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 5



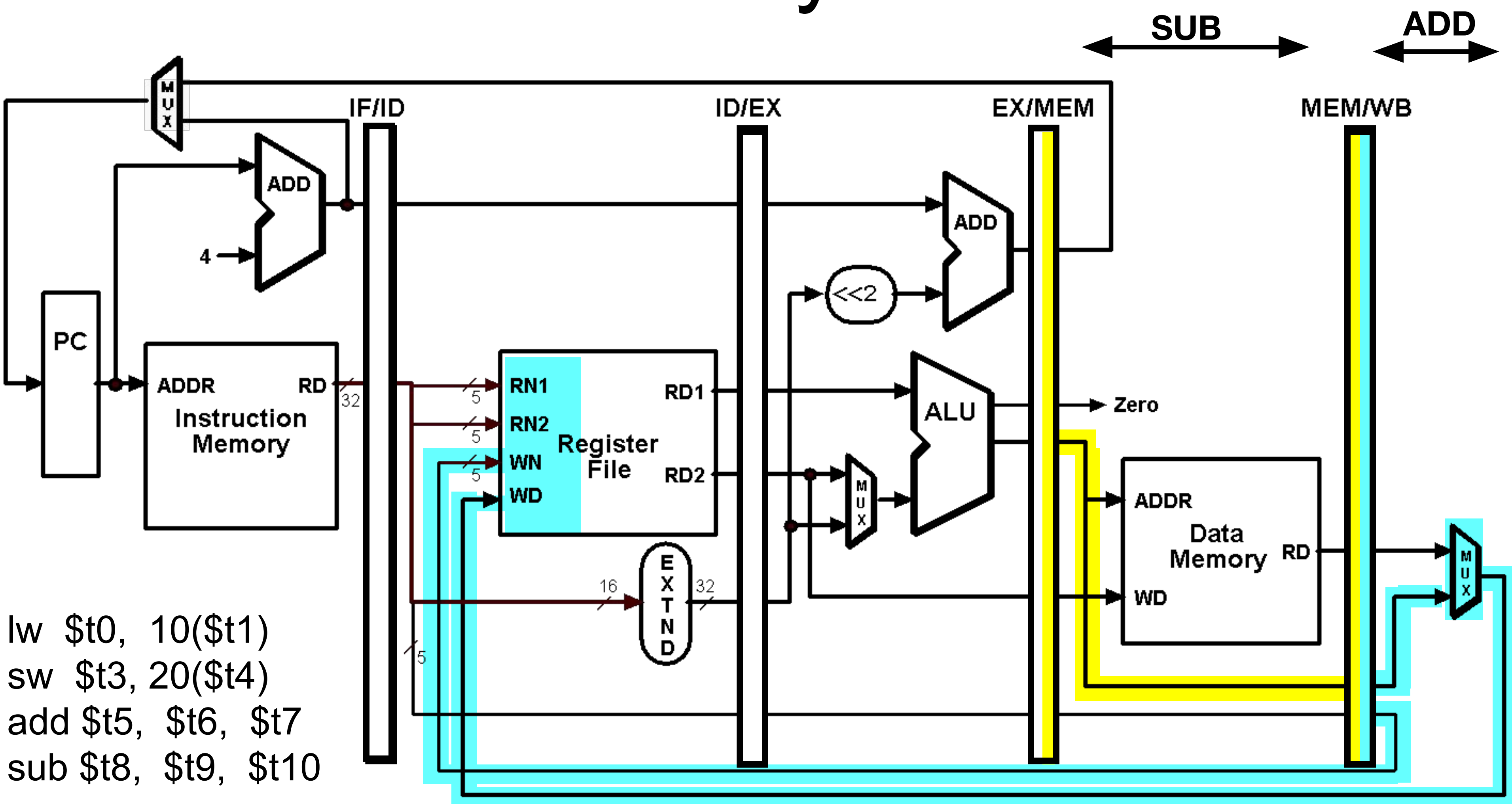
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 6



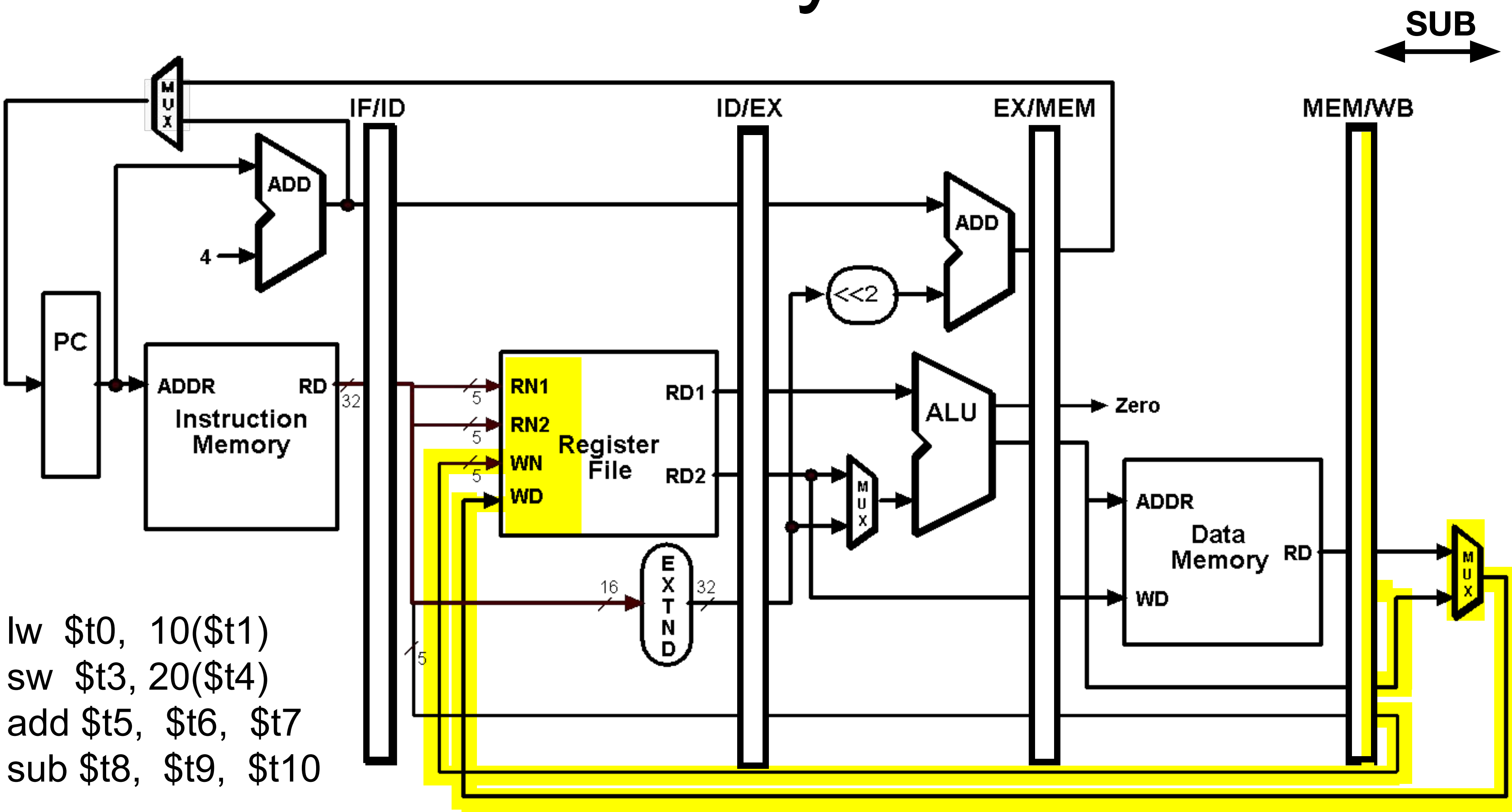
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 7



lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

Single-Clock-Cycle Diagram: Clock Cycle 8



lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

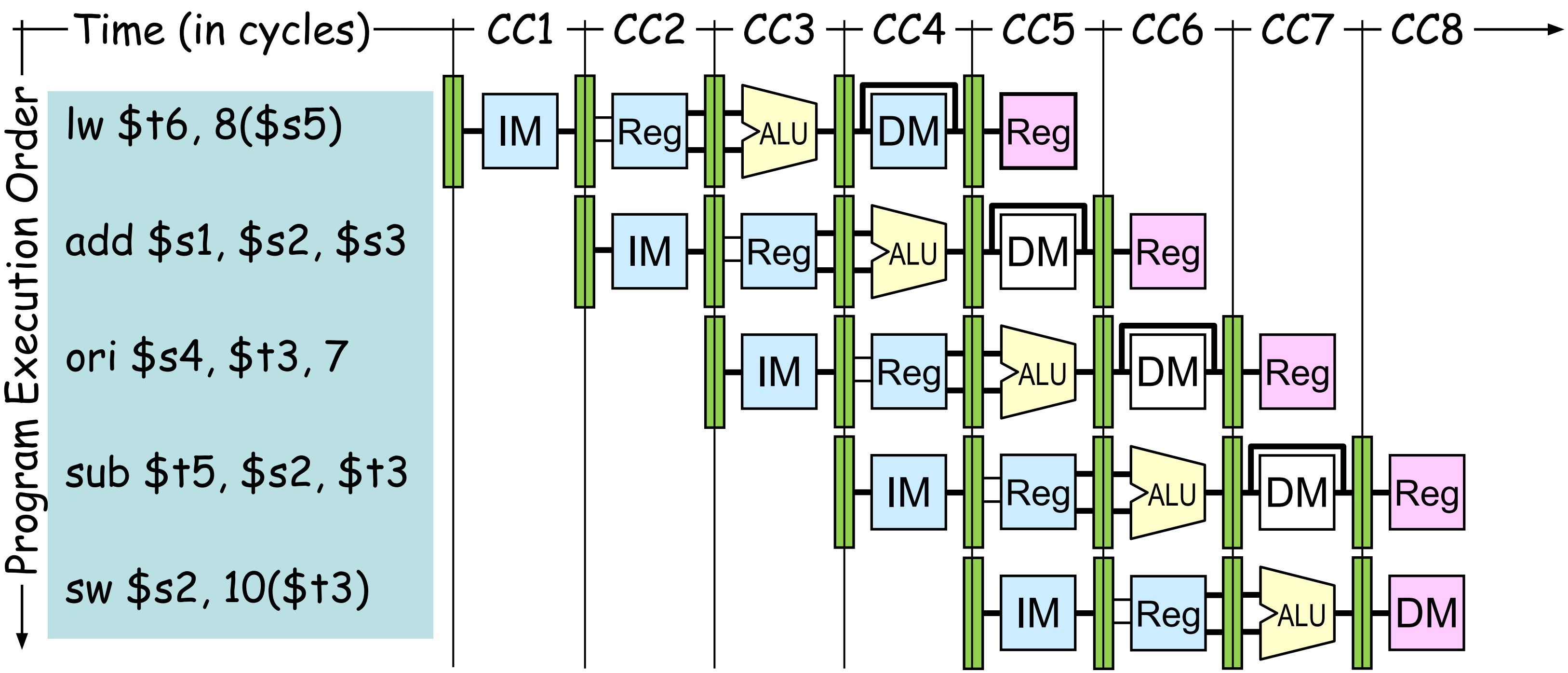
Represent Pipelines Graphically

- Multiple instruction execution over multiple clock cycles
  - Instructions are listed in execution order from top to bottom
  - Clock cycles move from left to right
  - Show the use of resources at each stage and each cycle

Represent Pipelines Graphically

1. Lw $t6, 8($s5)
2. Add $s1, $s2, $s3
3. Ori $s4, $t3, 7
4. Sub $t5, $s2, $t3
5. Sw $s2, 10($t3)

Graphically Representing Pipelines



Instruction-Time Diagram

- Instruction-Time Diagram shows:
  - Which instruction occupying what stage at each clock cycle
- Instruction flow is pipelined over the 5 stages

1. Lw $t7, 8($s3)
2. Lw $t6, 8($st)
3. Ori $t4, $s3, 7
4. Sub $s5, $s2, $t3
5. Sw $s2, 10($s3)

Instruction-Time Diagram

Up to five instructions can be in the pipeline during the same cycle
Instruction Level Parallelism (ILP)

ALU instructions skip the MEM stage.
Store instructions skip the WB stage

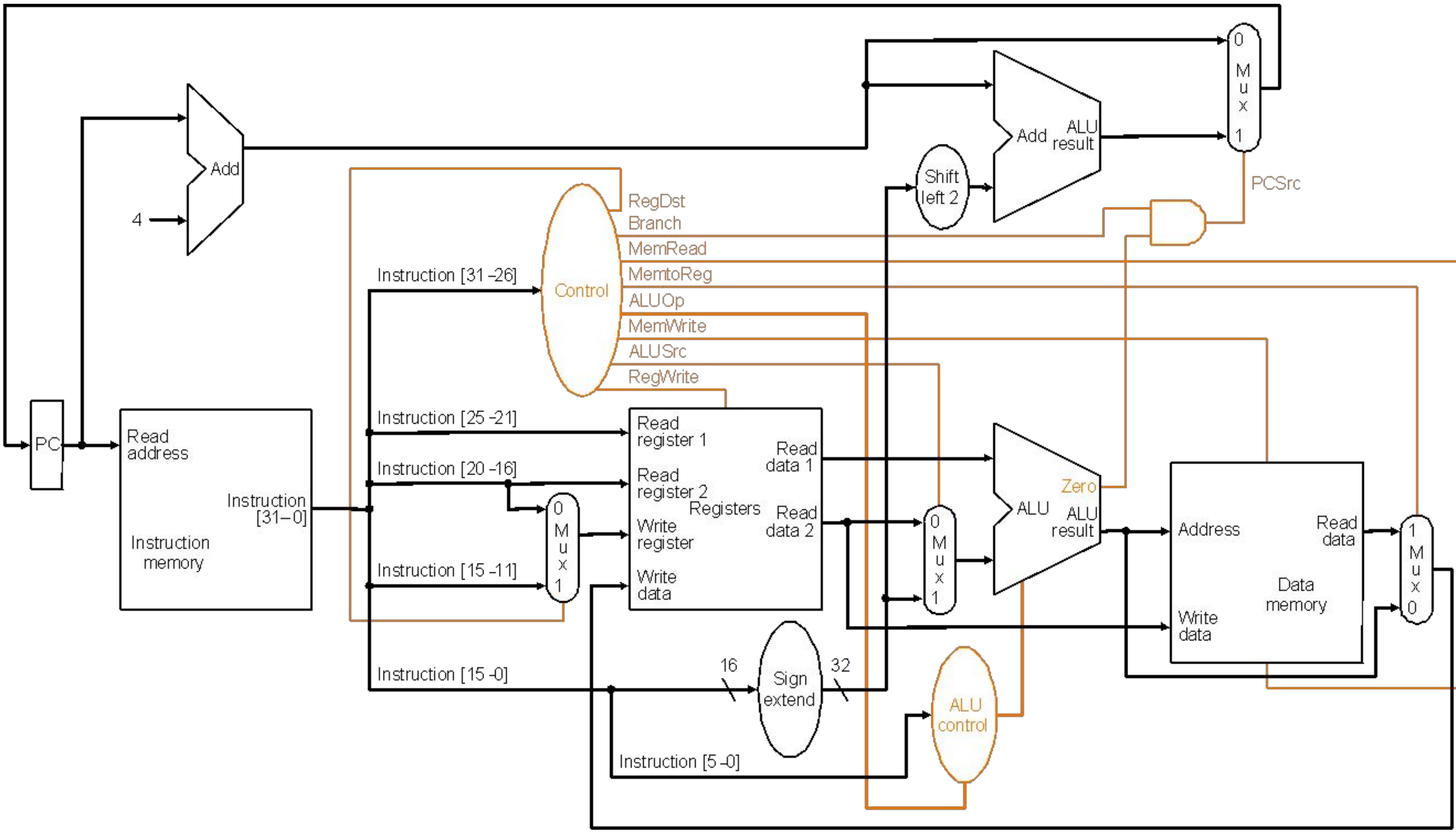
Instruction Order	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
lw $t7, 8($s3)	IF	ID	EX	MEM	WB				
lw $t6, 8($s5)		IF	ID	EX	MEM	WB			
ori $t4, $s3, 7			IF	ID	EX	–	WB		
sub $s5, $s2, $t3				IF	ID	EX	–	WB	
sw $s2, 10($s3)					IF	ID	EX	MEM	–

Time

How many Clock Cycles?

5 Instructions + (5 step pipelining - 1) = 9 Clock cycles

Recall Single-Cycle Control – the Datapath



Recall Single-Cycle – ALU Control

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

Recall Single-Cycle – Control Signals

Effect of control bits

Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
AlLUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from ALU	The value fed to the register Write data input comes from the data memory

Deter-mining control bits

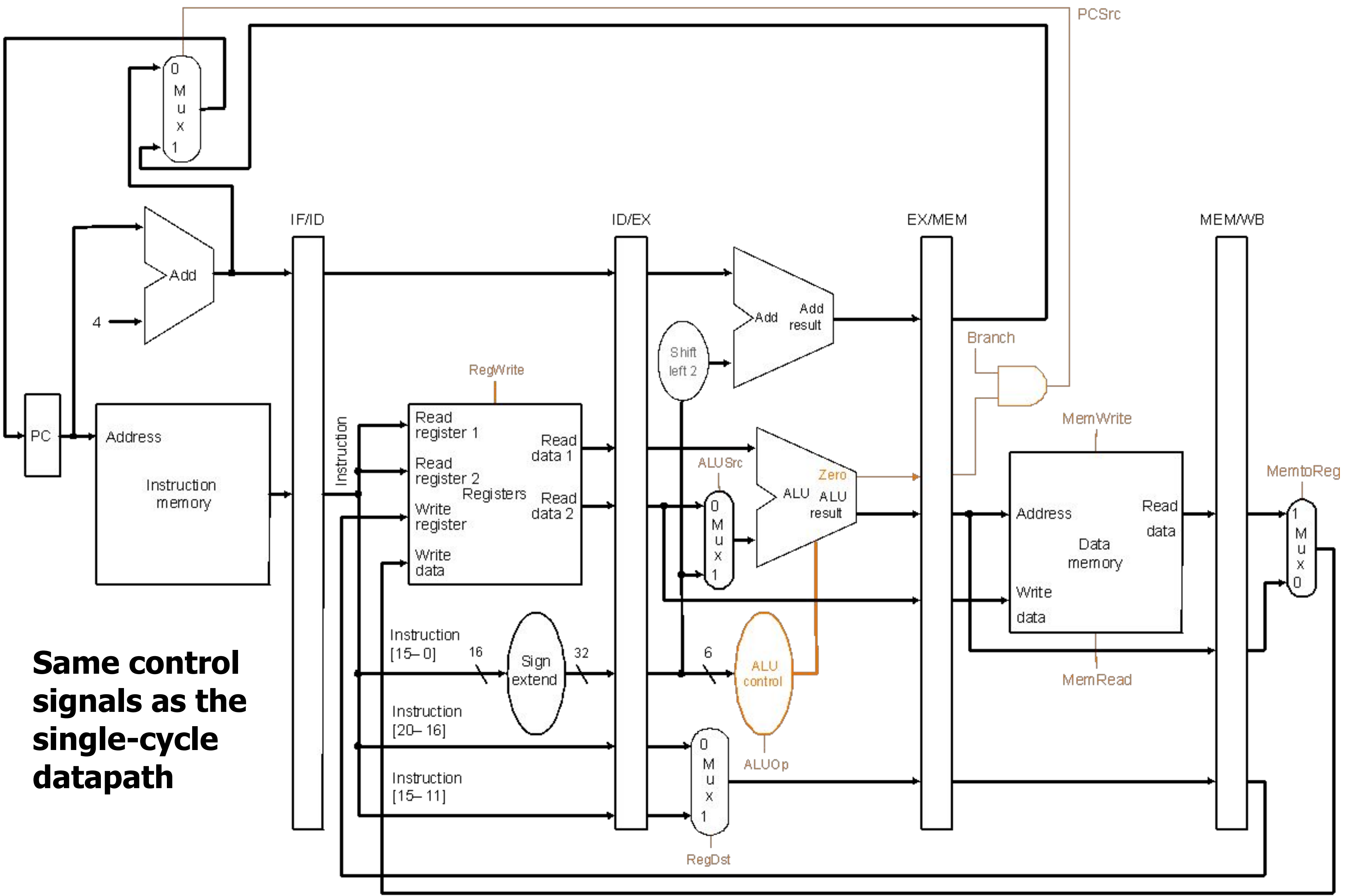
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Pipeline Control

- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
- Modified Signals:
  - No separate write signal for the PC as it is written every cycle
  - No separate write signals for the pipeline registers as they are written every cycle
  - *No separate read signal for instruction memory* as it is read every clock cycle
  - *No separate read signal for register file* as it is read every clock cycle
- Need to *set control signals during each pipeline stage*
- Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*

Will be modified by hazard detection unit!!

