ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

COMPUTER ORGANISATION
(TỔ CHỨC MÁY TÍNH)

Pipelining

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Road Map: **Part II**

- Performance
- Assembly Language
- Processor: Datapath
- Processor: Control
- **Pipelining**
- Cache

- **Pipelining**
  - MIPS Pipeline
  - Pipeline datapath & control
  - Pipeline hazards
  - Branch prediction

Pipelining: **Overview**

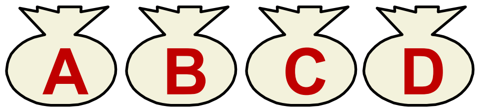
- MIPS Pipeline Stages
- Pipelined Datapath and Control
- Pipeline Hazards
  - Structural
  - Data
  - Control
- Forwarding
- Branch prediction

Inspiration: **Assembly Line**

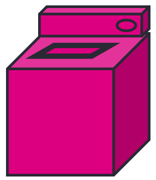
Simpler station tasks → more cars per hour.
Simple tasks take less time, clock is faster.

The Laundry Example

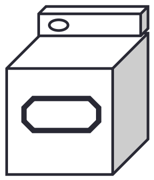
- **A**nn, **B**rian, **C**athy, **D**ave each have one load of clothes to wash, dry, fold and stash



- **Washer** takes 30 minutes

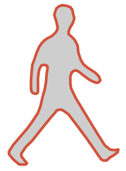


- **Dryer** takes 30 minutes

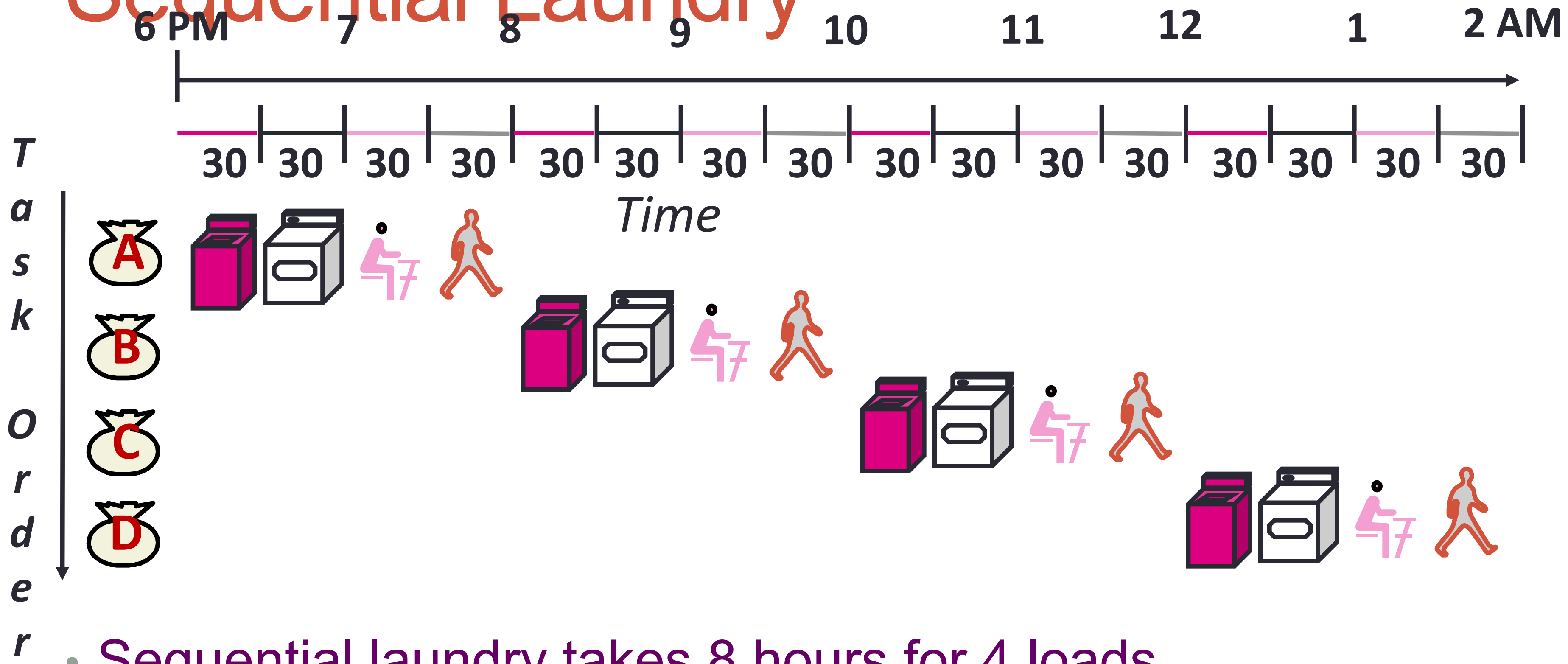


- "**Folder**" takes 30 minutes

- "**Stasher**" takes 30 minutes to put clothes into drawers

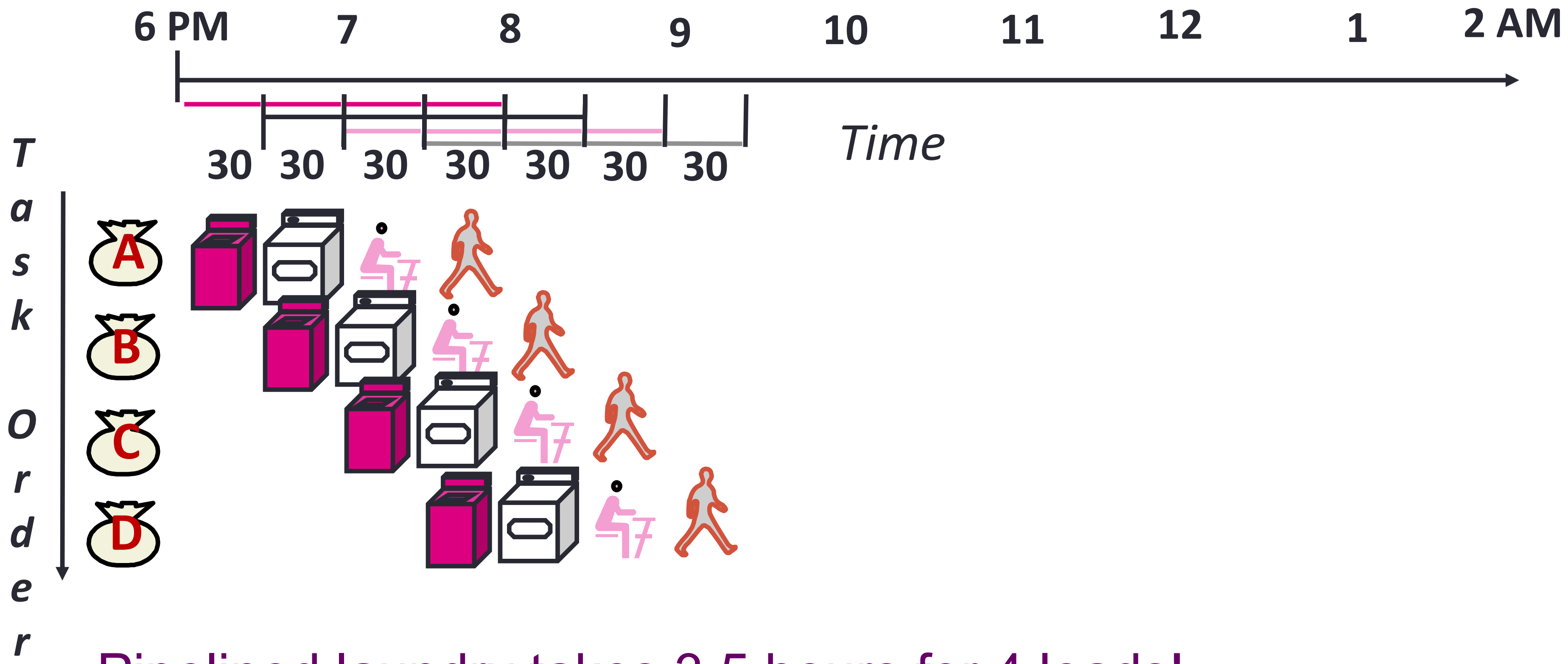


Sequential Laundry



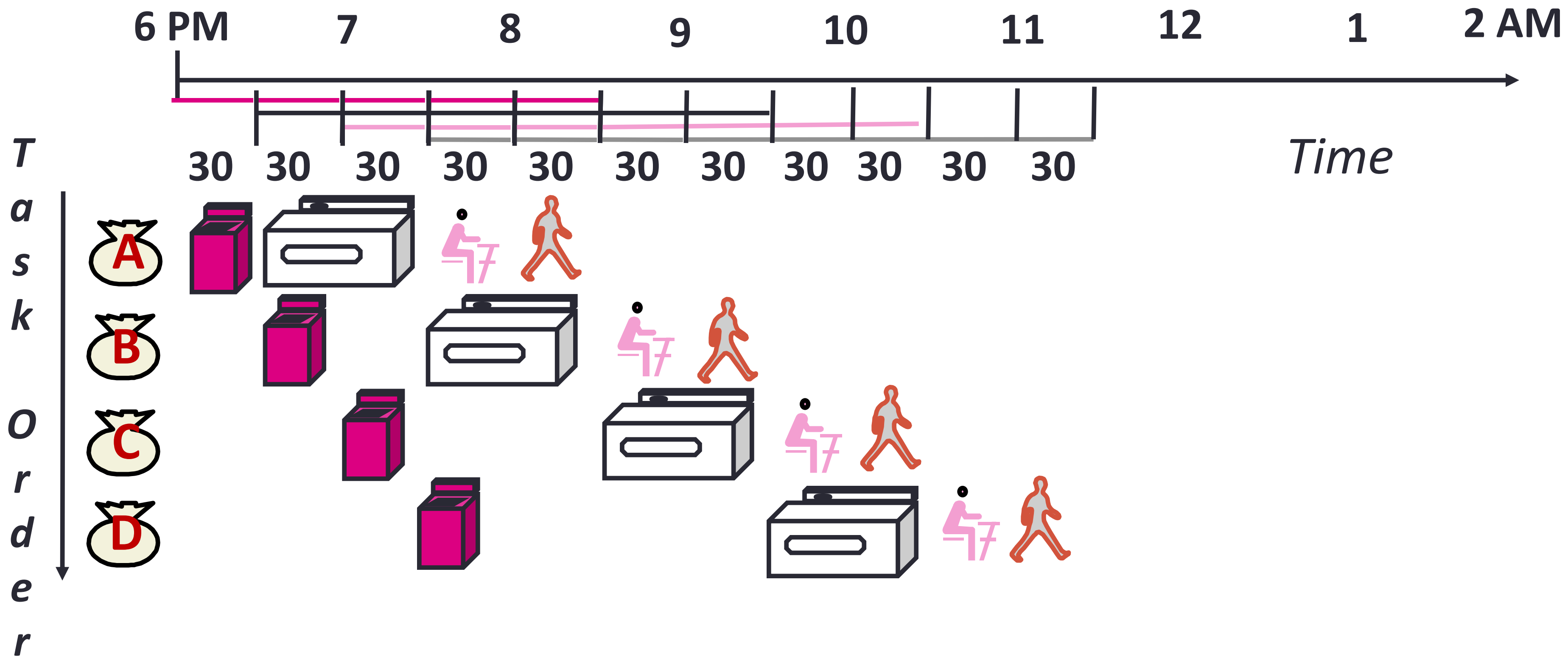
- Sequential laundry takes 8 hours for 4 loads
- Steady state: 1 load every 2 hours
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



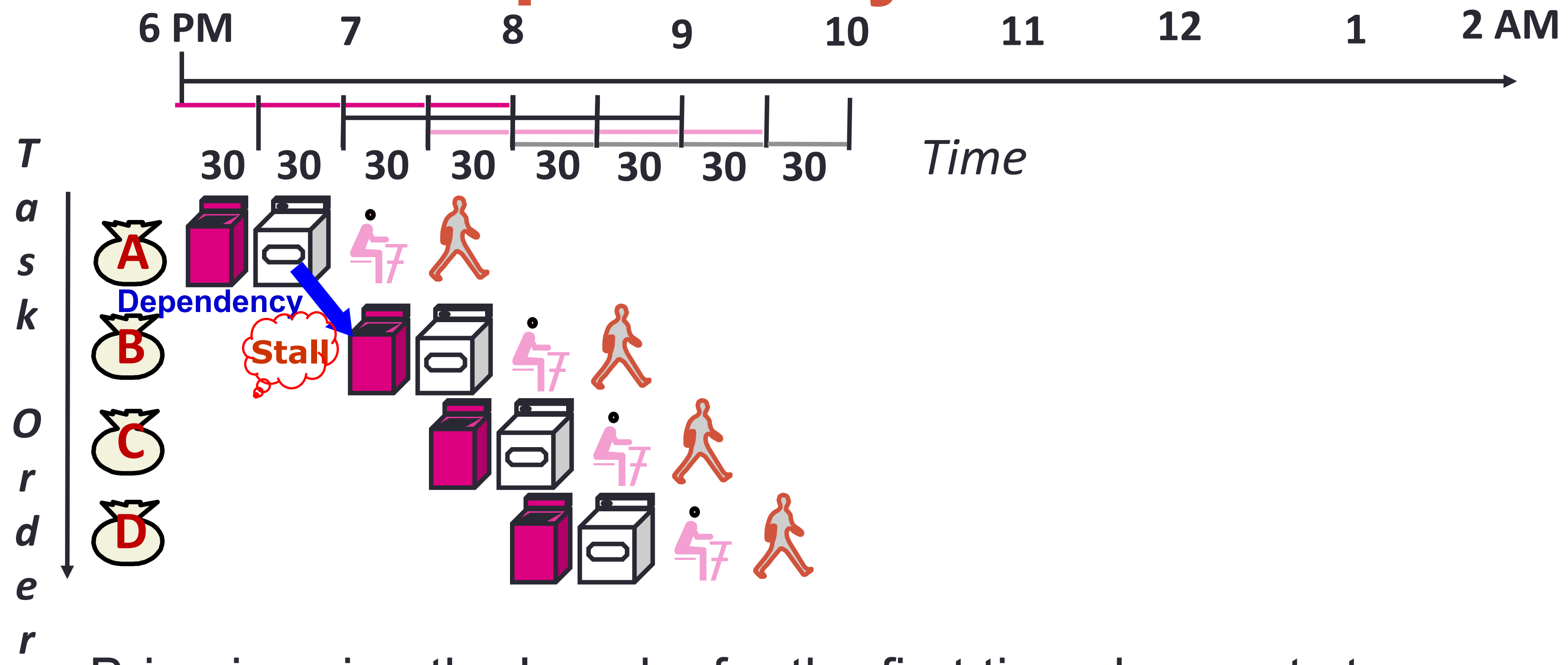
- Pipelined laundry takes 3.5 hours for 4 loads!
- Steady state: 1 load every 30 min
- Potential speedup = 2 hr/30 min = 4 (# stages)
- Time to fill pipeline takes 2 hours → speedup ↓

What IF: **Slow Dryer**



- **Pipelined laundry now takes 5.5 hours!**
- Steady state: One load every 1 hr (dryer speed)
- **Pipeline rate is limited by the slowest stage**

What IF: **Dependency**



- Brian is using the laundry for the first time; he wants to see the outcome of one wash + dry cycle first before putting in his clothes
- Pipelined laundry now takes 4 hours

Pipelining Lessons

- Pipelining doesn't help **latency** of single task:
  - It helps the **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Possible delays:
  - Pipeline rate limited by **slowest** pipeline stage
  - Stall for **dependencies** (more on this later)