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

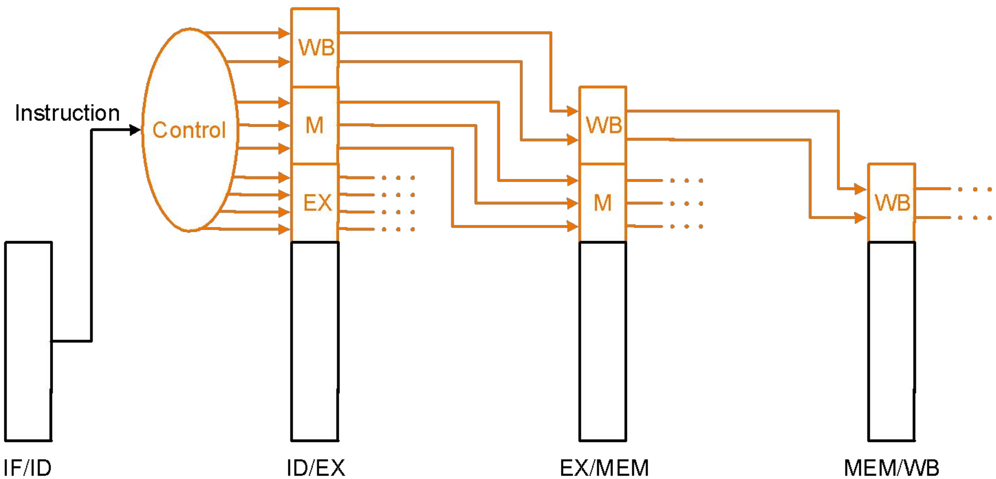
- There are five stages in the pipeline
  - *instruction fetch / PC increment*
  - *instruction decode / register fetch*
  - *execution / address calculation*
  - *memory access*
  - *write back*

Nothing to control as instruction memory read and PC write are always enabled

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	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

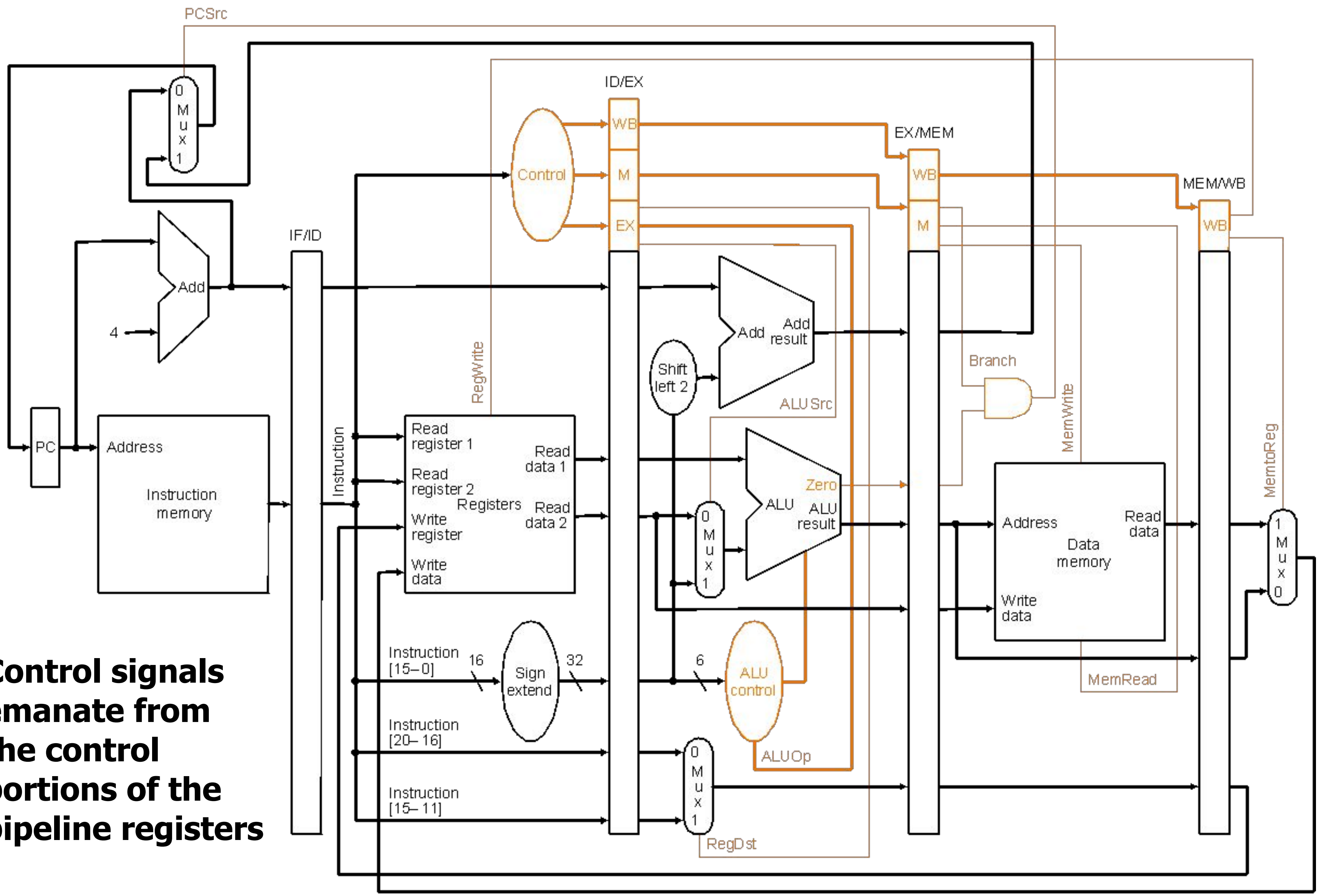
Pipeline Control Implementation

- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages



- *Note*: The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

Pipelined Datapath with Control II

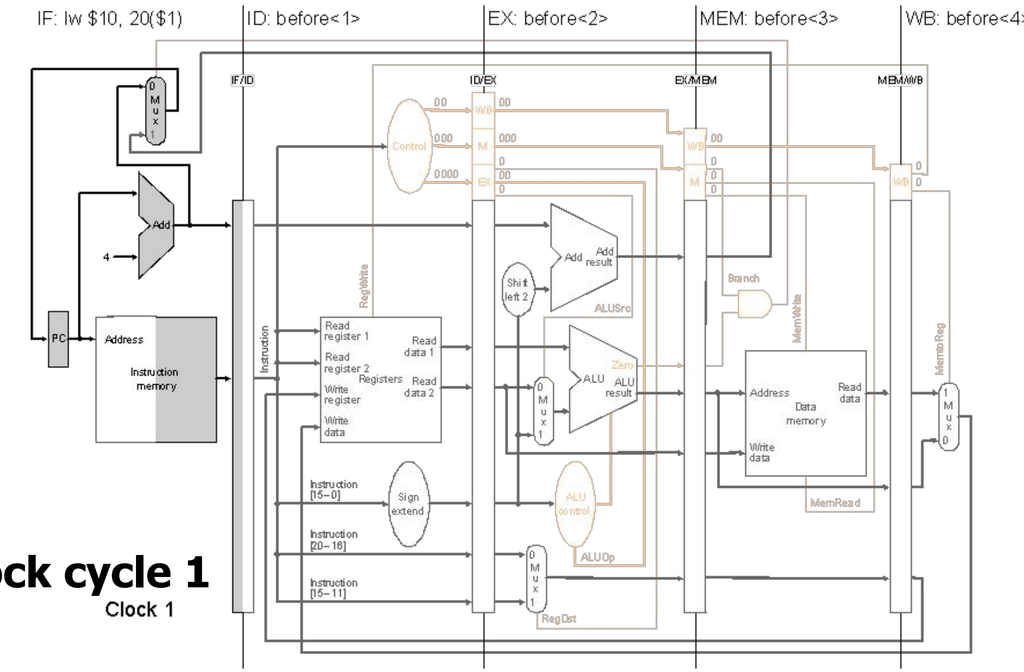


Control signals emanate from the control portions of the pipeline registers

Pipelined Execution and Control

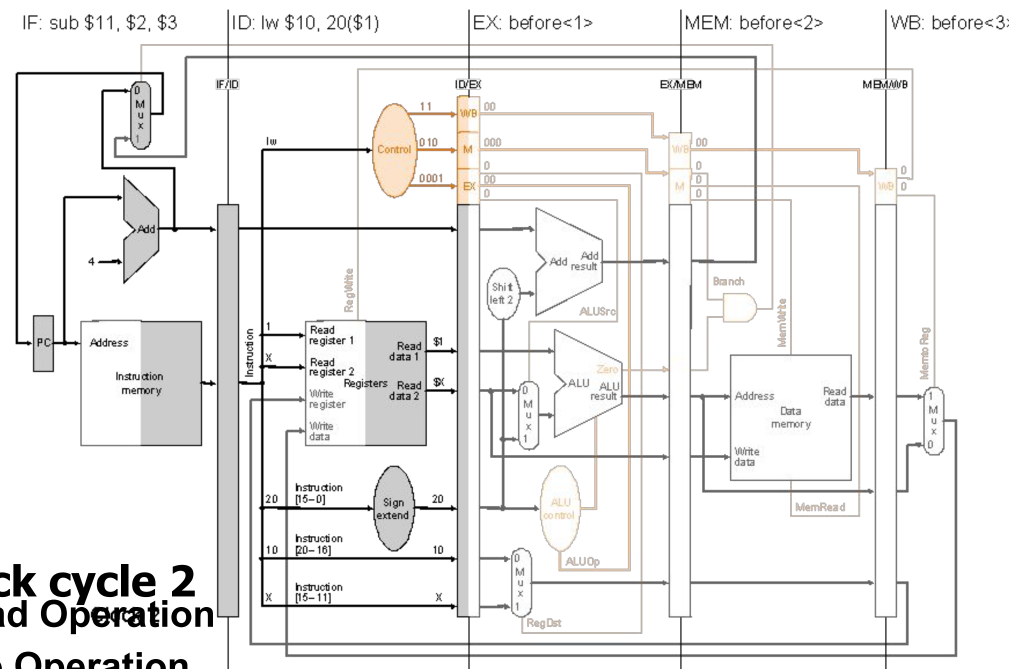
Instruction sequence:

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



Clock cycle 1

Clock 1



Clock cycle 2

Clock 2

Label "before<i>" means i th instruction before `lw`

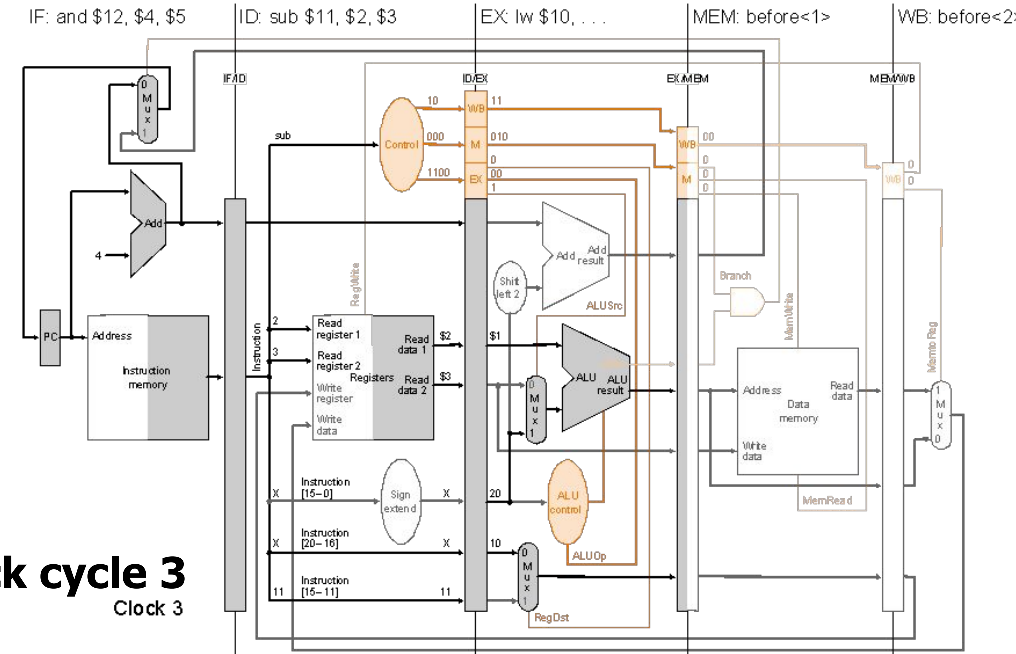
Dark Right Area indicates the Read Operation

Dark Left Area indicates the Write Operation

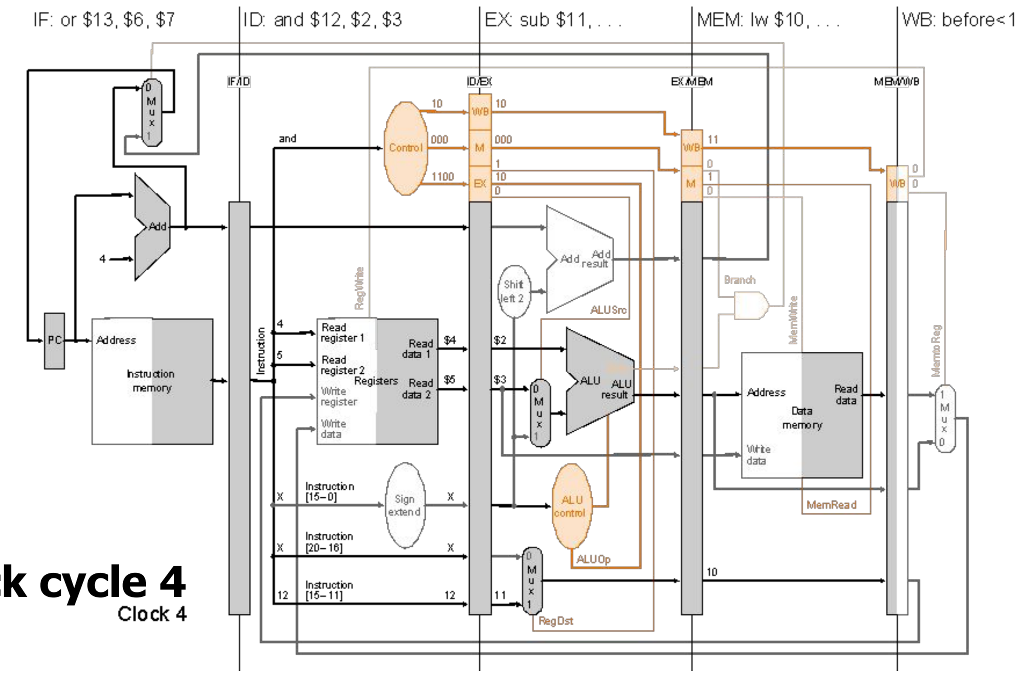
Pipelined Execution and Control

- Instruction sequence:

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



Clock cycle 3
Clock 3



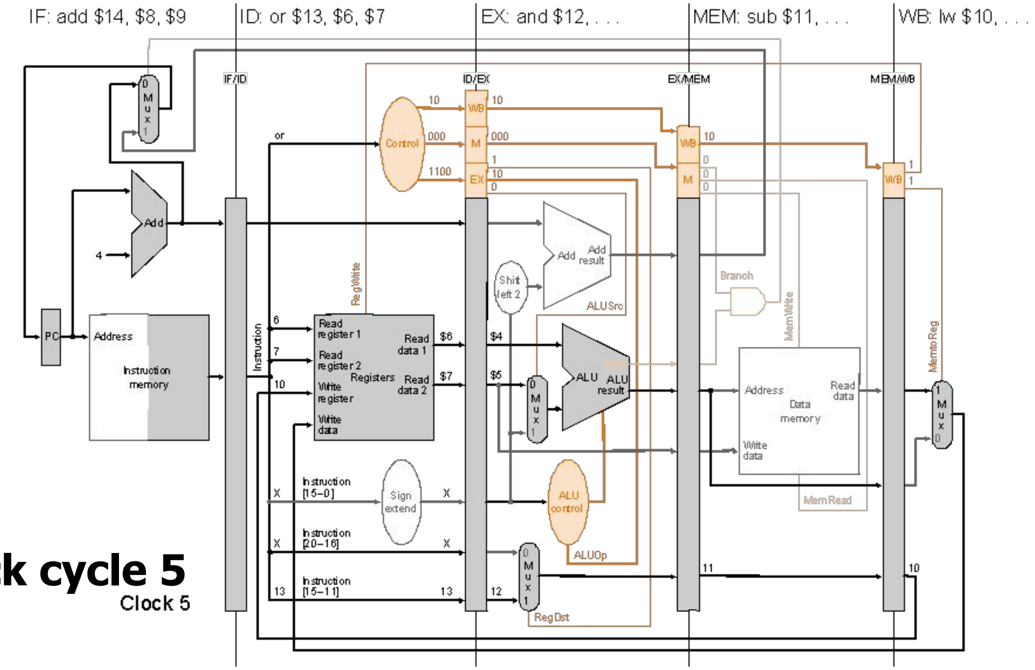
Clock cycle 4
Clock 4

Pipelined Execution and Control

- Instruction sequence:

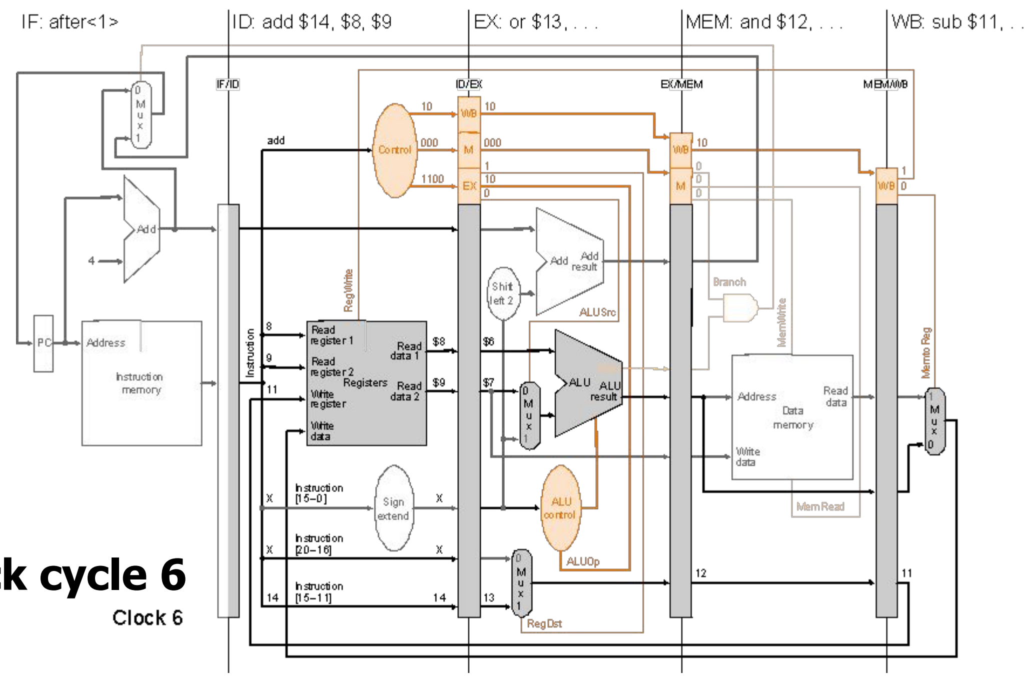
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label "after<i>" means i th instruction after `add`



Clock cycle 5

Clock 5



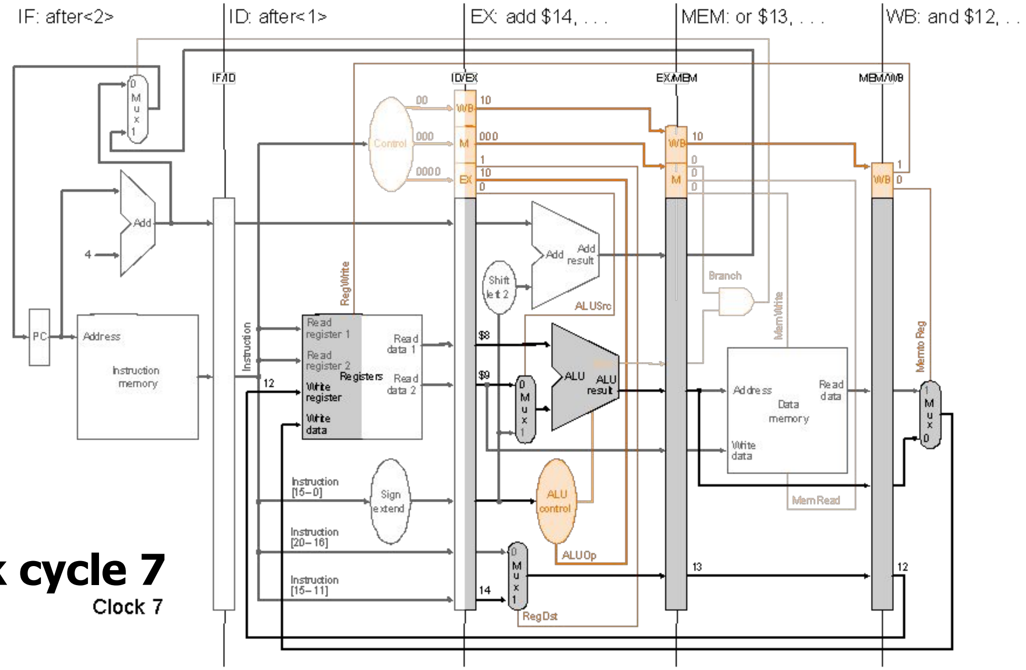
Clock cycle 6

Clock 6

Pipelined Execution and Control

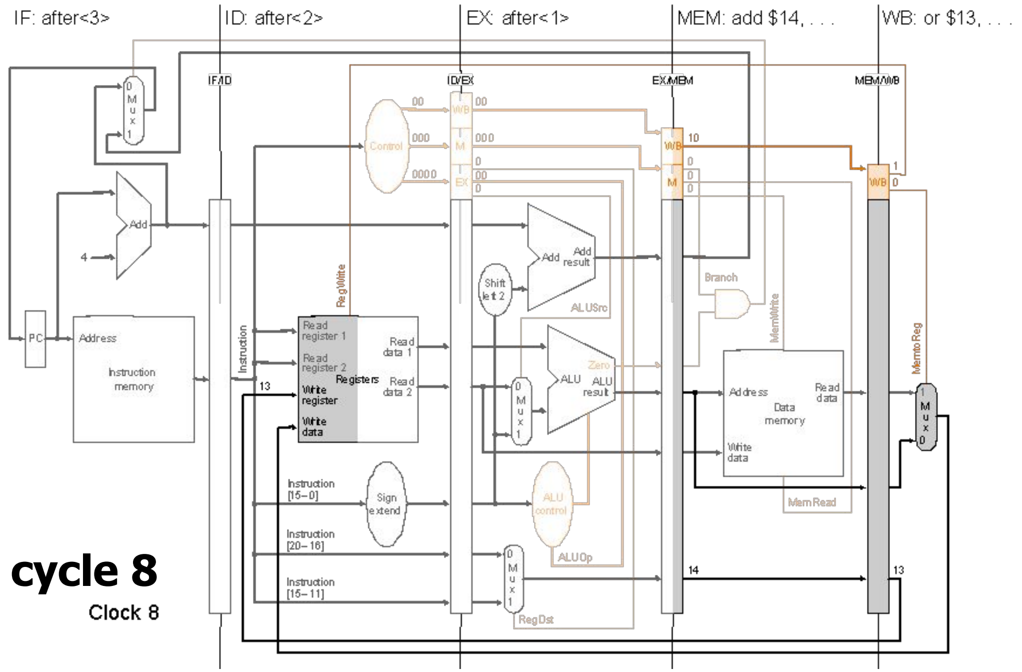
- Instruction sequence:

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



Clock cycle 7

Clock 7



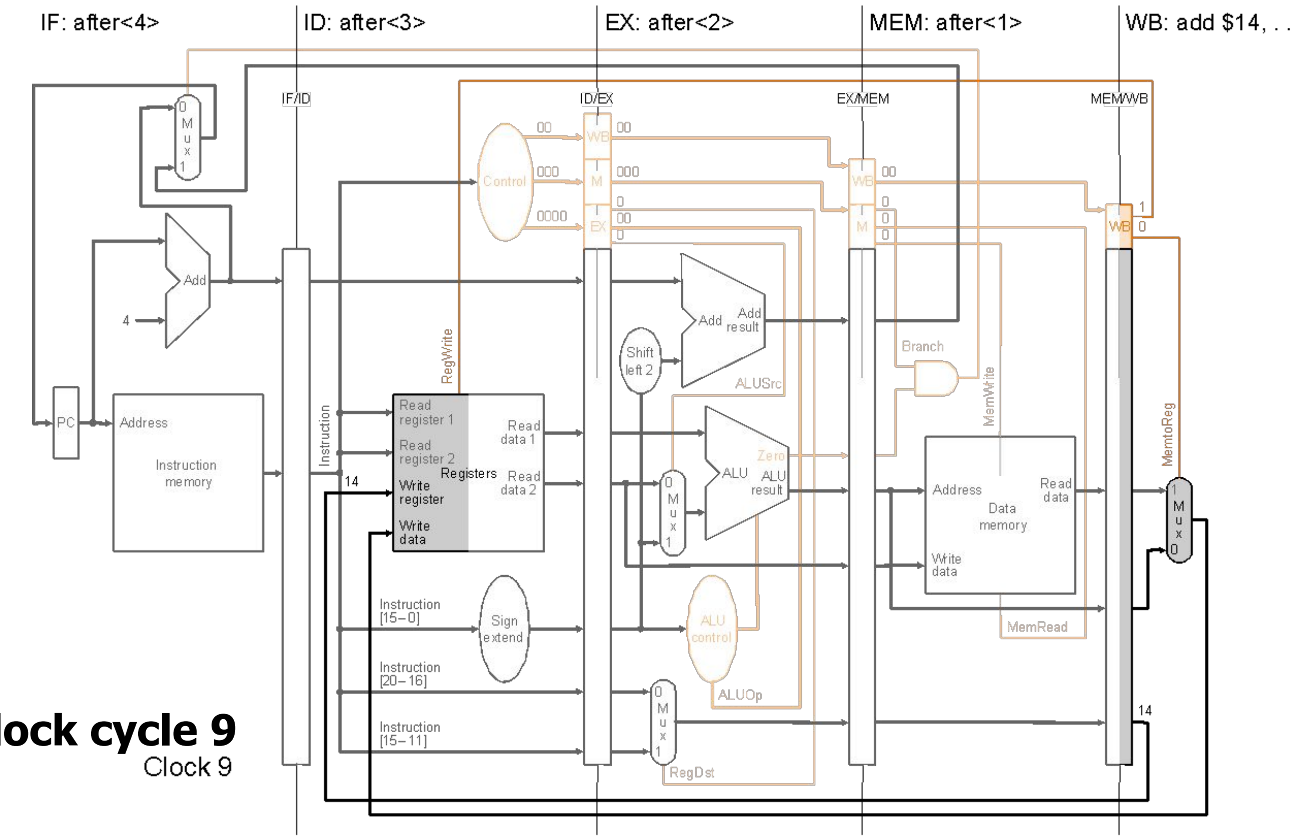
Clock cycle 8

Clock 8

Pipelined Execution and Control

- Instruction sequence:

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



Clock cycle 9

Clock 9

Pipeline Hazards

- Situations that would cause incorrect execution
- Data flow problems that arise as a result of pipelining
  - Limits the amount of parallelism, sometimes induces “penalties” that prevent one instruction per clock cycle

- **Types**
  - Structural hazards
  - Data hazards
  - Control hazards

Pipeline Hazards

- Structural hazards
  - Caused by resource contention
  - Two operations require a single piece of hardware e.g. Memory
  - Using same resource by two instructions during the same cycle
  - Structural hazards can be overcome by adding additional hardware
- Data hazards
- Control hazards

Pipeline Hazards

- Structural hazards
  - Caused by resource contention
  - Two operations require a single piece of hardware e.g. Memory
  - Using same resource by two instructions during the same cycle
  - Structural hazards can be overcome by adding additional hardware
- Data hazards
  - Instruction from one pipeline stage is “dependant” of data computed in previous pipeline stage
  - Hardware can detect dependencies between instructions
- Control hazards

Pipeline Hazards

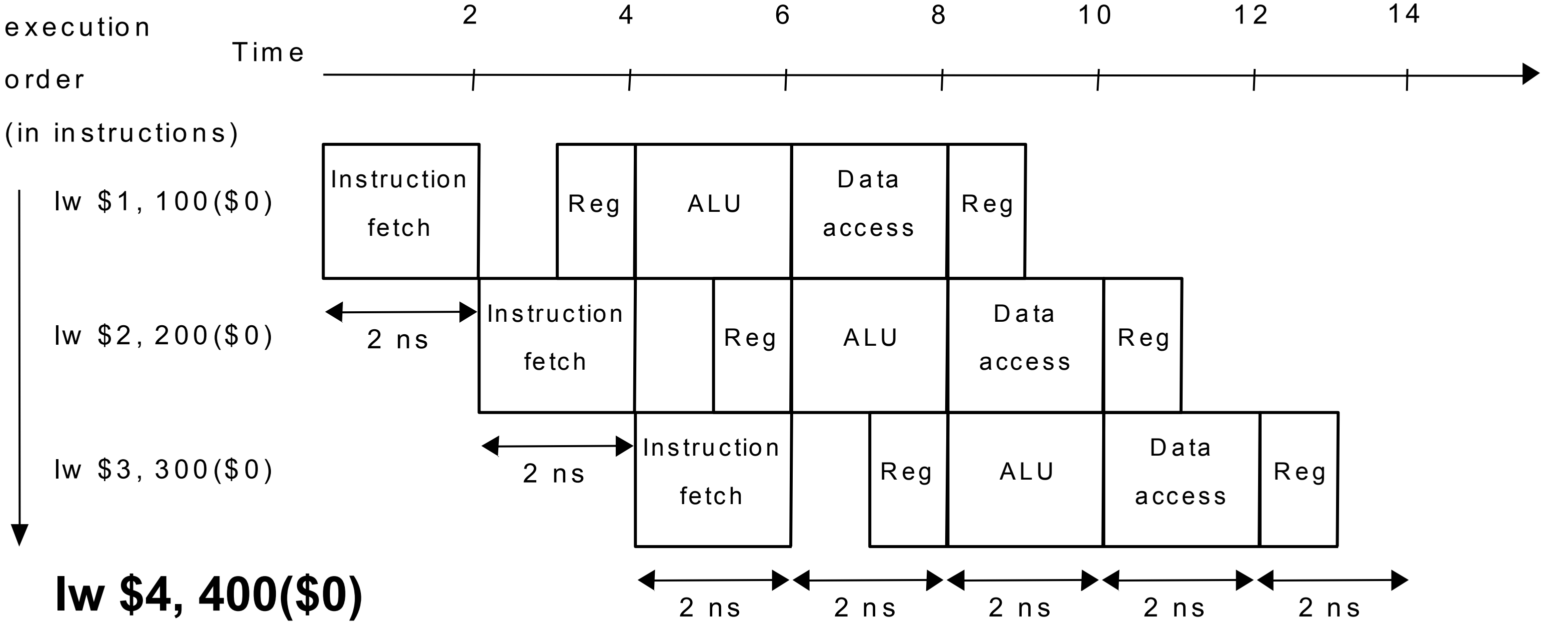
- Structural hazards
  - Caused by resource contention
  - Two operations require a single piece of hardware e.g. Memory
  - Using same resource by two instructions during the same cycle
  - Structural hazards can be overcome by adding additional hardware
- Data hazards
  - Instruction from one pipeline stage is “dependant” of data computed in previous pipeline stage
  - Hardware can detect dependencies between instructions
- Control hazards
  - Caused by instructions that change control flow (branches/jumps)
    - i.e. delays in changing the flow of control
  - Requiring subsequent instruction fetches to be predicted
    - Flushed if prediction does not hold (make sure no state change)
  - Branch hazards can use dynamic prediction/speculation, branch delay slot

Hazards

Draw pipeline diagram, and check hazard is exist or not?

- **lw $1, 100($0)**
- **lw $2, 200($0)**
- **lw $3, 300($0)**
- **lw $4, 400($0)**

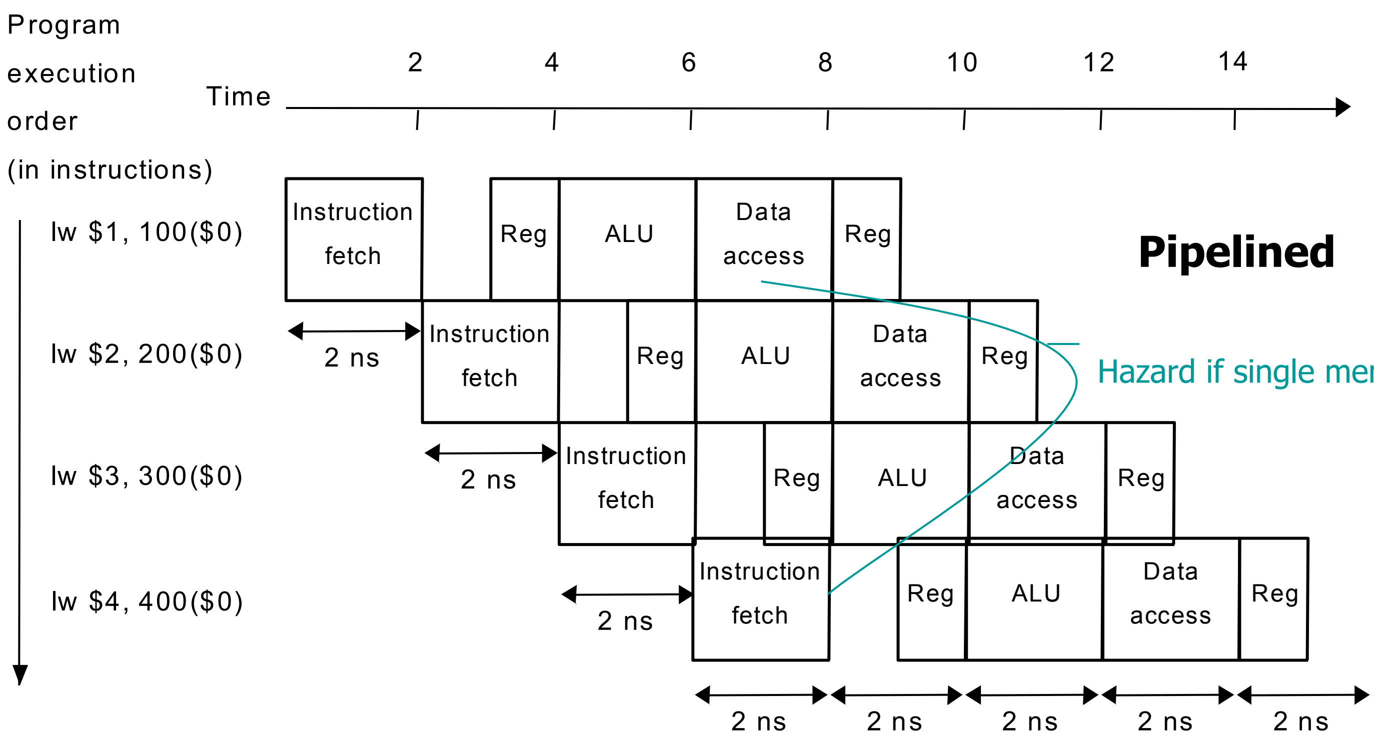
Hazard



lw $4, 400($0)

Structural Hazards

- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



Structural Hazards

- Inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle

- Attempt to use the same hardware resource by two different instructions during the same cycle

- **lw $1, 100($0)**
- **lw $2, 200($0)**
- **lw $3, 300($0)**
- **lw $4, 400($0)**

Resolving Structural Hazards

- Serious Hazard:
  - Hazard cannot be ignored
  - Easy to avoid
- Solution: Add more hardware resources (more costly)
  - Add more additional hardware to eliminate the structural hazard
  - Like, two separate memories

Data Hazards

- Dependency between instructions causes a data hazard
- Instruction needs data from the result of a previous instruction still executing in pipeline
- The dependent instructions are close to each other
  - Pipelined execution might change the order of operand access

Data Hazards Type

- **RAR (Read After Read) hazard**
  - Occurs when two instructions both read from the same register
  - **Example:**

    ADD $s1, $s2, **$s3**

    SUB $s4, $s5, **$s3**
  - Both instructions reading $s3, creating a RAR hazard
  - Don't cause a problem for the processor because reading a register doesn't change the register‘s value

Data Hazards Type

- **RAW (Read After Write) hazard**
  - Occurs when, one instruction reads a location after an earlier instruction writes new data to it
  - instruction ***j*** *tries to read a source before instruction* ***i*** *writes it, so* ***j*** *incorrectly gets the old value*
  - **Example:**

i: Add **$s3**, $s1, $s2

j: Add $s5, **$s3**, $s4

  - Result is the instruction reading stale data
  - Detected when Output_n register ($s3) and Input_{n+1} registers ($s3, $s4) contain at least one common register
  - Need to resolve

Data Hazards Type

- **WAR (Write After Read) hazard**
  - Hazards occur when the output register of an instruction is used for write after read by a previous instruction
  - Instruction **j** tries to write a destination before it is read by instruction **i**, so **i** incorrectly gets the new value
  - **Example:**

i: Add $s3, **$s1**, $s2

j: Add **$s1**, $s3, $s4

  - Detected when Input_n register and Output_{n+1} register contain at least one common operand
  - **Such hazards are rare**

Data Hazards Type

- **WAW (Write After Write) hazard**
  - Hazard occur when the output register of an instruction is used for write after written by a previous instruction
  - **Example:**

ADD **$s1**, $s2, $s3

SUB **$s1**, $s5, $s6 //Subtract writes the same register as the addition

  - If a processor executes instructions in the order that they appear in the program and uses the same pipeline for all instructions, WAR and WAW hazards do not cause the delays because of the way instructions flow through the pipeline

Hazard

Example: Draw pipeline diagram and show the hazards if any.

```
sub   $s2, $t1, $t3
add   $s4, $s2, $t5
or    $s6, $t3, $s2
and   $s7, $t4, $s2
sw    $t8, 10($s2)
```

RAW Data Hazard Solutions

Example:

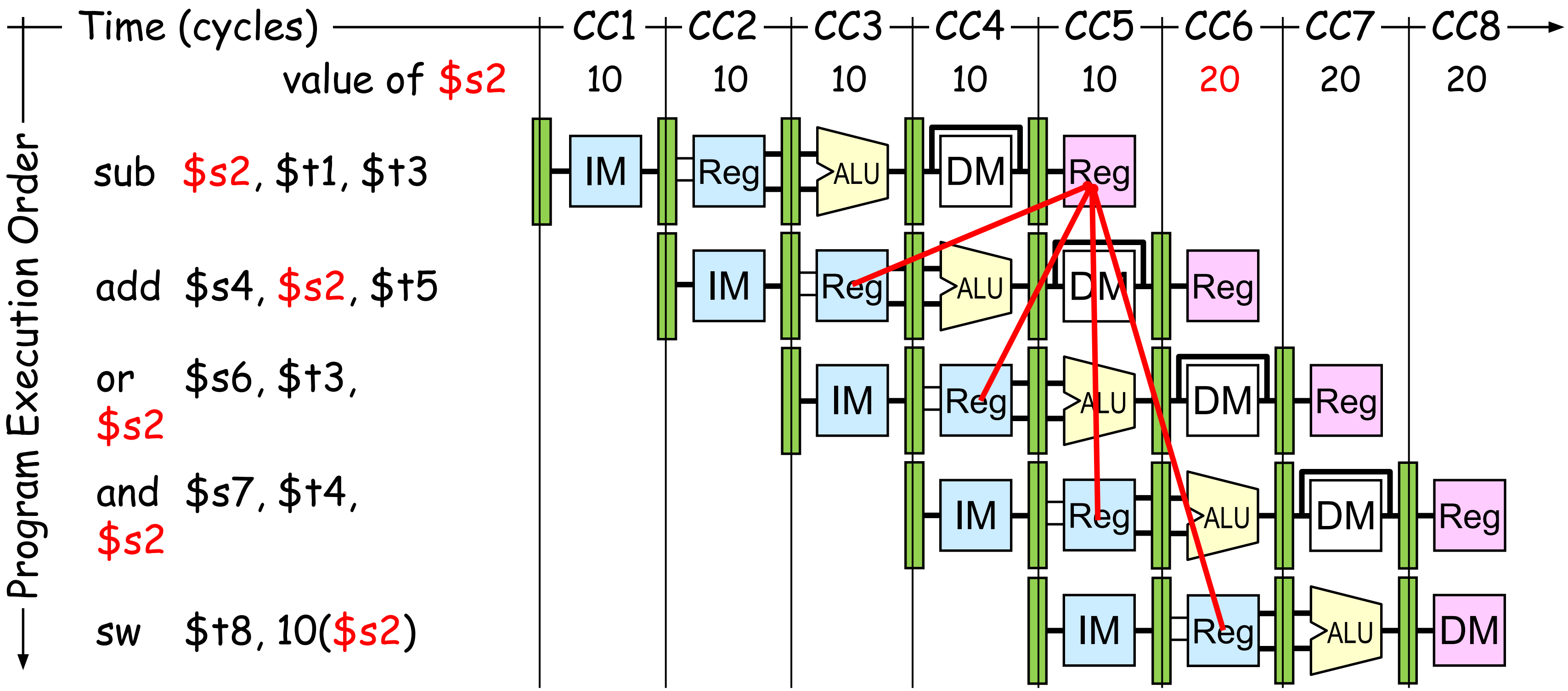
```
sub   $s2, $t1, $t3
add   $s4, $s2, $t5
or    $s6, $t3, $s2
and   $s7, $t4, $s2
sw    $t8, 10($s2)
```

RAW Data Hazard Solutions

Example:

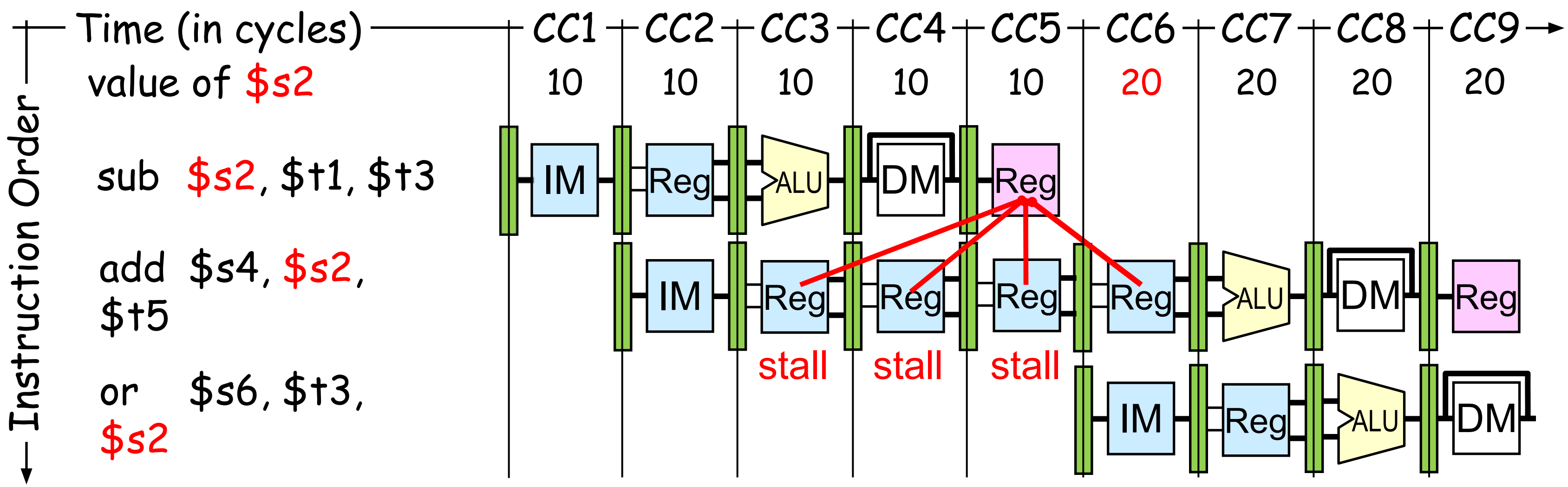
```
sub  $s2, $t1, $t3
add  $s4, $s2, $t5
or $s6, $t3, $s2
and  $s7, $t4, $s2
sw $t8, 10($s2)
```

Example of a RAW Data Hazard



- Result of sub is needed by add, or, and, & sw instructions
- Instructions add & or will read old value of $s2 from reg file
- During CC5, $s2 is written at end of cycle, old value is read
  - But, can be eliminated by considering in first half write to register and in second half read from register

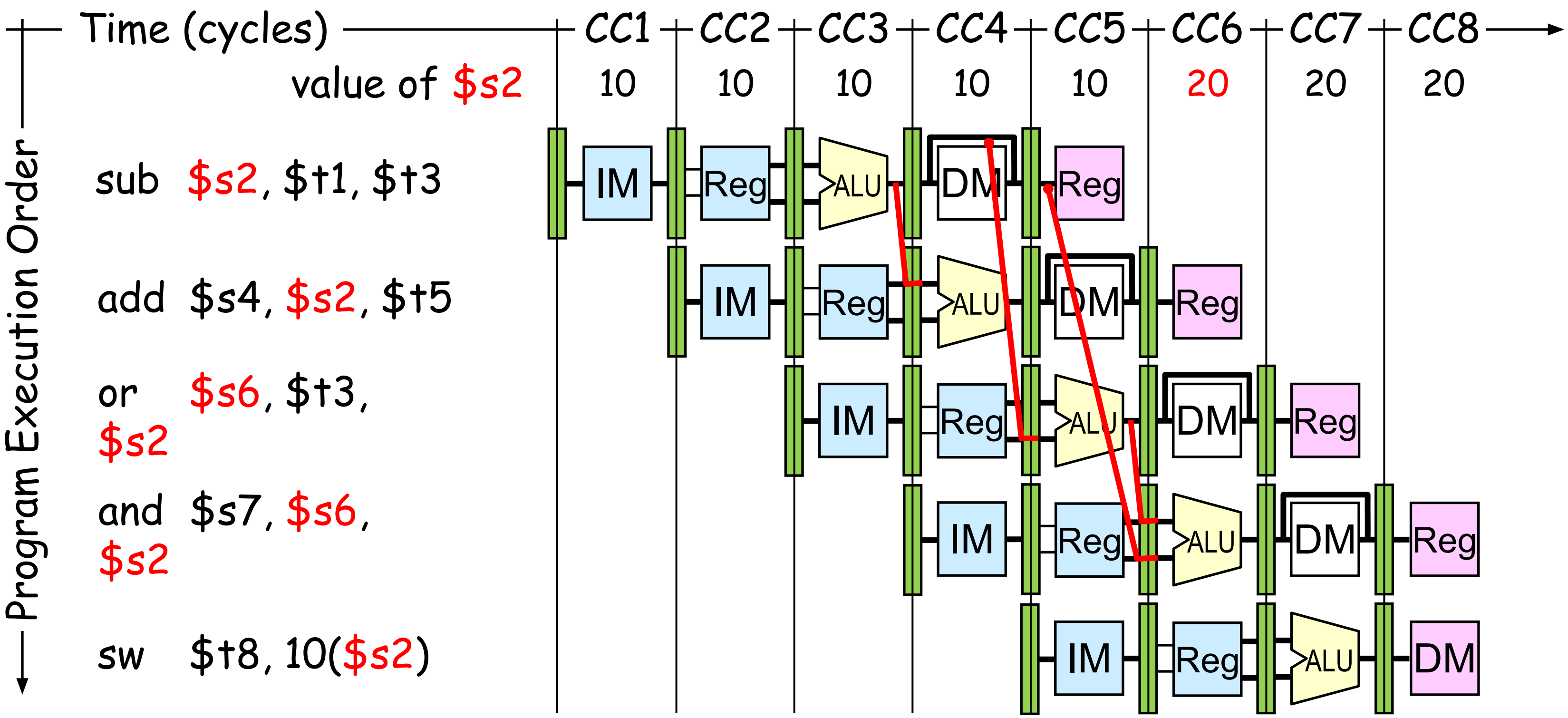
Solution 1: Stalling the Pipeline



- Three stall cycles during **CC3** thru **CC5** (wasting 3 cycles)
  - Stall cycles delay execution of **add** & fetching of **or** instruction
- The **add** instruction cannot read **$s2** until beginning of **CC6**
  - The **add** instruction remains in the **Instruction register** until **CC6**
  - The **PC register** is not modified **until beginning of CC6**

Solution 2: Forwarding ALU Result

- The **ALU result** is **forwarded** (fed back) to the **ALU input**
  - No bubbles are inserted into the pipeline and **no cycles are wasted**
- ALU result is forwarded from **ALU, MEM,** and **WB** stages



RAW Data Hazard Solutions

- For the following code, detect the hazard, if any.