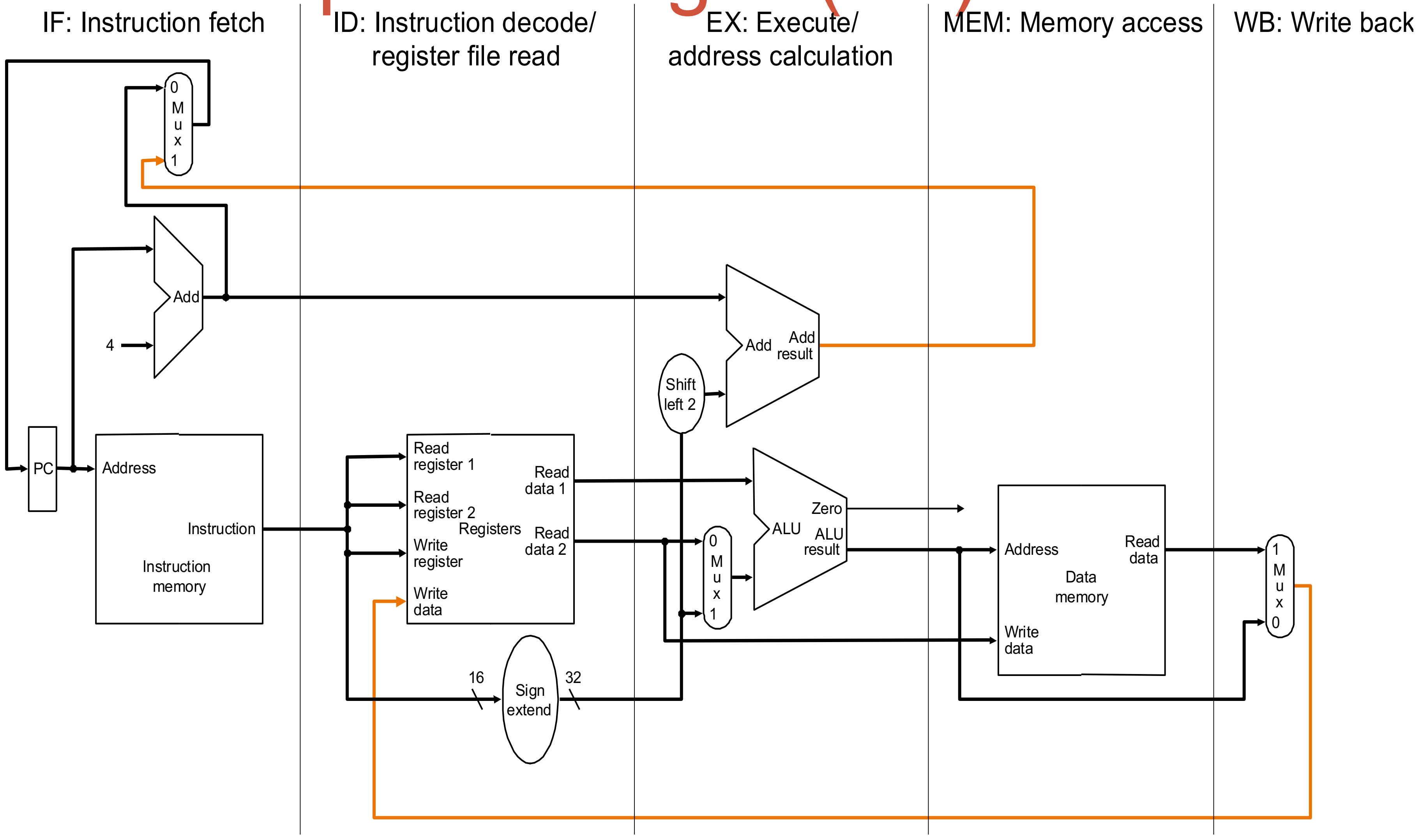
MIPS PIPELINE

Datapath and Control

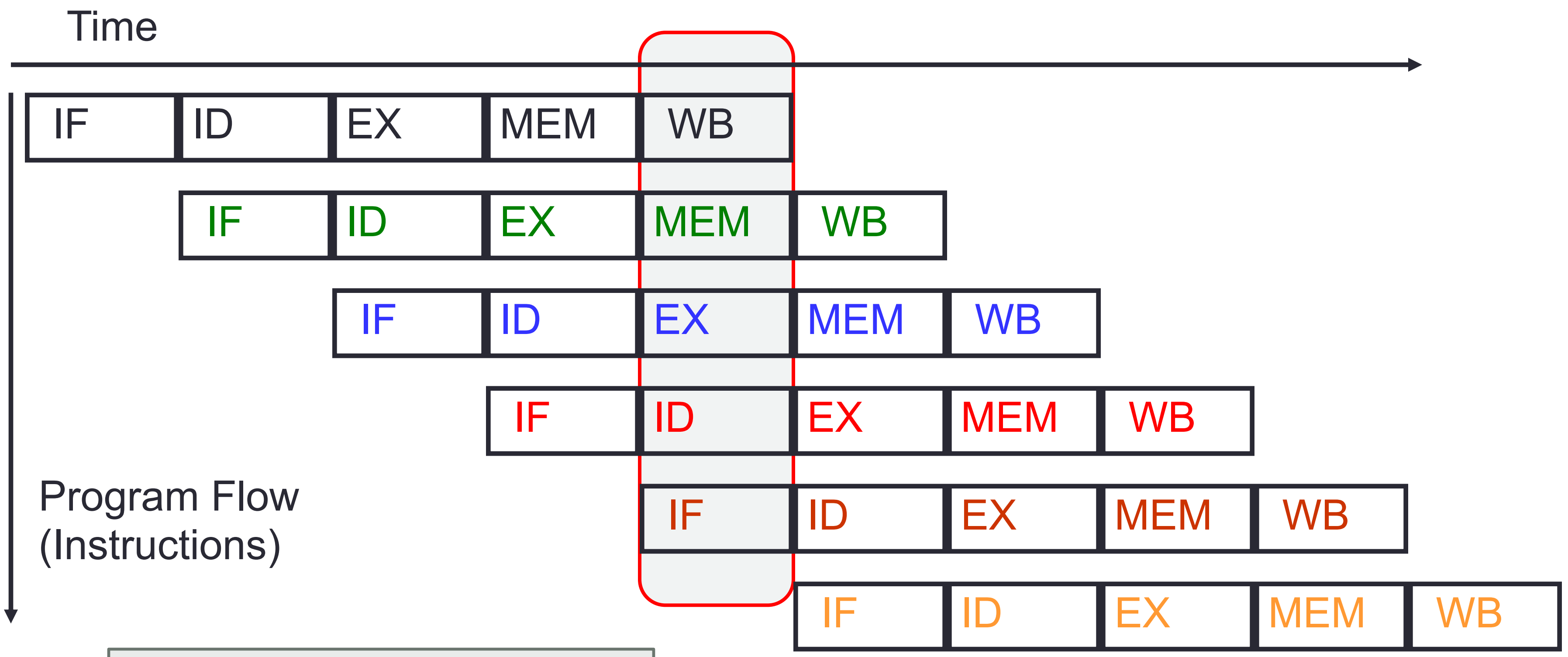
MIPS Pipeline Stages (1/2)