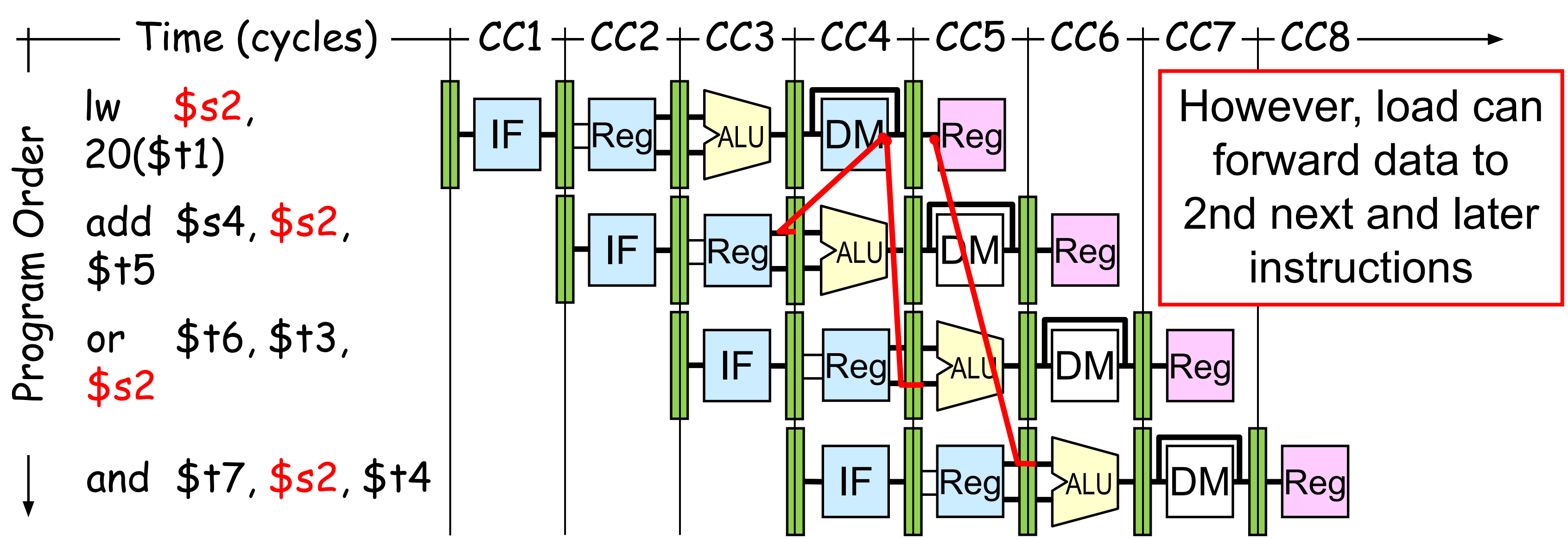
lw $s0, 20($t1)

sub $t2, $s0,$t3

- Is forwarding useful?
- If an R-type instruction following a load uses the result of the load – called *load-use data hazard*

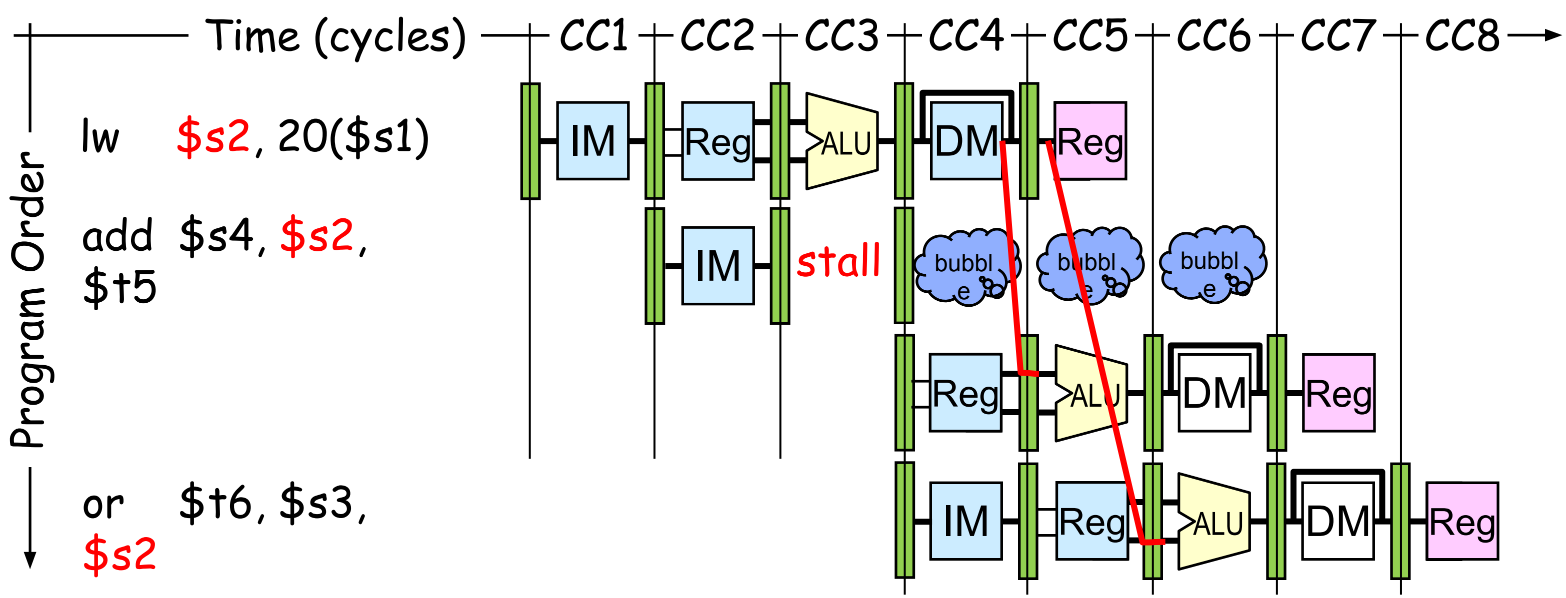
RAW Data Hazard Solutions

- Unfortunately, not all data hazards can be forwarded
  - **Load** has a delay that cannot be eliminated by forwarding
- In the example shown below …
  - The **LW** instruction does not read data until end of CC4
  - Cannot forward data to **ADD** at end of CC3 - **NOT possible**



Stall the Pipeline for one Cycle

- ***ADD*** instruction depends on ***LW*** → stall at CC3
  - Allow **Load** instruction in ***ALU*** stage to proceed
  - Freeze **PC** and **Instruction** registers (NO instruction is fetched)
  - Introduce a **bubble** into the ***ALU*** stage (bubble is a NO-OP)
- **Load** can forward data to next instruction after delaying it

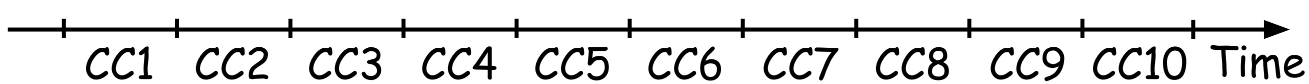


Showing Stall Cycles

- Stall cycles can be shown on instruction-time diagram
- Hazard is detected in the Decode stage
- Stall indicates that instruction is delayed
- Instruction fetching is also delayed after a stall
- **Example:**

Data forwarding is to be shown using **green arrows**

```
lw   $s1, ($t5)
lw   $s2, 8($s1)
add  $v0, $s2, $t3
sub  $v1, $s2, $v0
```



Showing Stall Cycles

- Stall cycles can be shown on instruction-time diagram
- Hazard is detected in the Decode stage
- Stall indicates that instruction is delayed
- Instruction fetching is also delayed after a stall
- **Example:**

Data forwarding is shown using **green arrows**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
lw $s1, ($t5)	IF	ID	EX	MEM	WB					
lw $s2, 8($s1)		IF	**Stall**	ID	EX	MEM	WB			
add $v0, $s2, $t3				IF	**Stall**	ID	EX	MEM	WB	
sub $v1, $s2, $v0						IF	ID	EX	MEM	WB

Time

RAW Data Hazard Solutions

- Software Solution
  - Reordering Code to Avoid Pipeline Stall
- **Example:**

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

RAW Data Hazard Solutions

- **Example:**

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

- **Reordered code:**

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

Interchanged

Example

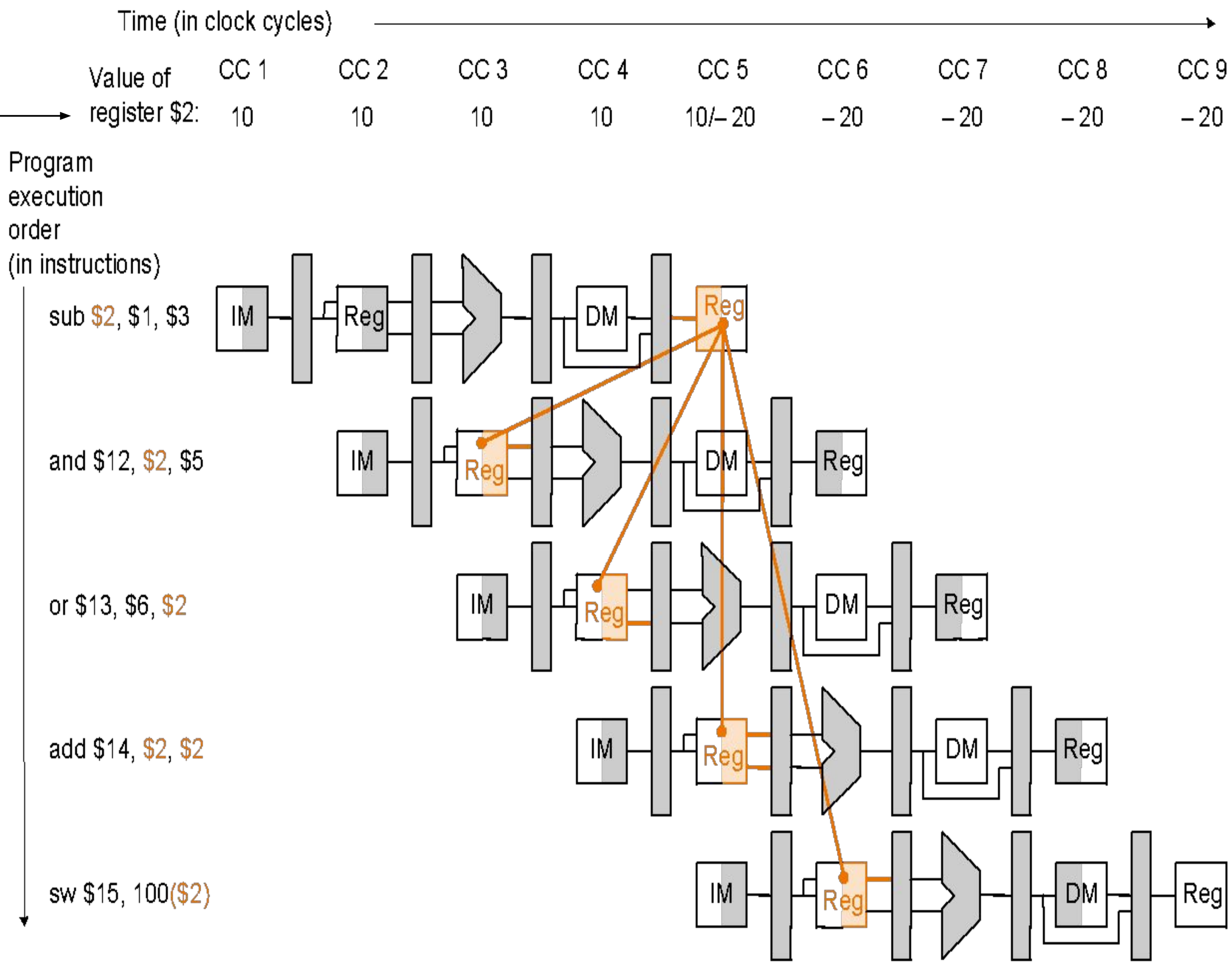
- Draw the pipelining execution for the following code and detect and resolve the hazard, if any.

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

$2 = 10 before `sub`
$2 = -20 after `sub`

Data Hazards and Forwarding

$2 = 10 before `sub`;
$2 = -20 after `sub`



Example: Software Solution

- By *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,
- Or, if such rearrangement is not possible, *insert* `nop`s

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

```
sub  $2,  $1, $3
lw   $10, 40($3)
slt  $5,  $6, $7
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

or

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

- **Such compiler solutions may not always be possible, and `nop`s slow the machine down**

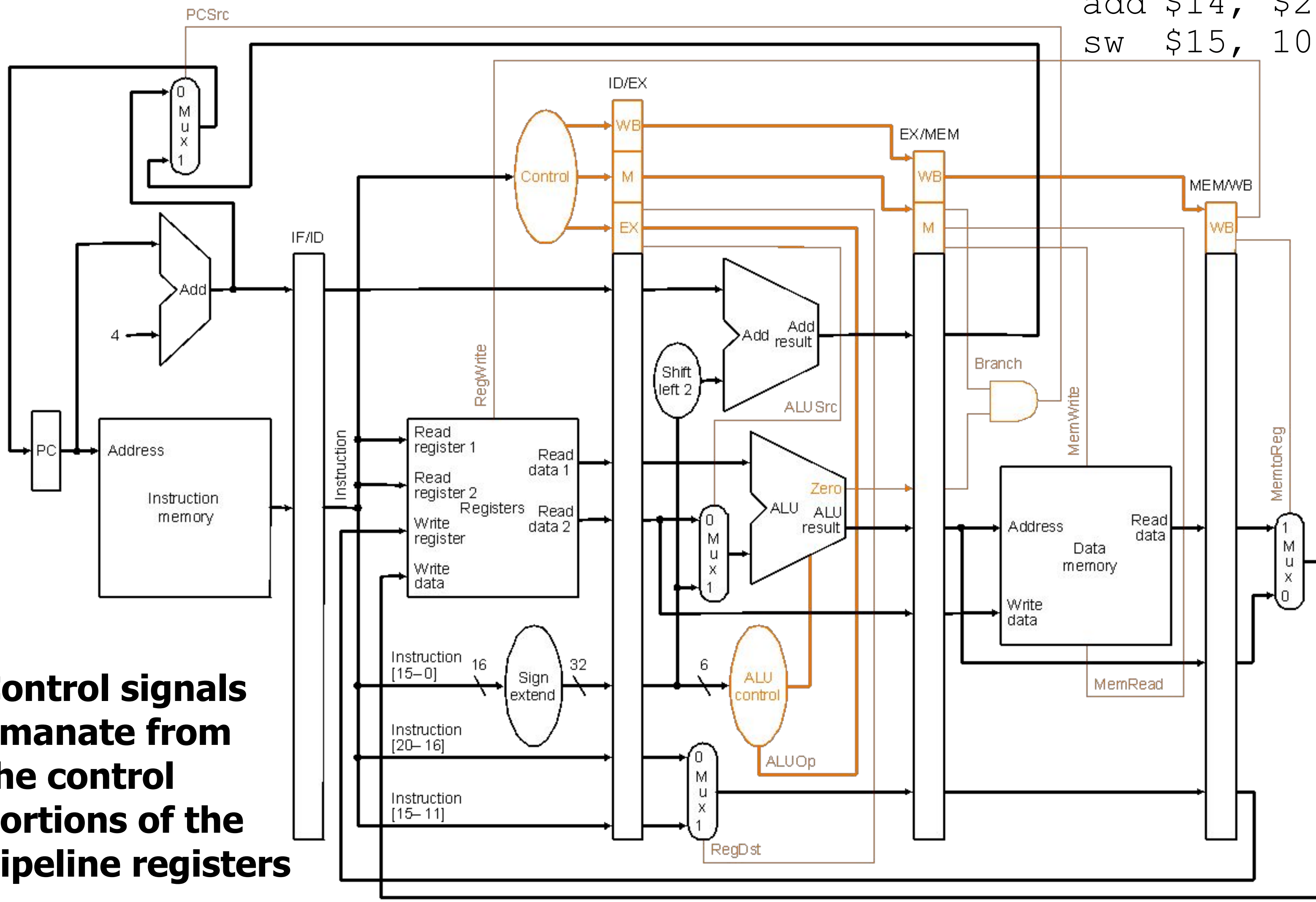
MIPS: `nop` = "no operation" = 00…0 (32bits) = `sll $0, $0, 0`

RAW Hazard-Hardware Solution

- Forwarding
- **Idea:** *Use intermediate data*, do not wait for result to be finally written to the destination register.
- **Two steps:**
  1. *Detect* data hazard
  2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)

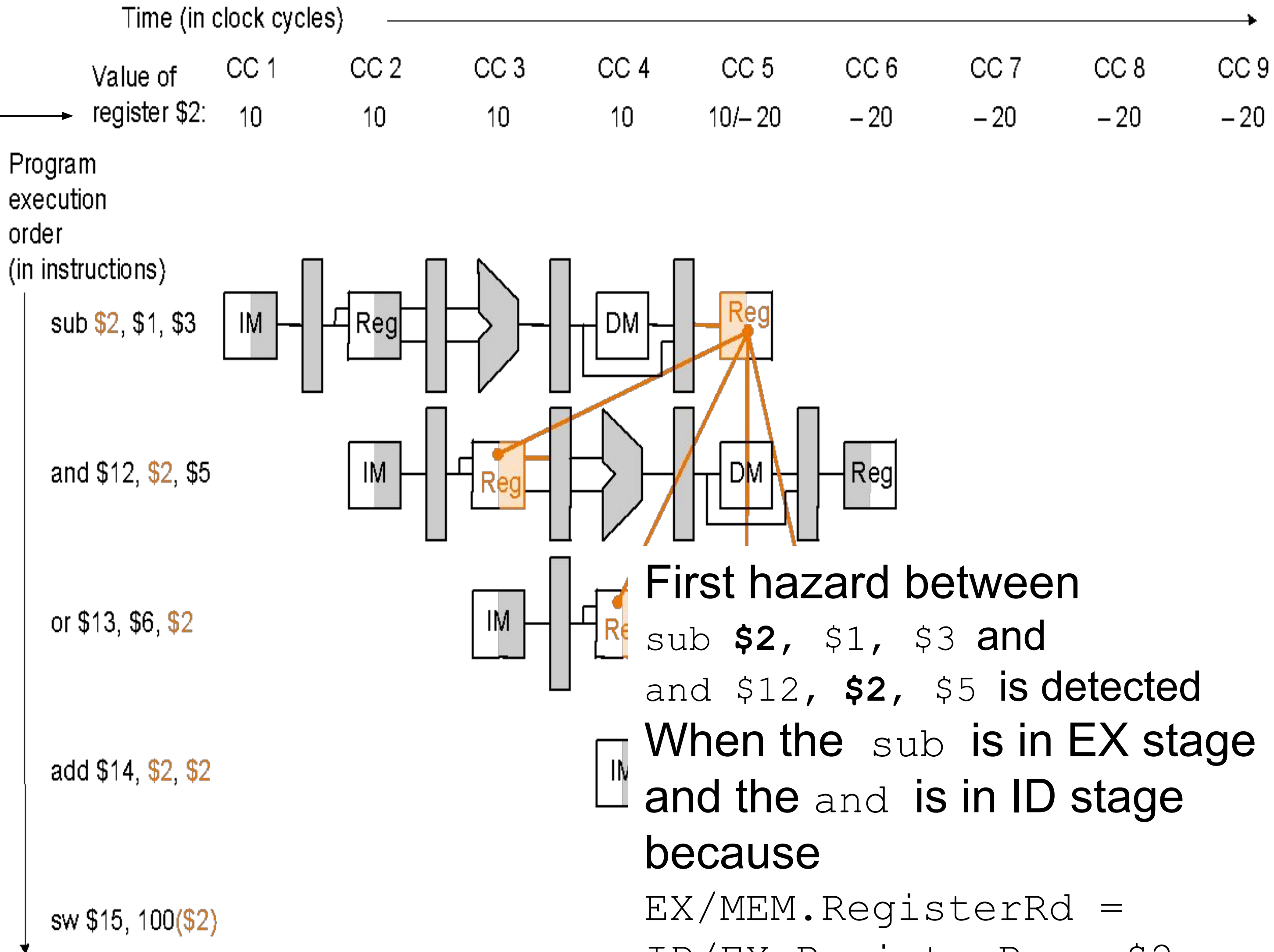
```
sub **$2**,  $1, $3
and $12, **$2**, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```



Control signals emanate from the control portions of the pipeline registers

Data Hazards and Forwarding

$2 = 10 before `sub`;
$2 = -20 after `sub`



First hazard between
`sub $2, $1, $3` and
`and $12, $2, $5` is detected
When the `sub` is in EX stage and the `and` is in ID stage because
`EX/MEM.RegisterRd = ID/EX.RegisterRs = $2`

Hazard Detection

- Hazard conditions:

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

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

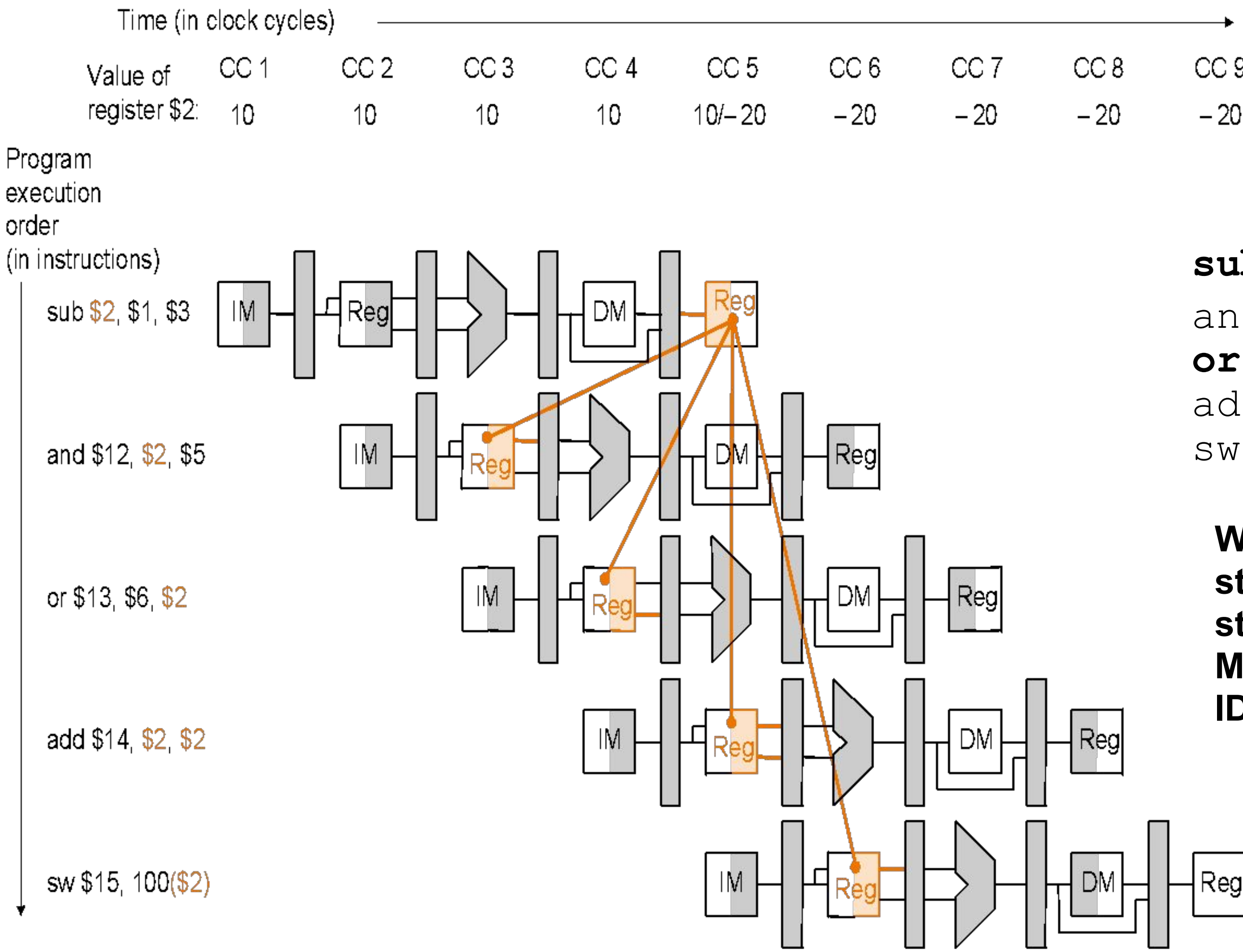
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

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

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

  - Eg., in the example, first hazard between
    - `sub $2, $1, $3` and
    - `and $12, $2, $5` is detected
  - When the `sub` is in EX stage and the `and` is in ID stage because
    - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`

Hazard Detection




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

When the sub is in WB stage and the or is in ID stage
MEM/WB.RegisterRd = ID/EX.RegisterRt = $2 (2b)

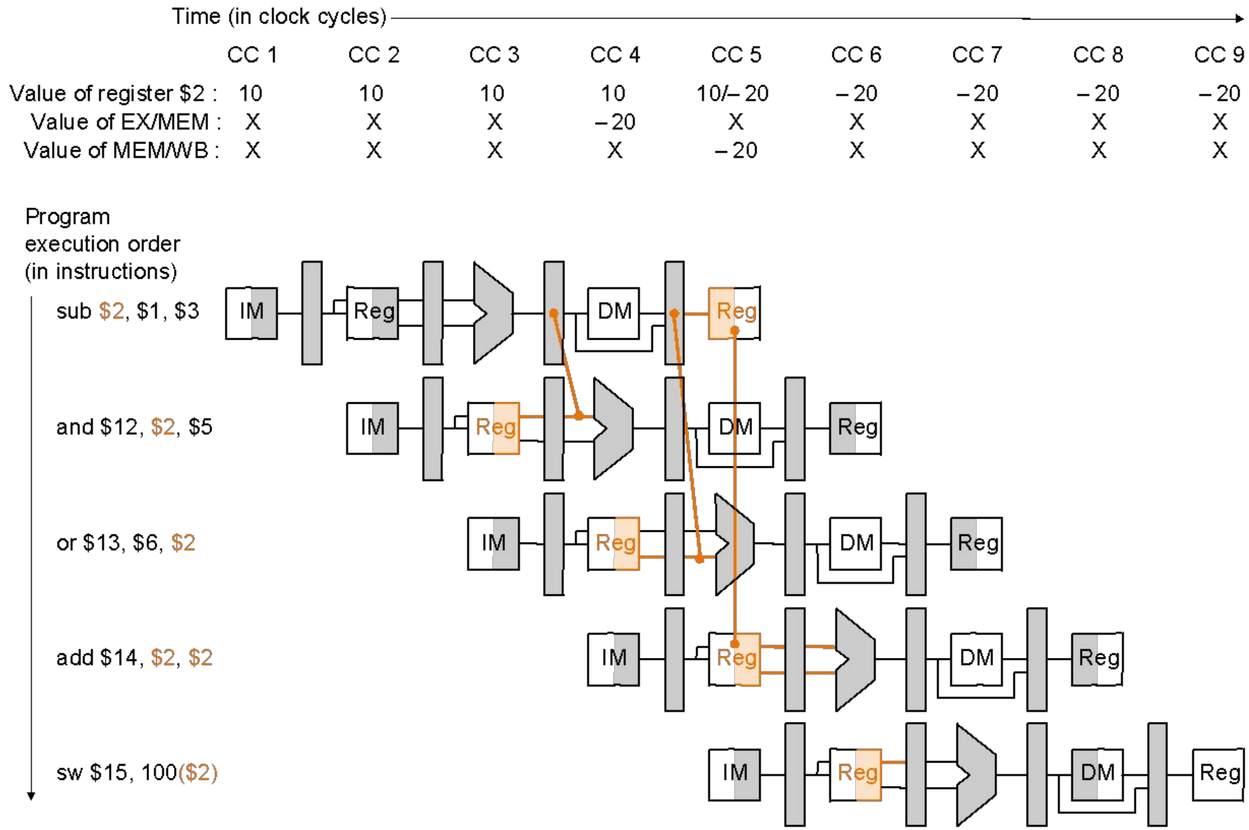
Hazard Detection

- Whether to forward also depends on:
  - *if the later instruction is going to write a register*
    - if *not*, no need to forward
  - *if the destination register of the later instruction is $0*
    - no need to forward value ($0 is always 0 and never overwritten)

Data Forwarding

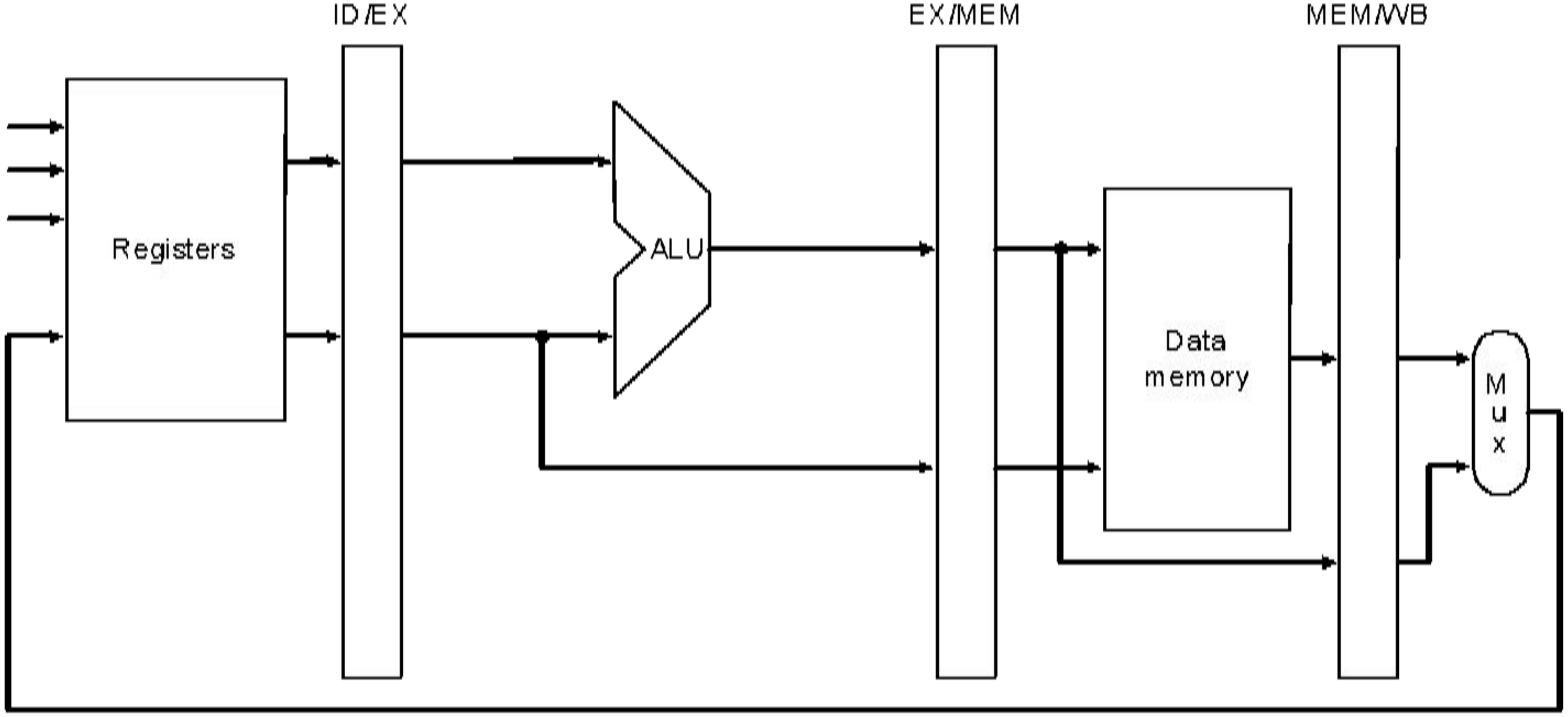
- Plan:
  - *Allow inputs to the ALU not just from ID/EX, but also later pipeline registers*, and
  - *Use multiplexors and control signals to choose appropriate inputs* to ALU

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



Dependencies between pipelines move forward in time

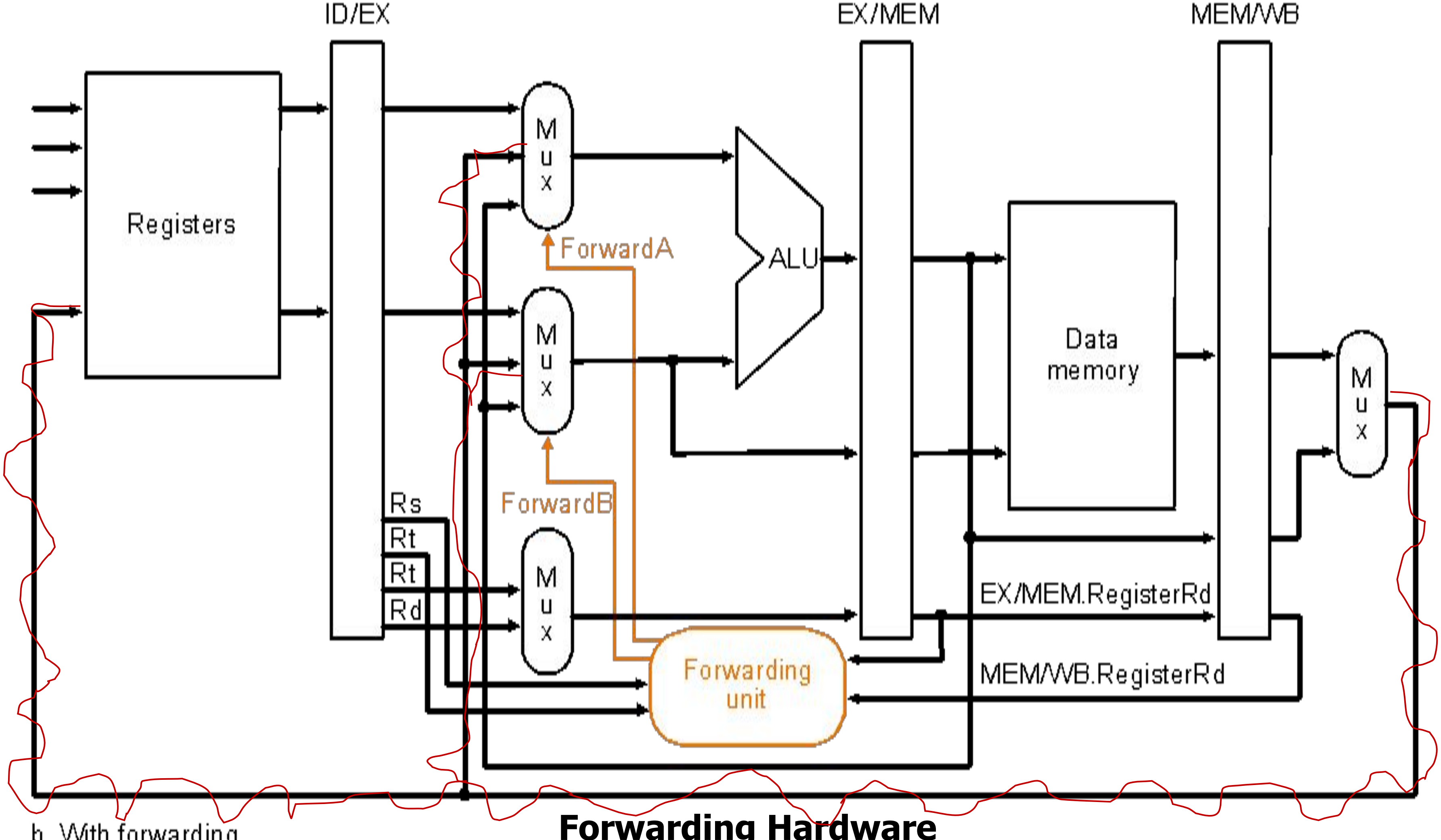
Datapath Before Forwarding Hardware



a. No forwarding

Datapath after adding forwarding hardware

Datapath after adding Forwarding Hardware



Forwarding Hardware

Forwarding Hardware: Multiplexor Control

Mux control	Source	Explanation
Forward**A** = 00	ID/EX	The first ALU operand comes from the register file
Forward**A** = 10	EX/MEM	The first ALU operand is forwarded from prior ALU result
Forward**A** = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
Forward**B** = 00	ID/EX	The second ALU operand comes from the register file
Forward**B** = 10	EX/MEM	The second ALU operand is forwarded from prior ALU result
Forward**B** = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

Depending on the selection in the rightmost multiplexor (see datapath with control diagram)

Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. **EX hazard**

if (EX/MEM.RegWrite // if there is a write…
and (EX/MEM.RegisterRd ≠ 0) // to a non-$0 register…
and (EX/MEM.RegisterRd = ID/EX.RegisterR**s**)) // matches, then
ForwardA = 10

if (EX/MEM.RegWrite // if there is a write…
and (EX/MEM.RegisterRd ≠ 0) // to a non-$0 register…
and (EX/MEM.RegisterRd = ID/EX.RegisterR**t**)) // matches then…
ForwardB = 10

Data Hazard: Detection and Forwarding

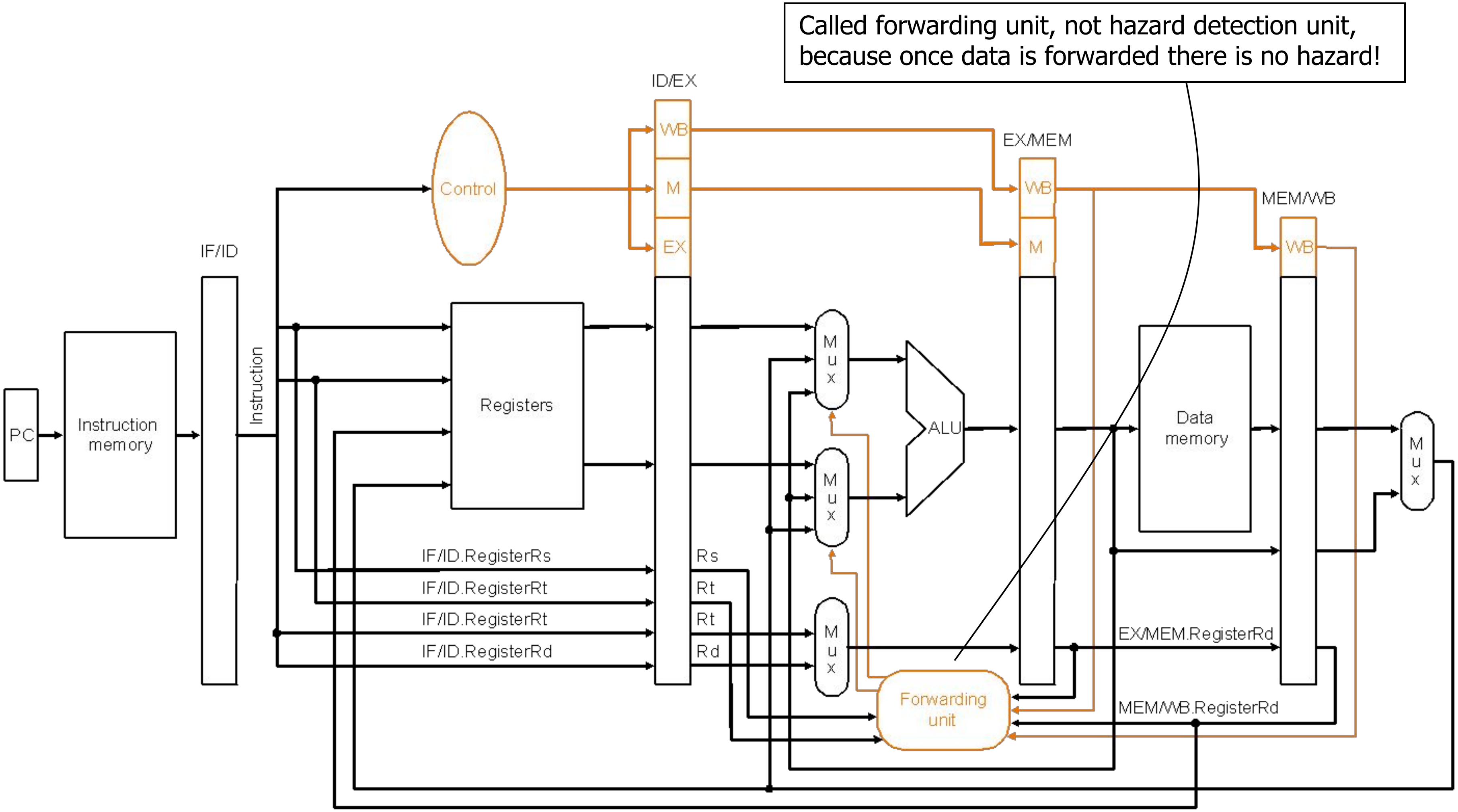
2. **MEM hazard**

```
if (      MEM/WB.RegWrite                                  // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                           // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs )  // and not already a
//register match with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )  // but match with later
//pipeline register, then…
ForwardA = 01

if (      MEM/WB.RegWrite                                  // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                           // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt )      // and not already a
// register match with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRt ) )  // but match with later
pipeline register, then…
ForwardB = 01
```

This check is necessary, e.g., for sequences such as add $1, $1, $2; add $1, $1, $3; add $1, $1, $4; (array summing…), where an earlier pipeline (EX/MEM) register has more recent data

Forwarding Hardware with Control



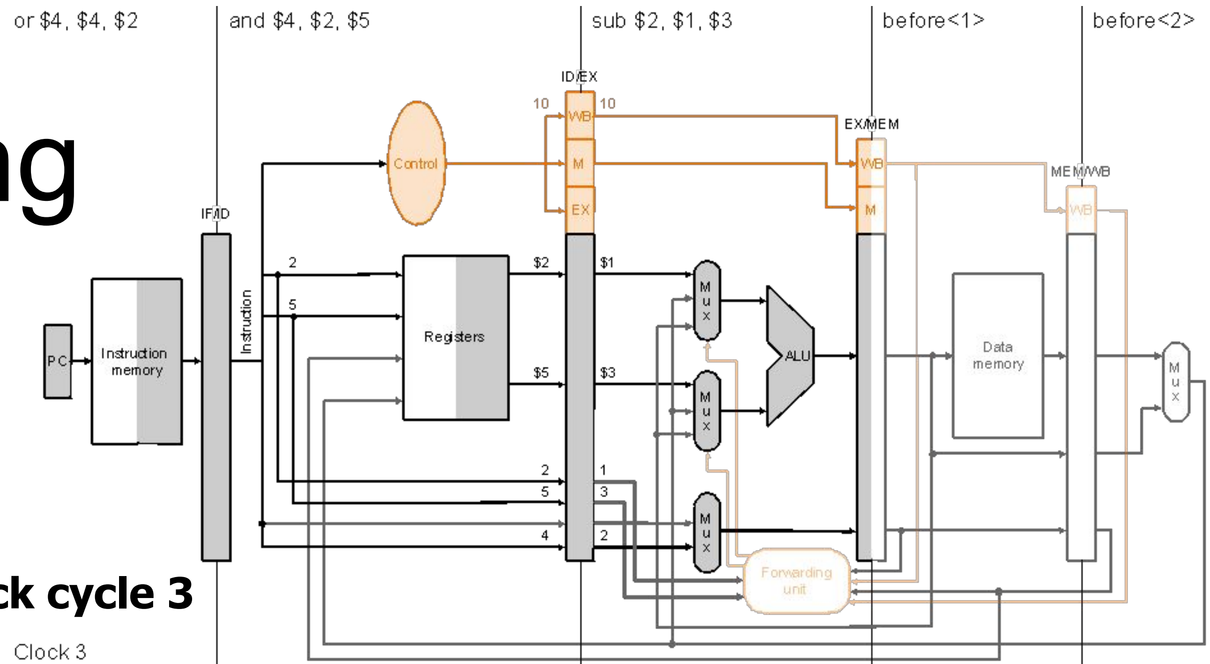
Datapath with forwarding hardware and control wires – certain details, e.g., branching hardware, are omitted to simplify the drawing

Note: so far we have only handled forwarding to R-type instructions…!