- **Five** Execution Stages
  - `IF`: Instruction Fetch
  - `ID`: Instruction Decode and Register Read
  - `EX`: Execute an operation or calculate an address
  - `MEM`: Access an operand in data memory
  - `WB`: Write back the result into a register
- **Idea:**
  - **Each execution stage takes 1 clock cycle**
  - General flow of data is from one stage to the next
- **Exceptions:**
  - Update of PC and write back of register file – more about this later…

MIPS Pipeline Stages (2/2)



Pipelined Execution: **Illustration**



Several instructions are in the pipeline simultaneously!

MIPS Pipeline: **Datapath**(1/3)

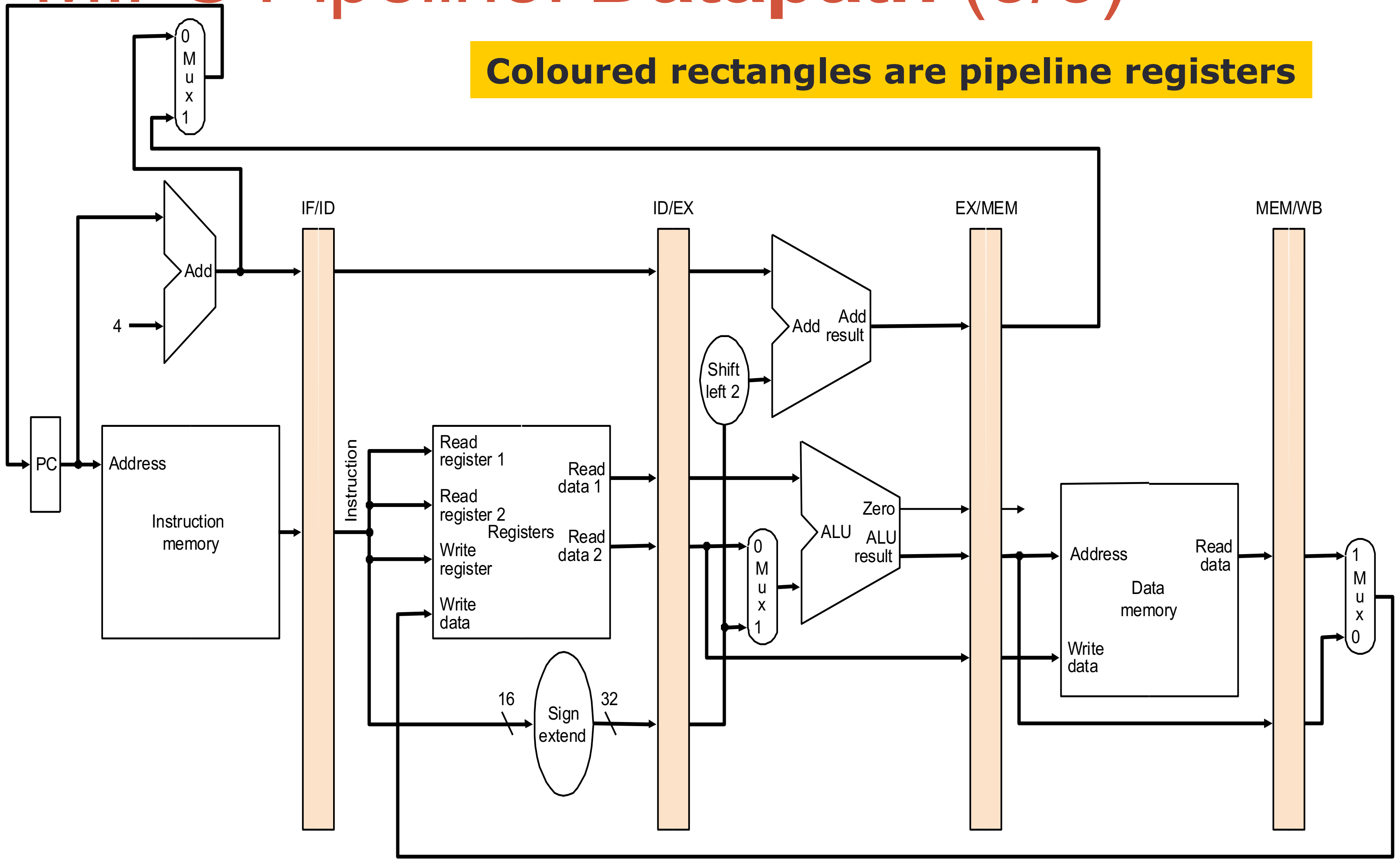
- Single-cycle implementation:
  - Update all state elements (`PC`, register file, data memory) at the end of a clock cycle
- Pipelined implementation:
  - One cycle per pipeline stage
  - Data required for each stage needs to be stored separately (why?)