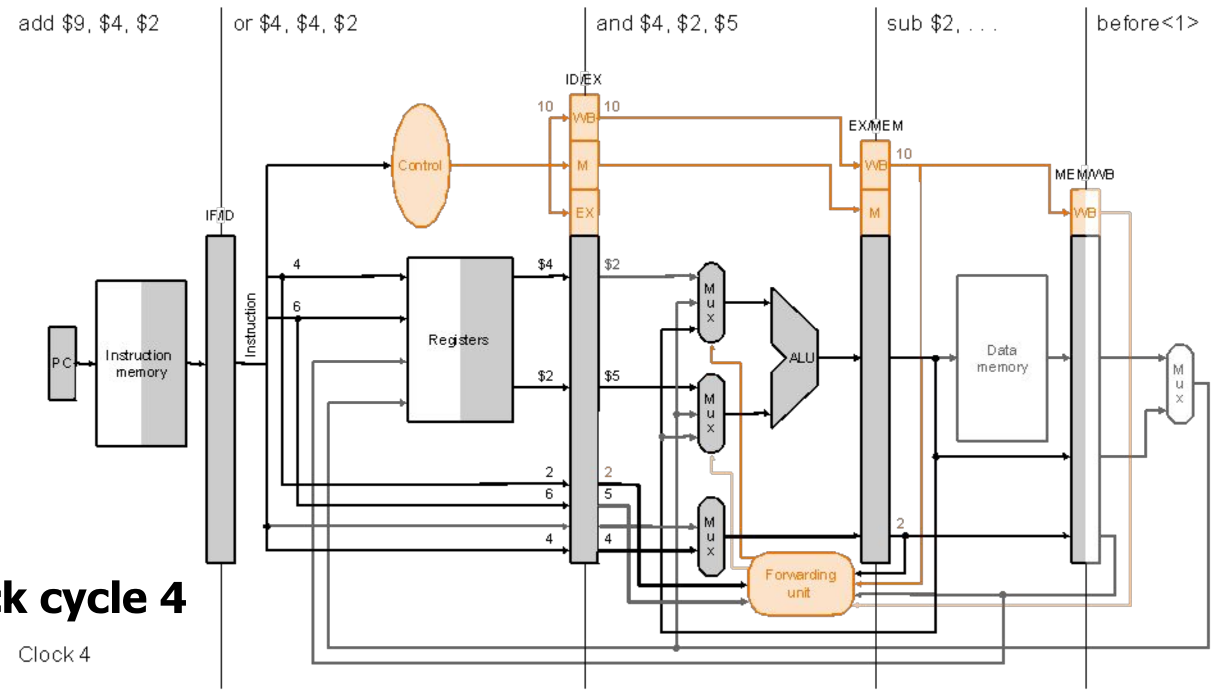
Forwarding

- Execution example:

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



Clock cycle 3



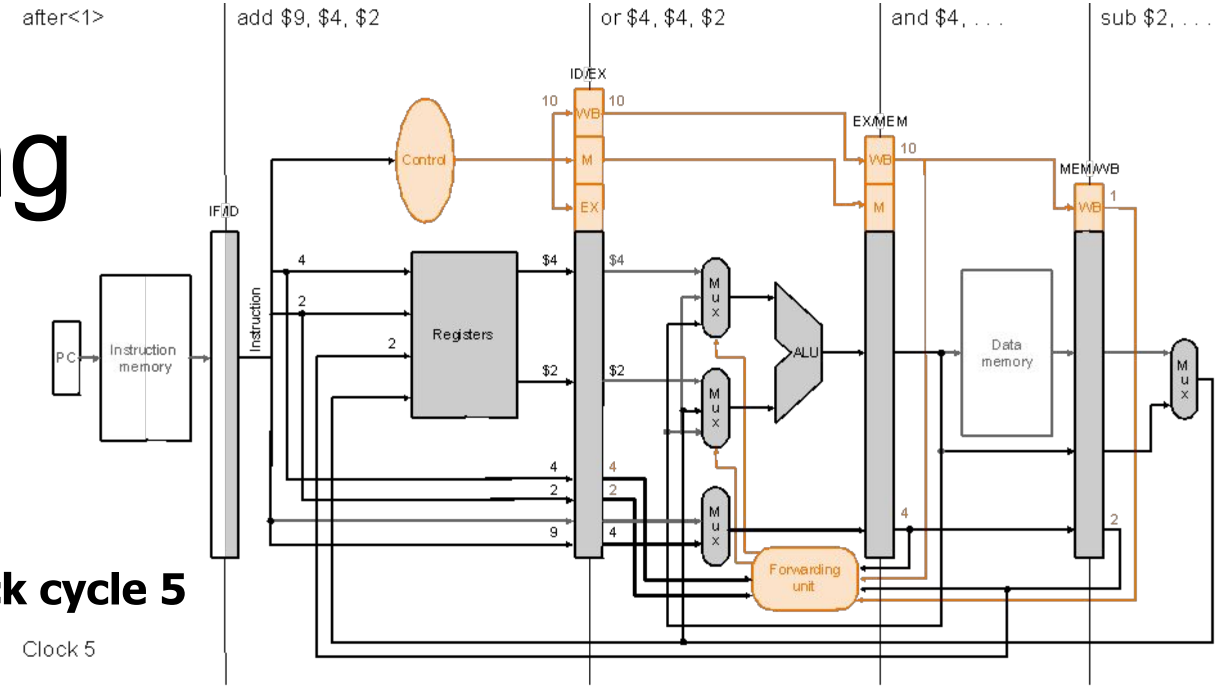
Clock cycle 4

Forwarding

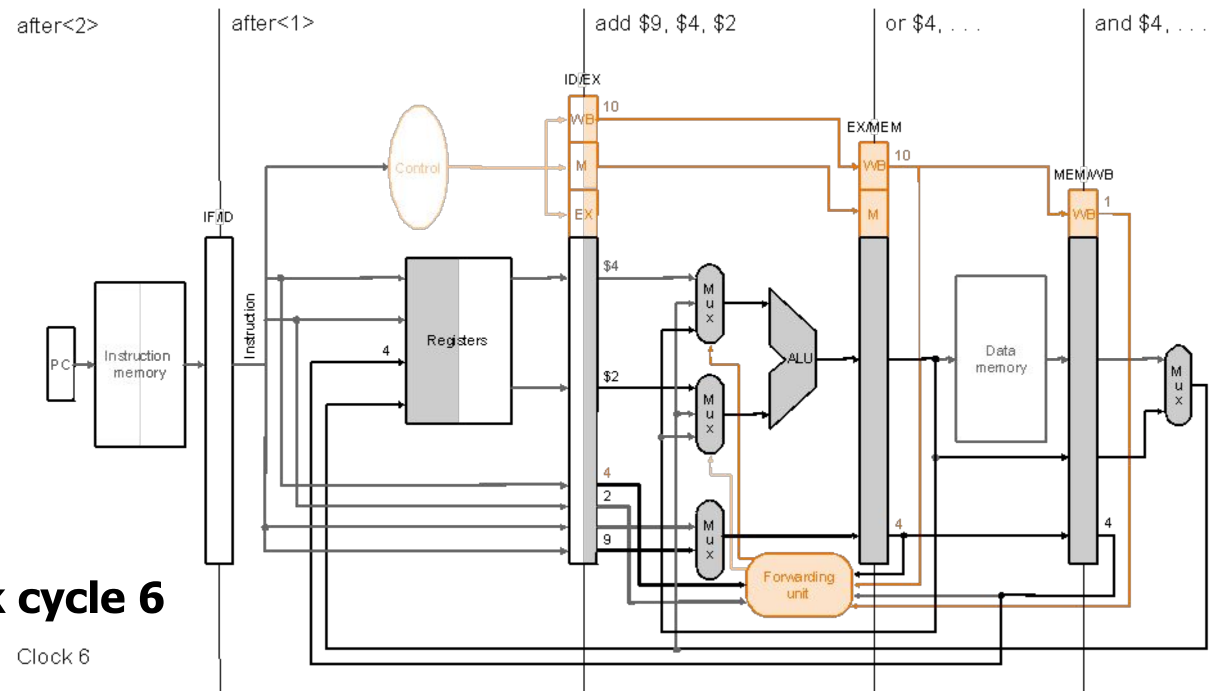
- Execution example (cont.):

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

Clock cycle 5



Clock cycle 6

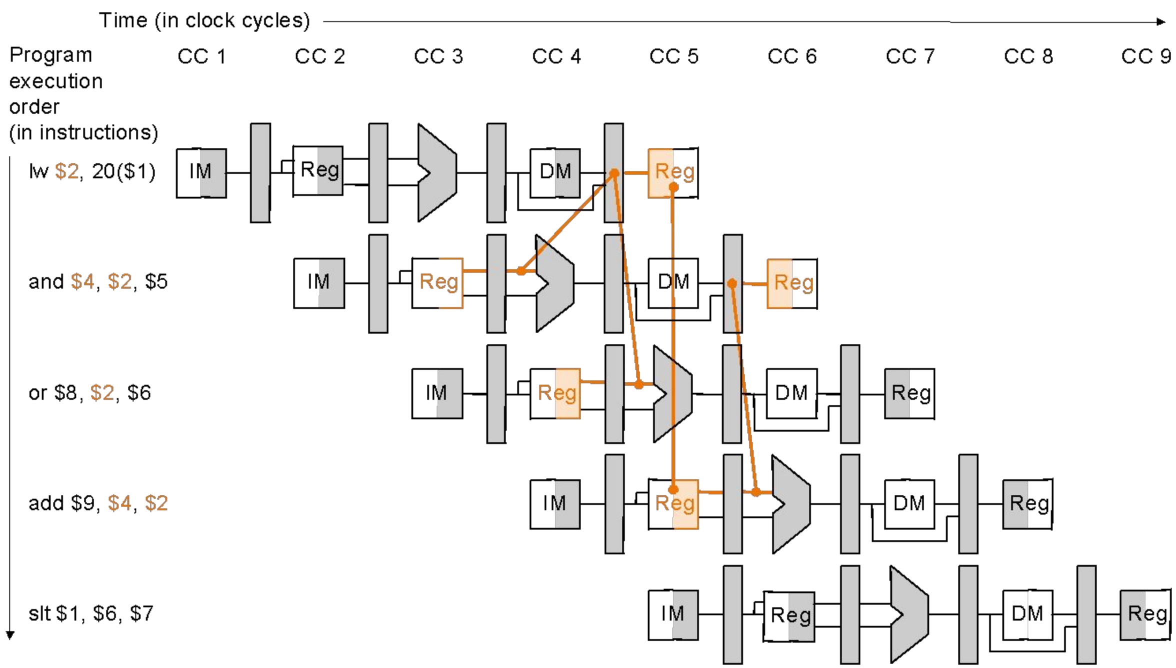


Data Hazards and Stalls

- Load word can cause a hazard:
  - An instruction tries to read a register following a load instruction that writes to the same register

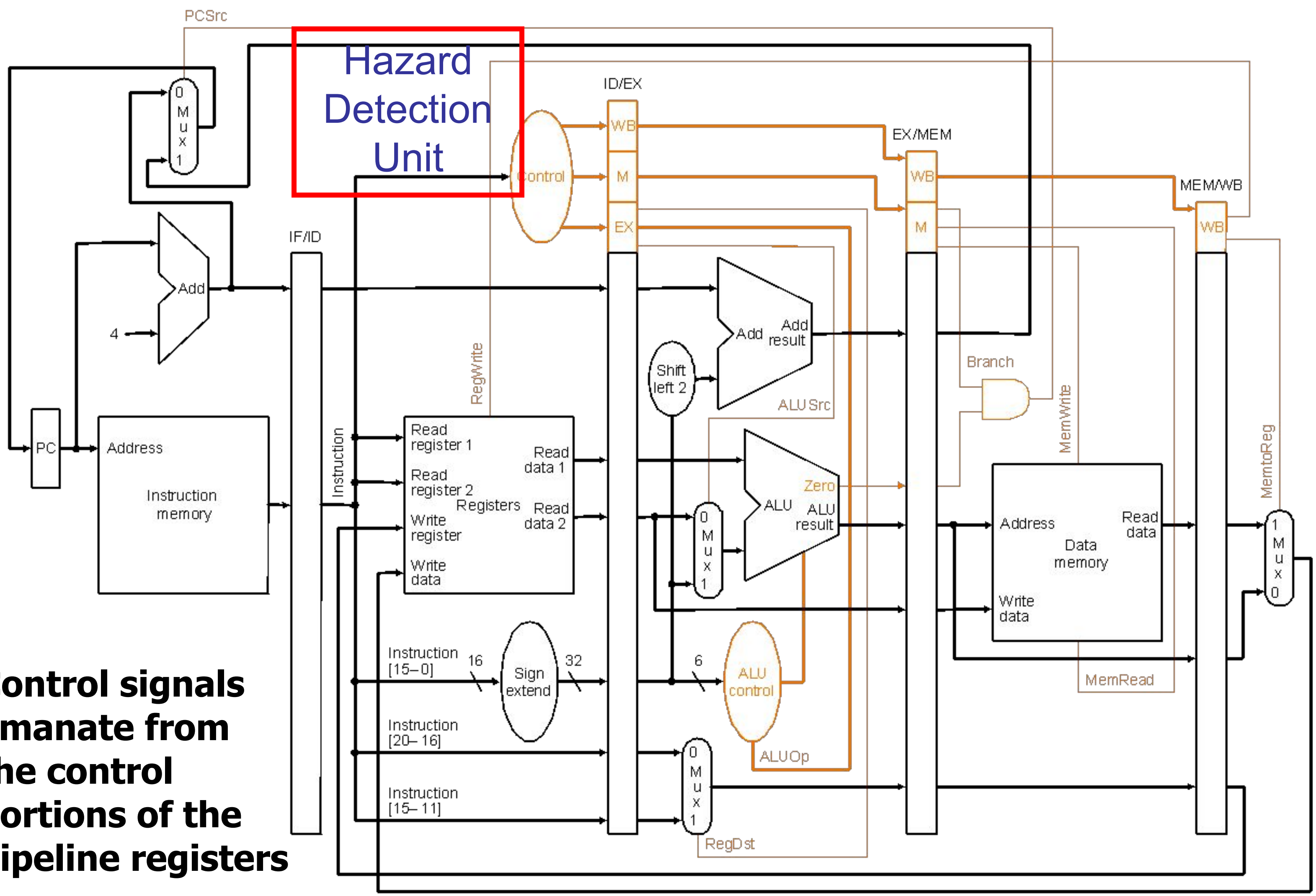
```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

As even a pipeline dependency goes backward in time forwarding will not solve the hazard



Therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

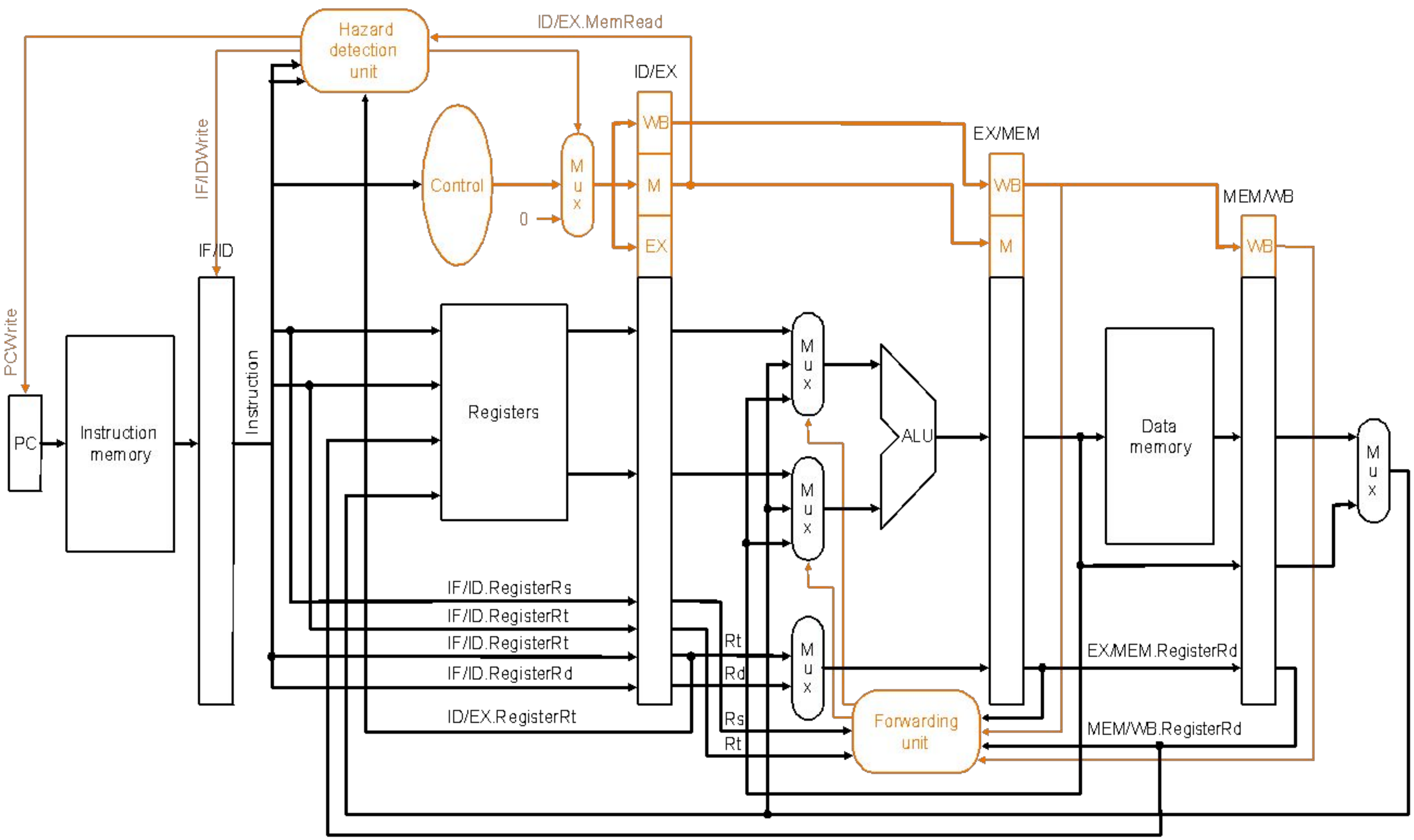
Hazard Detection Logic to Stall

- Hazard detection unit implements the following check at **ID stage**, if to stall by inserting a bubble into the pipeline by changing the EX, MEM and WB control fields of the ID/EX pipeline register to 0

```
if ( ID/EX.MemRead                              // if the instruction in the EX stage is a load…
    and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )      // and the destination register
      or  ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )  // matches either source register
                 STALL          // of the instruction in the ID stage, then…stall the pipeline
```

- Insert a bubble into the EX stage after a load instruction
  - Bubble is a no-op that wastes one clock cycle
  - By deasserting all nine control signals (setting them to 0) in EX, MEM and WB stages
    - Restrict the write operation to any register or memory

Hazard Detection Unit



Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing

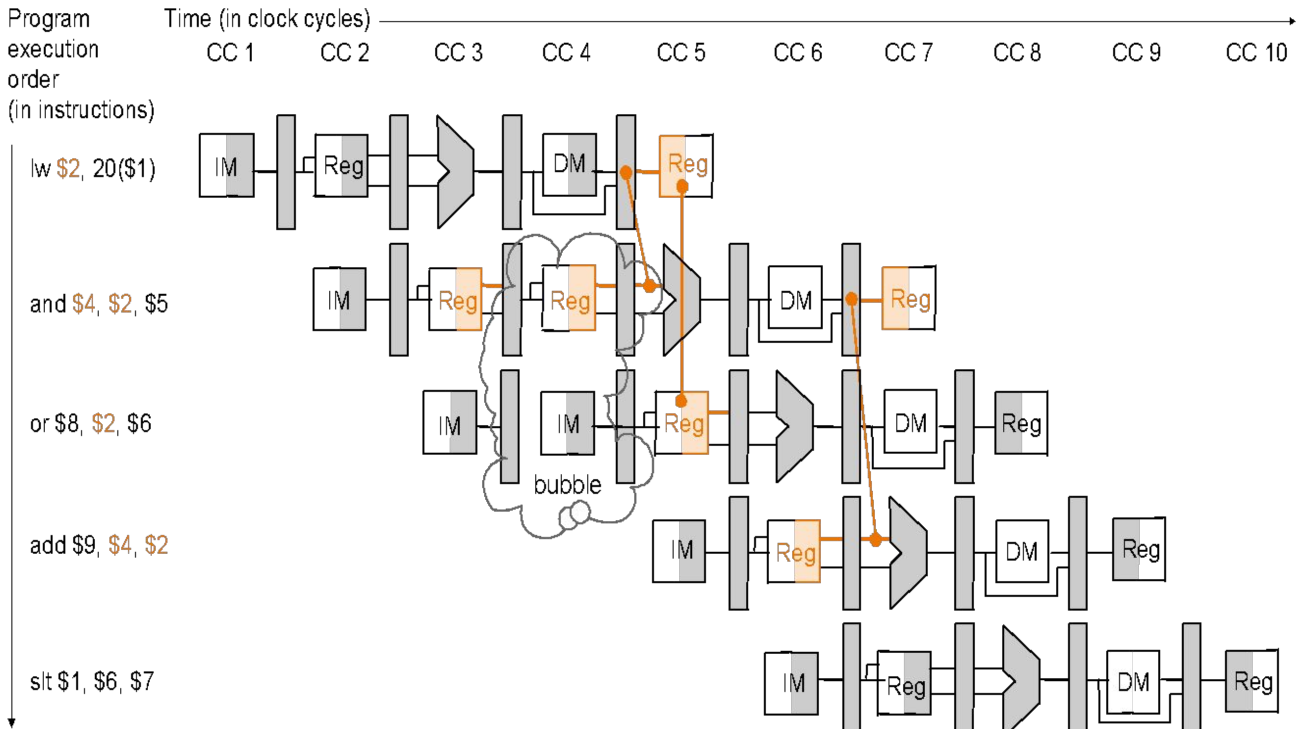
Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- What the hardware does to stall the pipeline 1 cycle:
  - *does not let the IF/ID register change* (*disable write*!) – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware *does not let the PC change* (*disable write*!) – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - *changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0*, so effectively the instruction just behind the load becomes a `nop` – a *bubble* is said to have been inserted into the pipeline
    - note that we cannot turn that instruction into an `nop` by 0ing all the bits in the instruction itself – recall `nop` = 00…0 (32 bits) – because it has already been decoded and control signals generated

Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

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

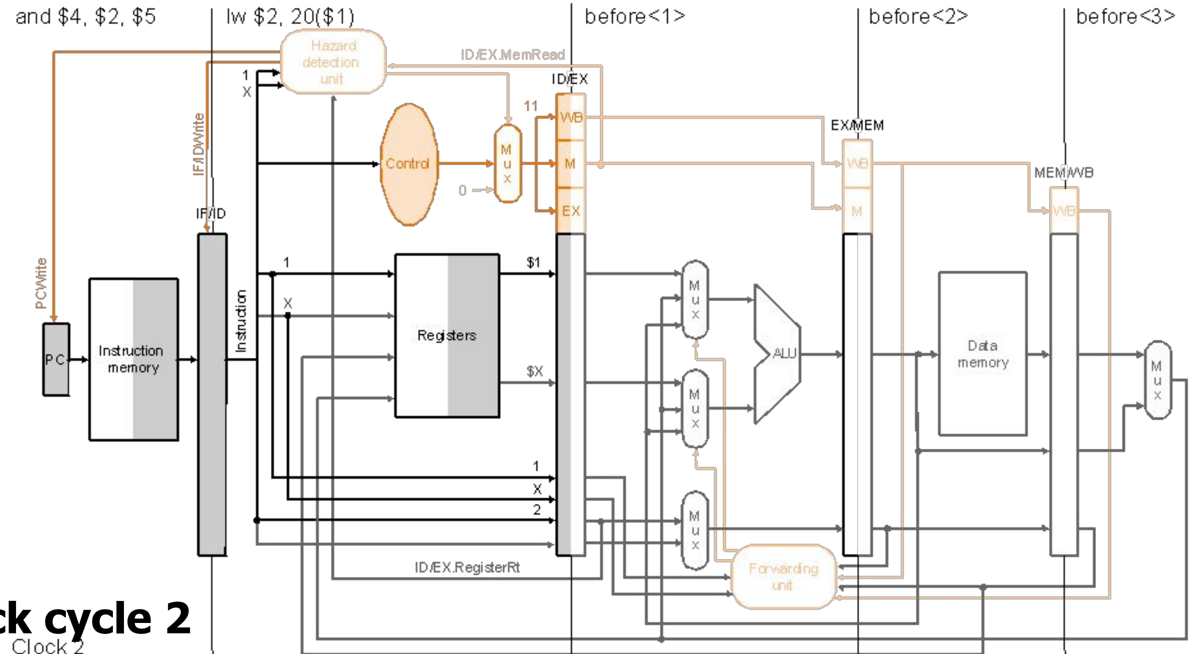


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

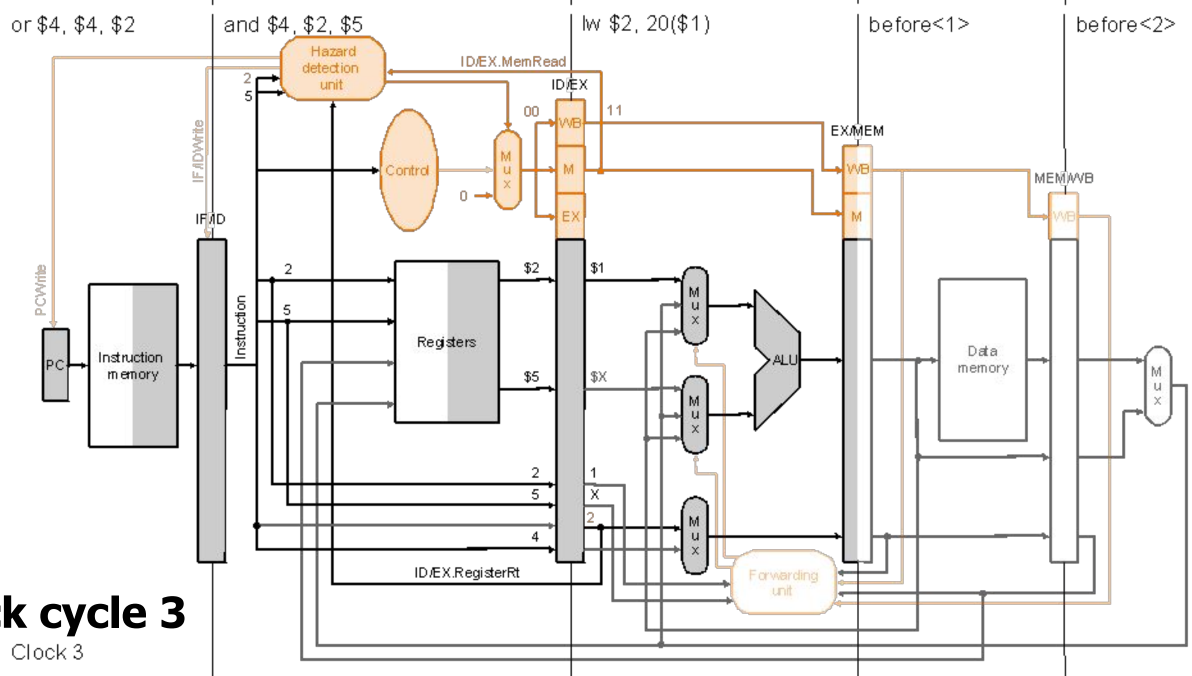
Stalling

- Execution example:

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



Clock cycle 2

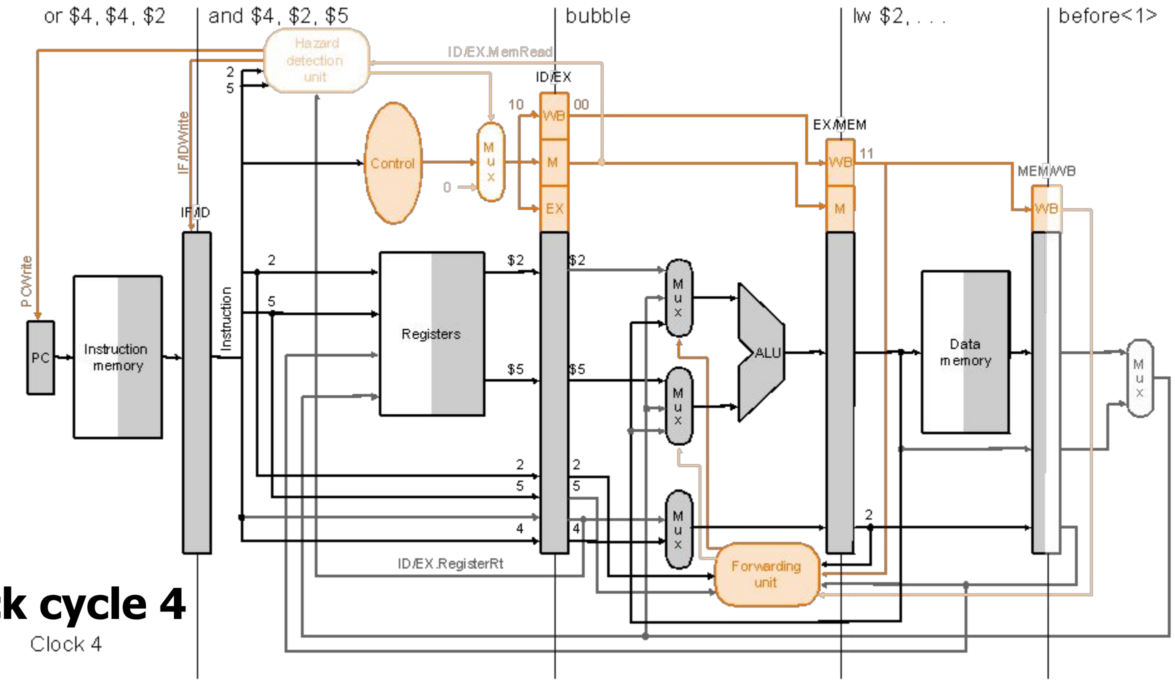


Clock cycle 3

Stalling

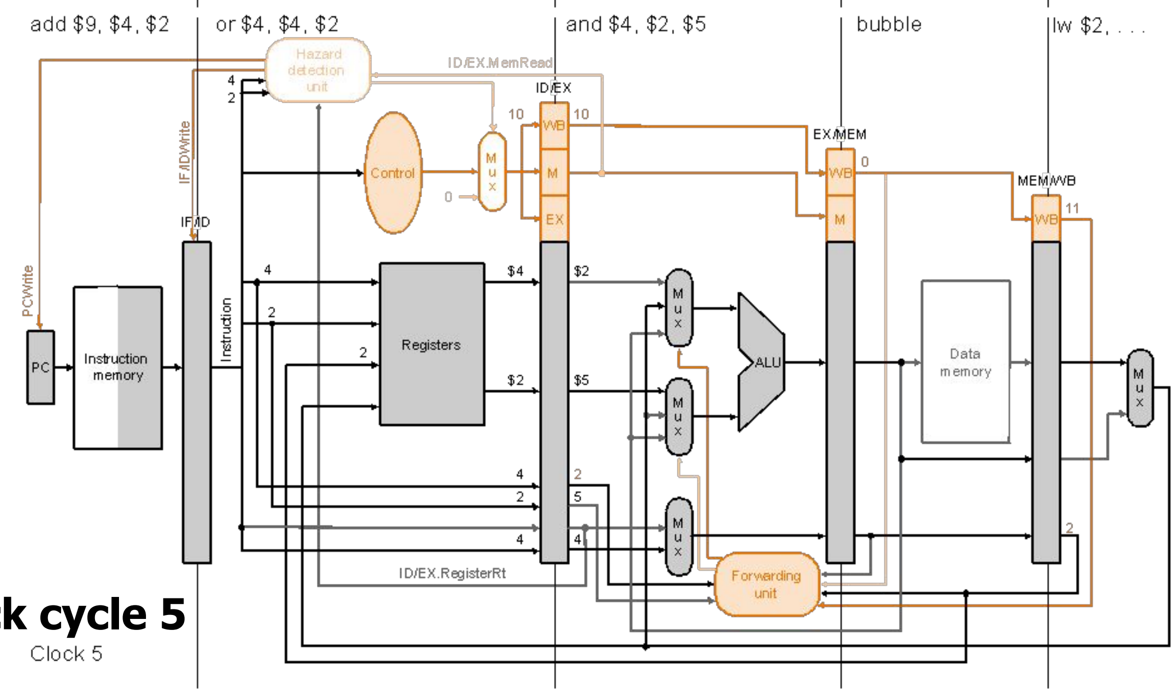
- Execution example (cont.):

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



Clock cycle 4

Clock 4



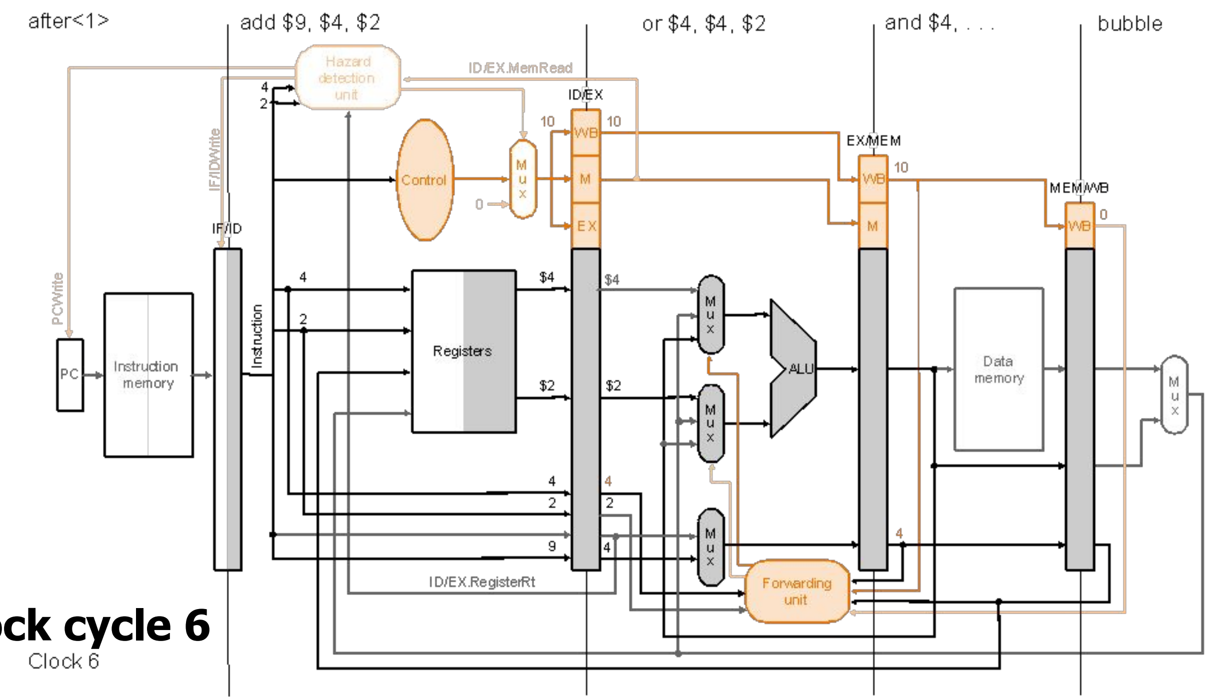
Clock cycle 5

Clock 5

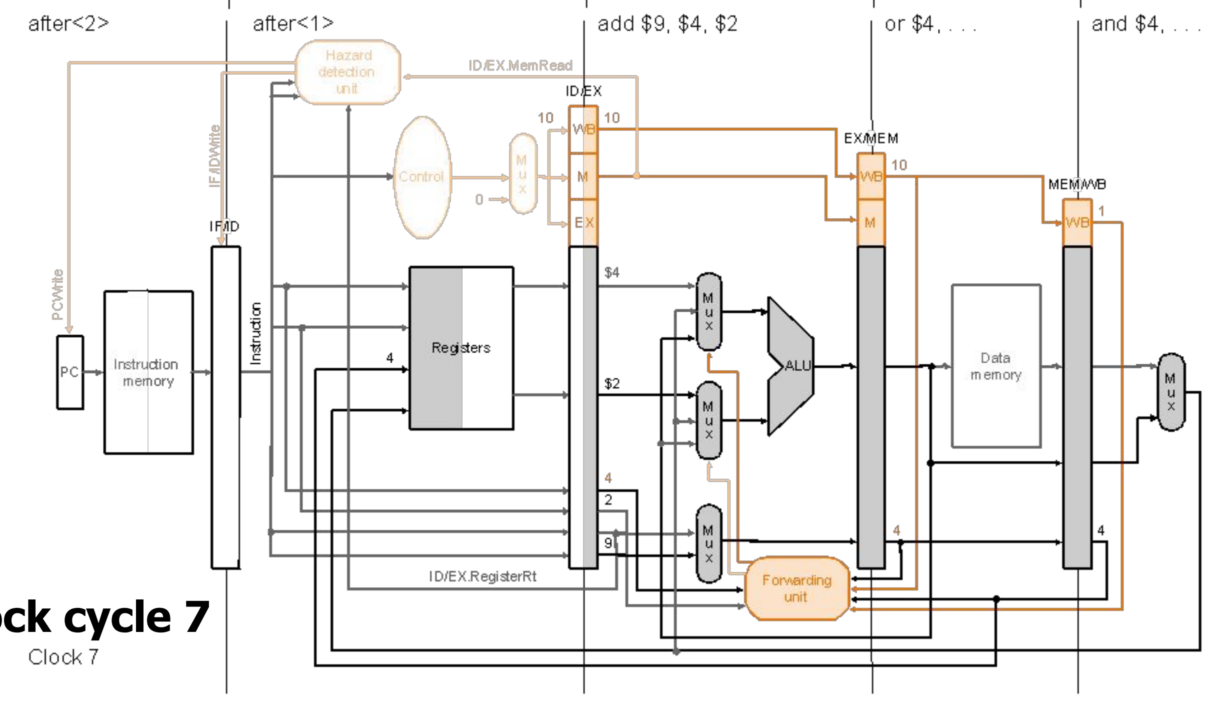
Stalling

- Execution example (cont.):

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



Clock cycle 6



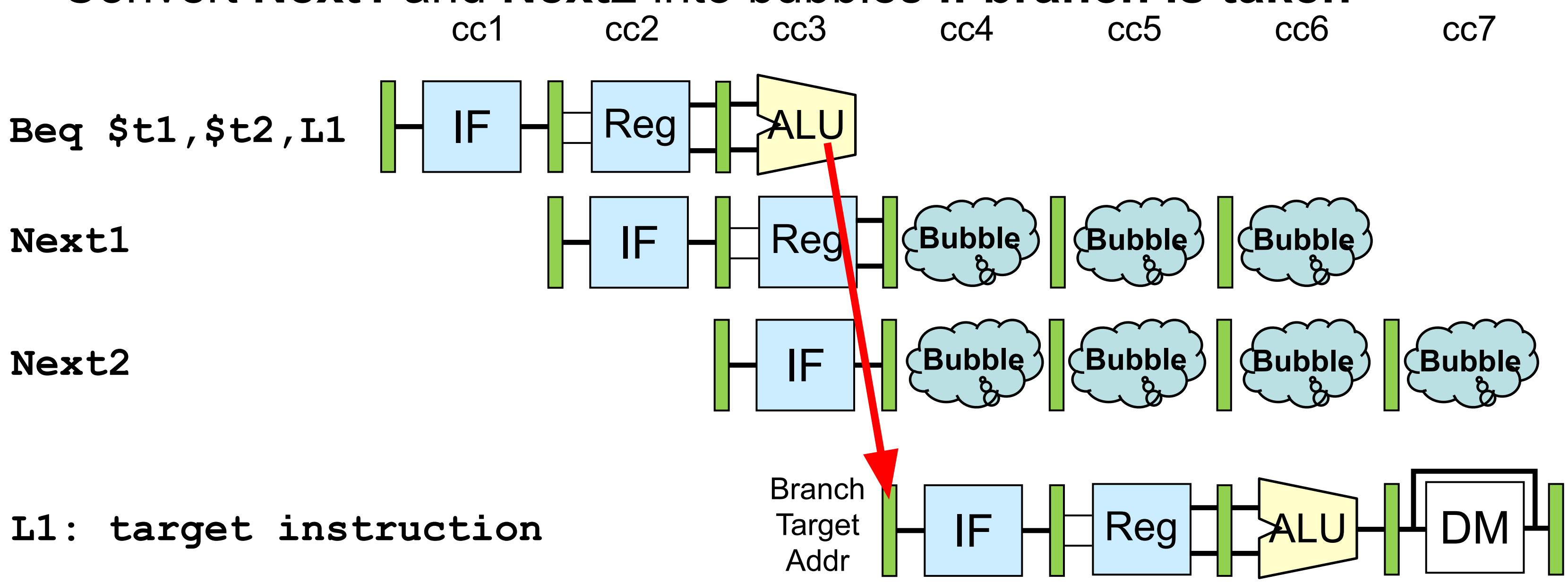
Clock cycle 7

Control Hazards

- Need to make a decision based on the result of a previous instruction still executing in pipeline
- Jump and Branch can cause great performance loss
- Jump instruction needs only the **jump target address**
- Branch instruction needs two things:
  - **Branch Result   Taken or Not Taken**
  - **Branch Target Address**
    - **PC + 4 If Branch is NOT taken**
    - **PC + 4 + 4 × immediate   If Branch is Taken**

Control Hazards

- <u>Solution 1</u> *Stall* the pipeline
- Control logic detects a **Branch** instruction in the 2nd Stage
- ALU computes the **Branch outcome** in the 3rd Stage
- **Next1** and **Next2** instructions will be fetched anyway
- Convert **Next1** and **Next2** into bubbles **if branch is taken**

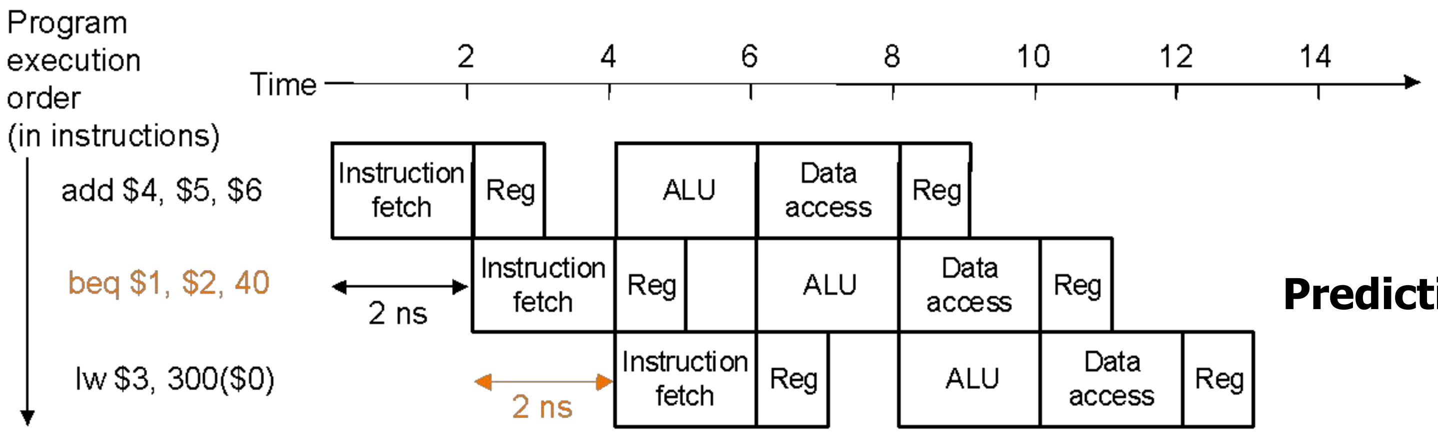


- Branch outcome is computed in ID stage with added hardware (later…)

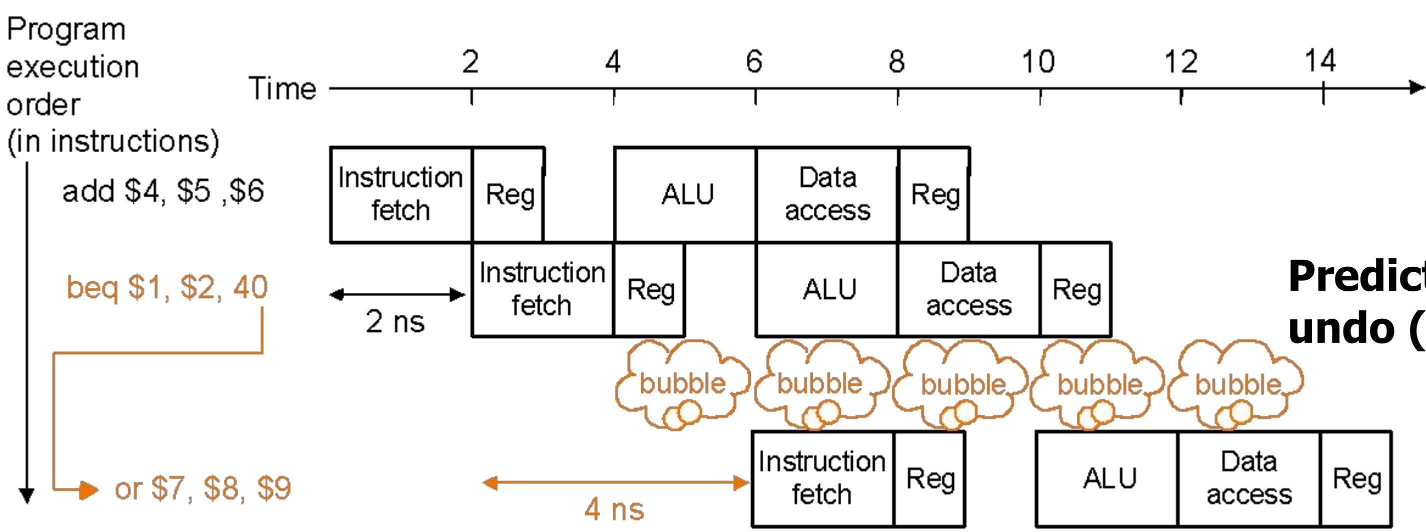
Control Hazards

<u>Solution 2</u> *Predict* branch outcome, e.g., **predict *branch-not-taken*** :

No waste of cycles, if success



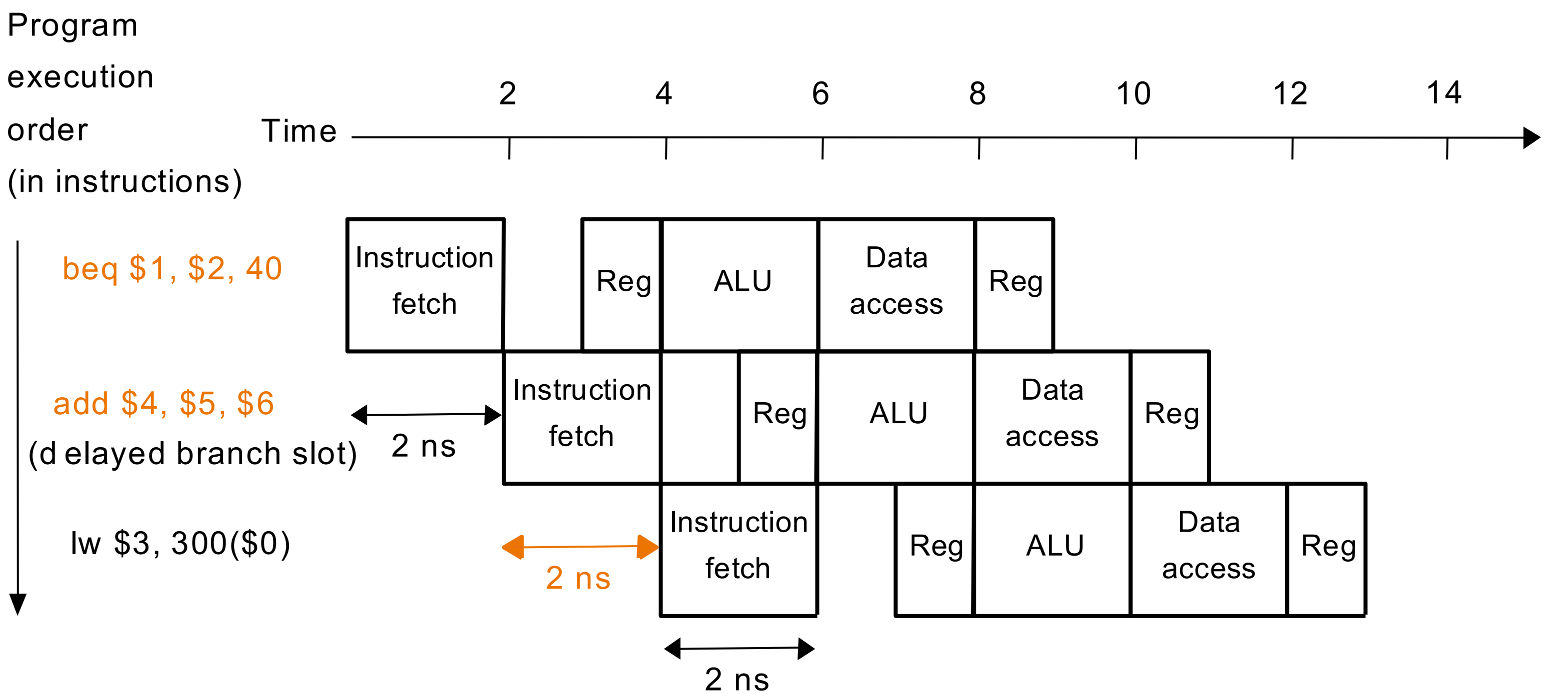
Prediction success



Prediction failure: undo (=flush) `lw`

Control Hazards

<u>Solution 3</u> *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome



Delayed branch `beq` is followed by `add` that is independent of branch outcome

Control (or Branch) Hazards

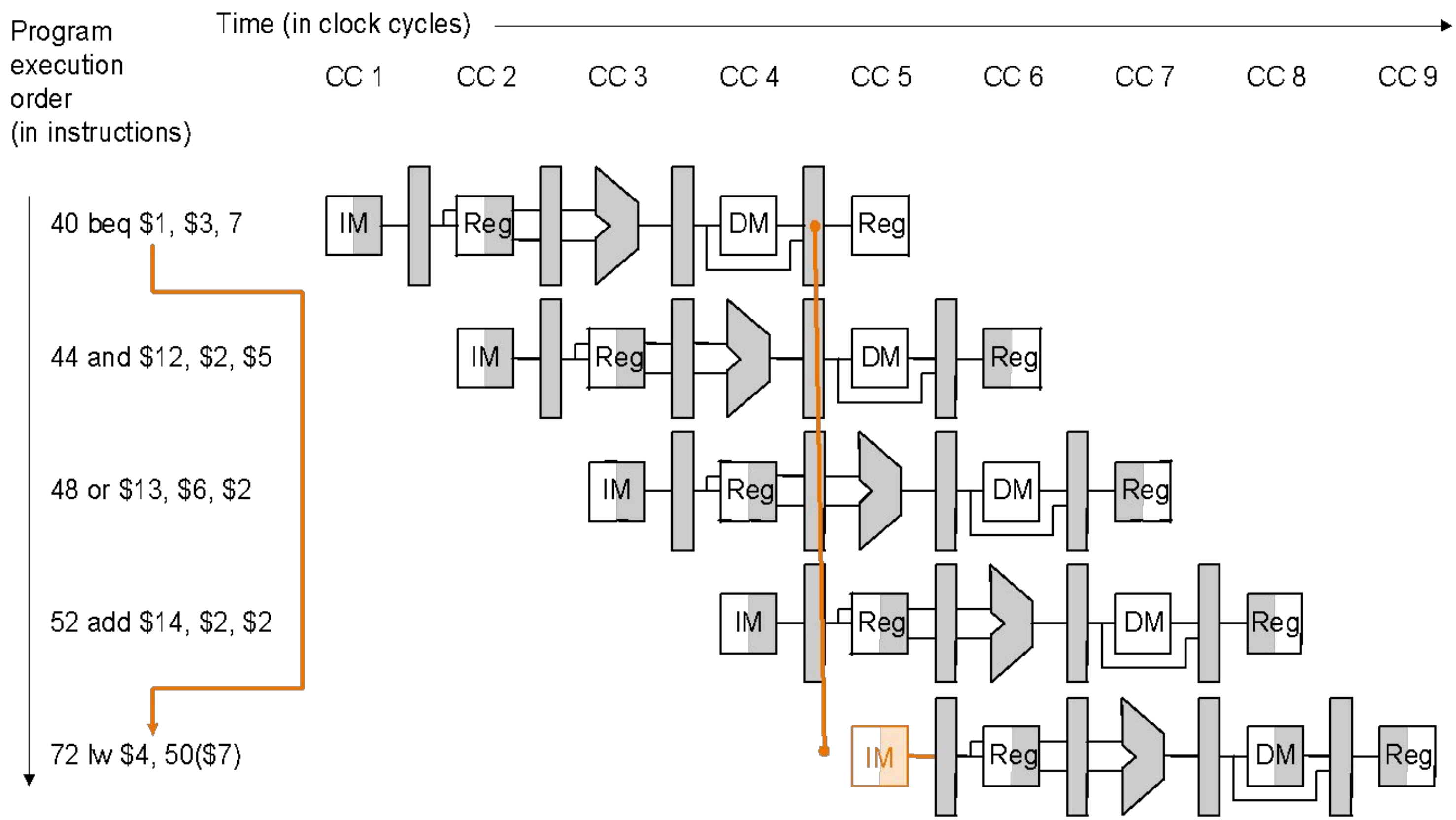
- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

- ***Is there any other OPTIMAL solution?***

Predicting Branch-not-taken: Misprediction delay



The outcome of branch taken (prediction wrong) is decided only when `beq` is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at `lw`

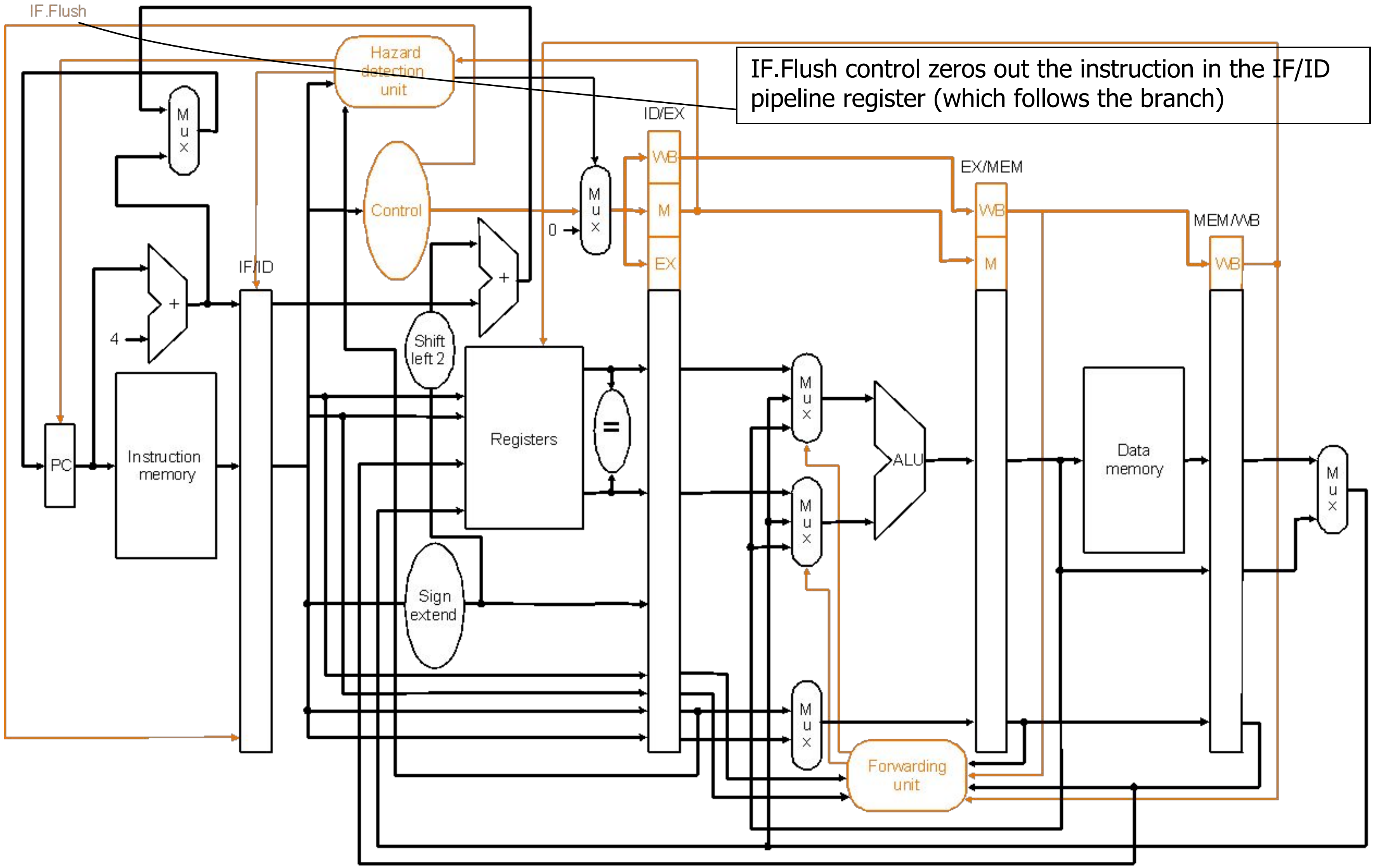
Optimizing the Pipeline to Reduce Branch Delay

- *Move the branch decision* from the MEM stage (as in our current pipeline) *earlier to the ID stage*
  - *calculating the branch target address* involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register
  - *calculating the branch decision* is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)
    - with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember an objective of pipeline design is to keep pipeline stages balanced
  - *we must correspondingly make additions to the forwarding and hazard detection units* to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard…
- Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline – effectively turning them into `nop`s – so they are flushed
  - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimized Datapath for Branch



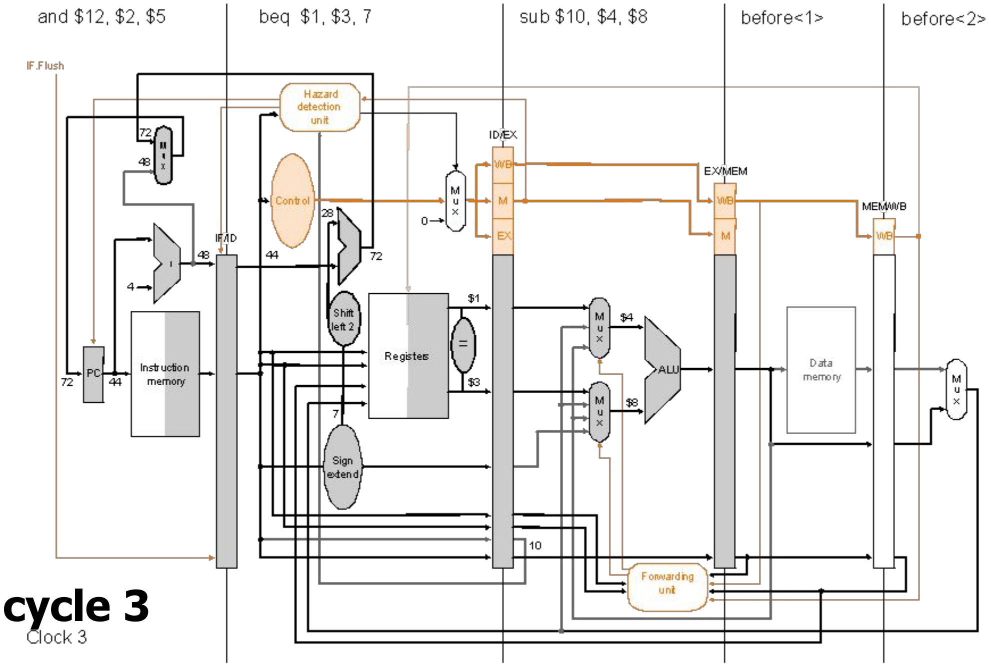
Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch

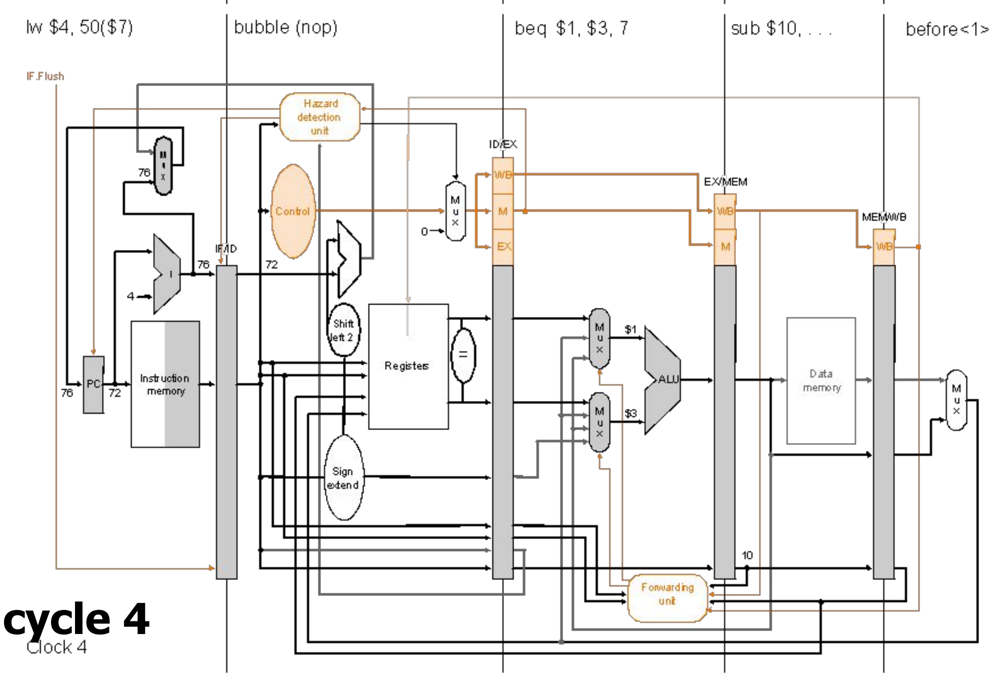
- Execution example:

```
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)
```

Optimized pipeline with only one bubble as a result of the taken branch



Clock cycle 3



Clock cycle 4

Simple Example: Comparing Performance

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
  - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
  - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
  - for pipelined execution assume
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Simple Example: Comparing Performance

- *Single-cycle* (p. 373): average instruction time 8 ns
- *Multicycle* (p. 397): average instruction time 8.04 ns
- *Pipelined*:
  - loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
  - stores use 1 cc each
  - branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
  - jumps use 2 cc each
  - ALU instructions use 1 cc each
  - therefore, average CPI is

  $1.5 \times 23\% + 1 \times 13\% + 1.25 \times 19\% + 2 \times 2\% + 1 \times 43\% = 1.18$
  - therefore, average instruction time is $1.18 \times 2 = 2.36$ ns
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Pipelining Advantages

- Higher maximum throughput
- Higher utilization of CPU resources

- But, more hardware needed, perhaps complex control

Pipelining Exercise

Consider the following MIPS assembly code:

```
add $3, $2, $3
lw $4, 100($3)
sub $7, $6, $2
xor $6, $4, $3
```

Assume there is no forwarding or stalling circuitry in a pipelined processor that uses the standard 5-stages (IF, ID, EX, Mem, WB). Instead, we will require the compiler to add no-ops to the code to ensure correct execution. (Assume that if the processor reads and writes to the same register in a given cycle, the value read out will be the new value that is written in.)

1. Rewrite the code to include the no-ops that are needed. Do not change the order of the four statements. Use as few no-ops as possible.

2. Suppose the complier is allowed to change the order of the four statements, provided it doesn't change the final answer. Is it possible to reduce the number of no-ops needed? Why or why not?

Tutorial Question

Draw an execution diagram that shows where forwarding and stalling would take place, if any.

add \$6,\$5,\$2

lw \$7,0(\$6)

addi \$7,\$7,10

add \$6,\$4,\$2

sw \$7,0(\$6)

addi \$2,\$2,4

blt \$2,\$3,loop

add \$6,\$5,\$2

Refer

Patterson Chapter 6: Topics 6.1 to 6.6

End…