MIPS Pipeline: **Datapath**(2/3)

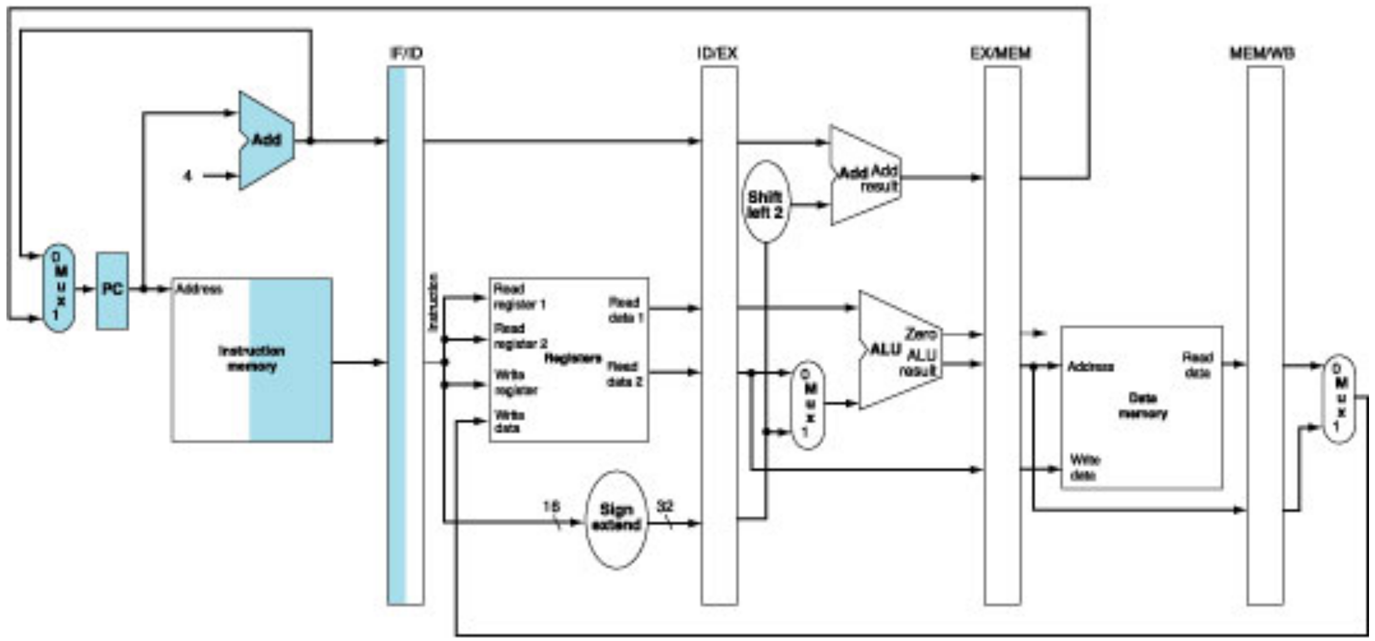
- Data used by **subsequent instructions:**
  - Store in programmer-visible state elements: **`PC`**, register file and memory
- Data used by **same instruction** in later pipeline stages:
  - Additional registers in datapath called **pipeline registers**
  - **`IF/ID`**: register between **`IF`** and **`ID`**
  - **`ID/EX`**: register between **`ID`** and **`EX`**
  - **`EX/MEM`**: register between **`EX`** and **`MEM`**
  - **`MEM/WB`**: register between **`MEM`** and **`WB`**
- Why no register at the end of **`WB`** stage?

MIPS Pipeline: **Datapath** (3/3)

Coloured rectangles are pipeline registers

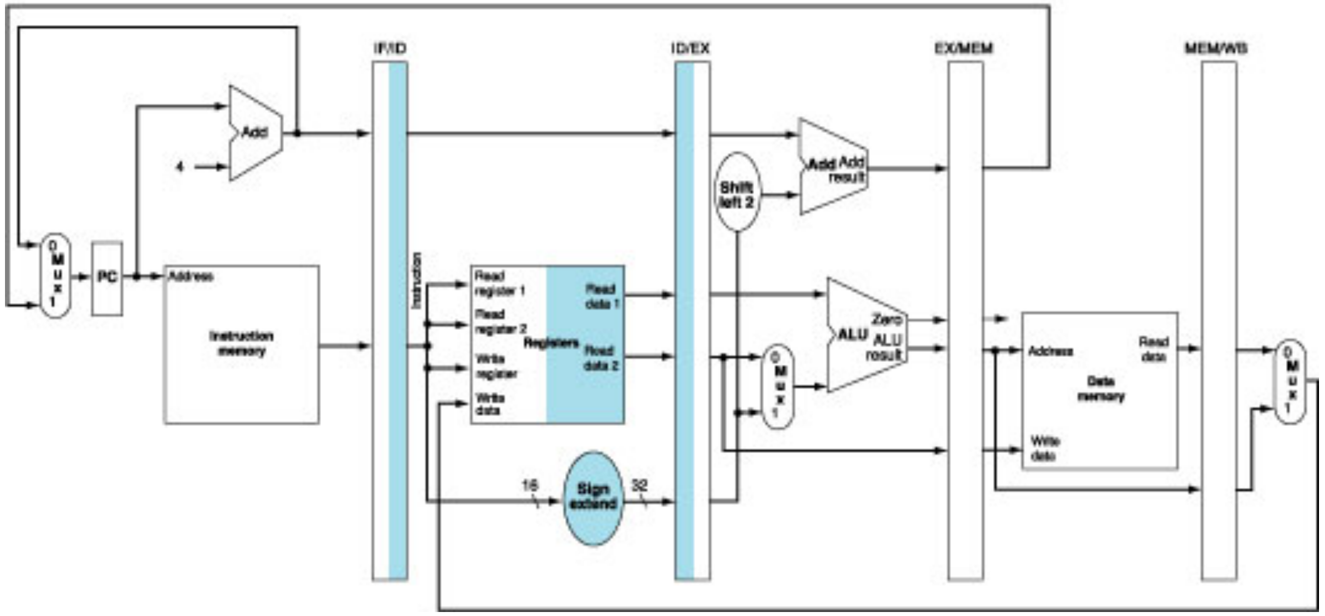


Pipeline Datapath: `IF` Stage



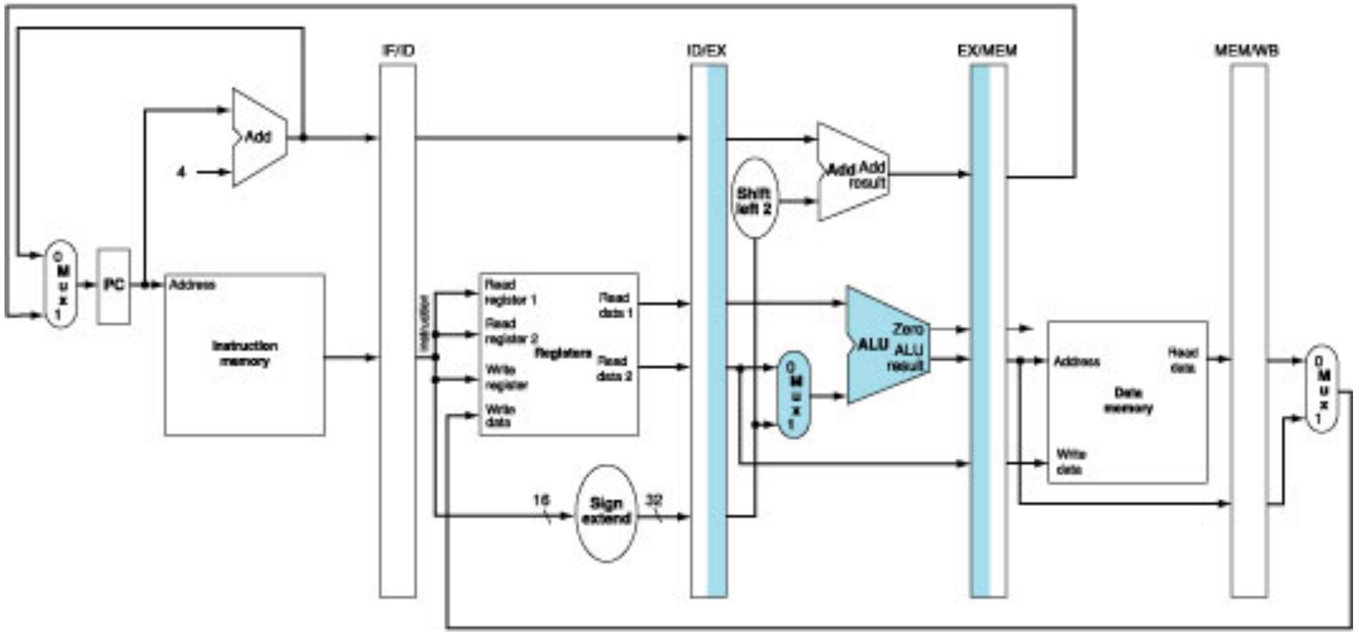
- At the end of a cycle, `IF/ID` receives (stores):
  - Instruction read from DataMemory[PC]
  - PC + 4
- PC + 4
  - Also connected to one of the MUX's inputs (another coming later)

Pipeline Datapath: ID Stage



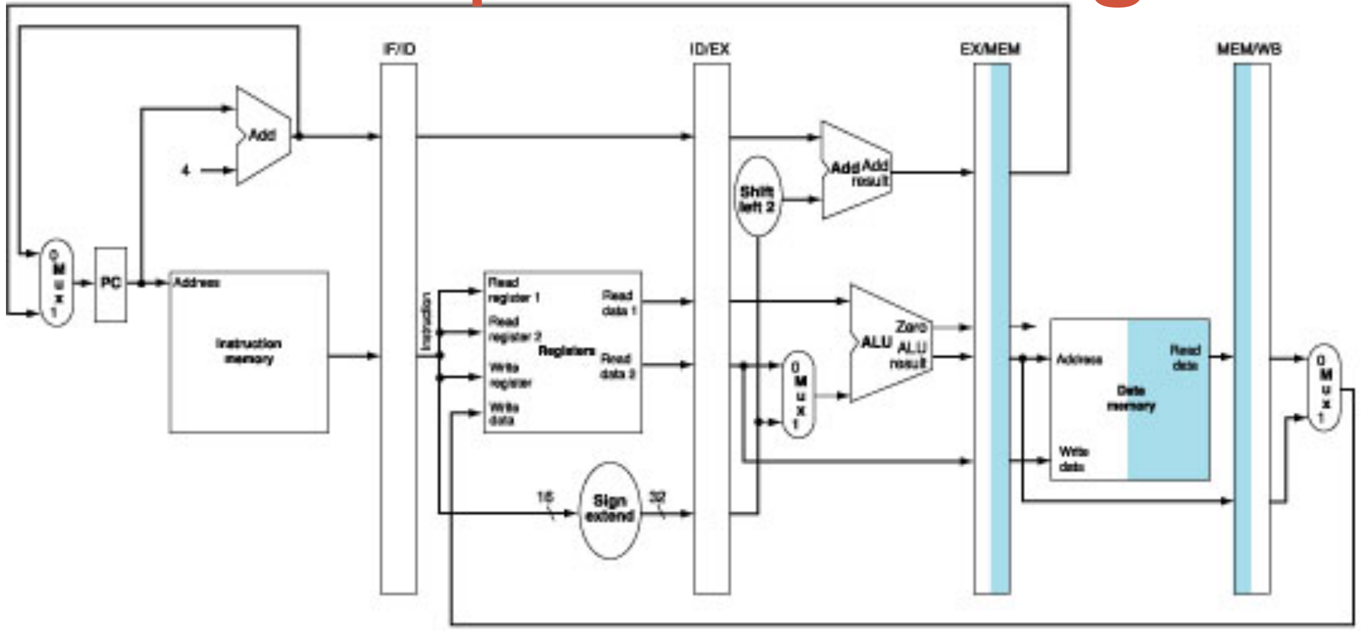
At the beginning of a cycle **IF/ID** register supplies:	At the end of a cycle **ID/EX** receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	❖ Data values read from register file ❖ 32-bit immediate value ❖ `PC + 4`

Pipeline Datapath: EX Stage



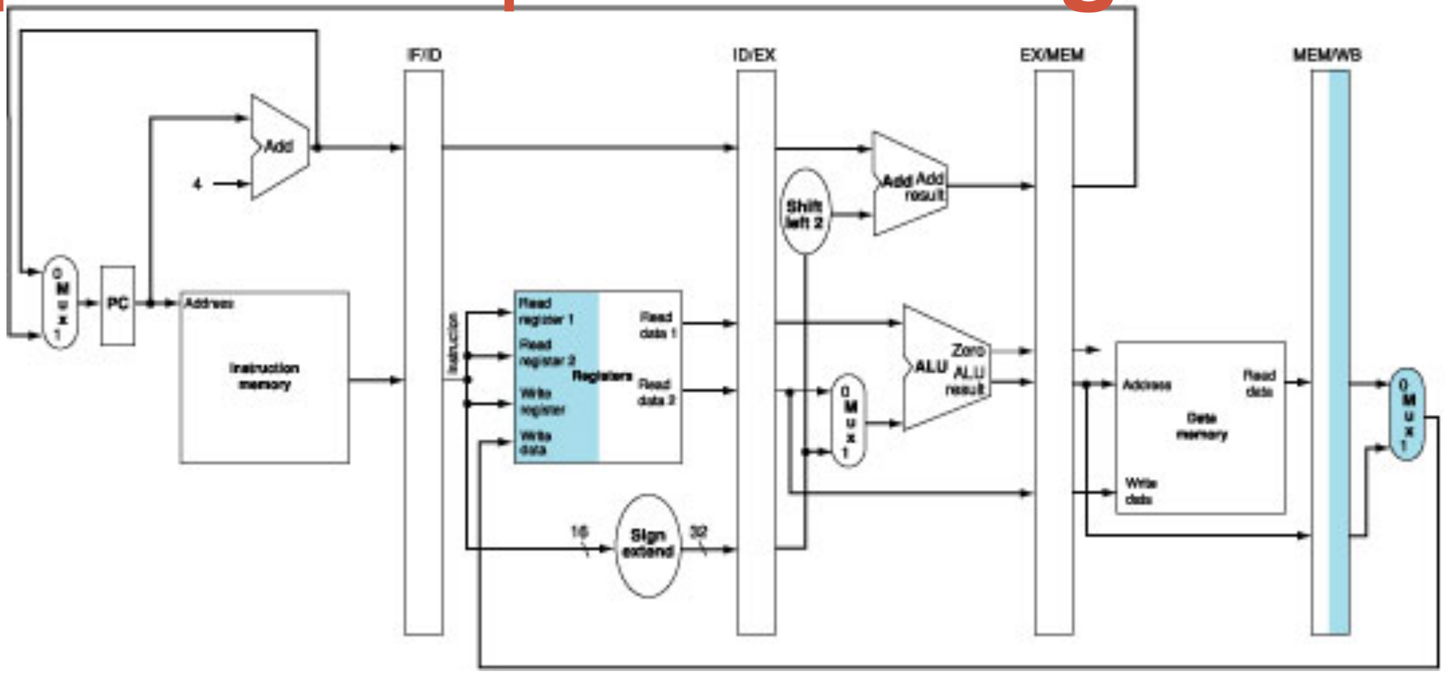
At the beginning of a cycle **ID/EX** register supplies:	At the end of a cycle **EX/MEM** receives:
❖ Data values read from register file ❖ 32-bit immediate value ❖ `PC + 4`	❖ (PC + 4) + (Immediates x 4) ❖ ALU result ❖ `isZero?` signal ❖ Data Read 2 from register file

Pipeline Datapath: `MEM` Stage



At the beginning of a cycle `EX/MEM` register supplies:	At the end of a cycle `MEM/WB` receives:
❖ (PC + 4) + (Immediates x 4) ❖ ALU result ❖ `isZero?` signal ❖ Data Read 2 from register file	❖ ALU result ❖ Memory read data

Pipeline Datapath: WB Stage



At the beginning of a cycle MEM/WB register supplies:	At the end of a cycle
❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable) ❖ **There is a bug here…….**

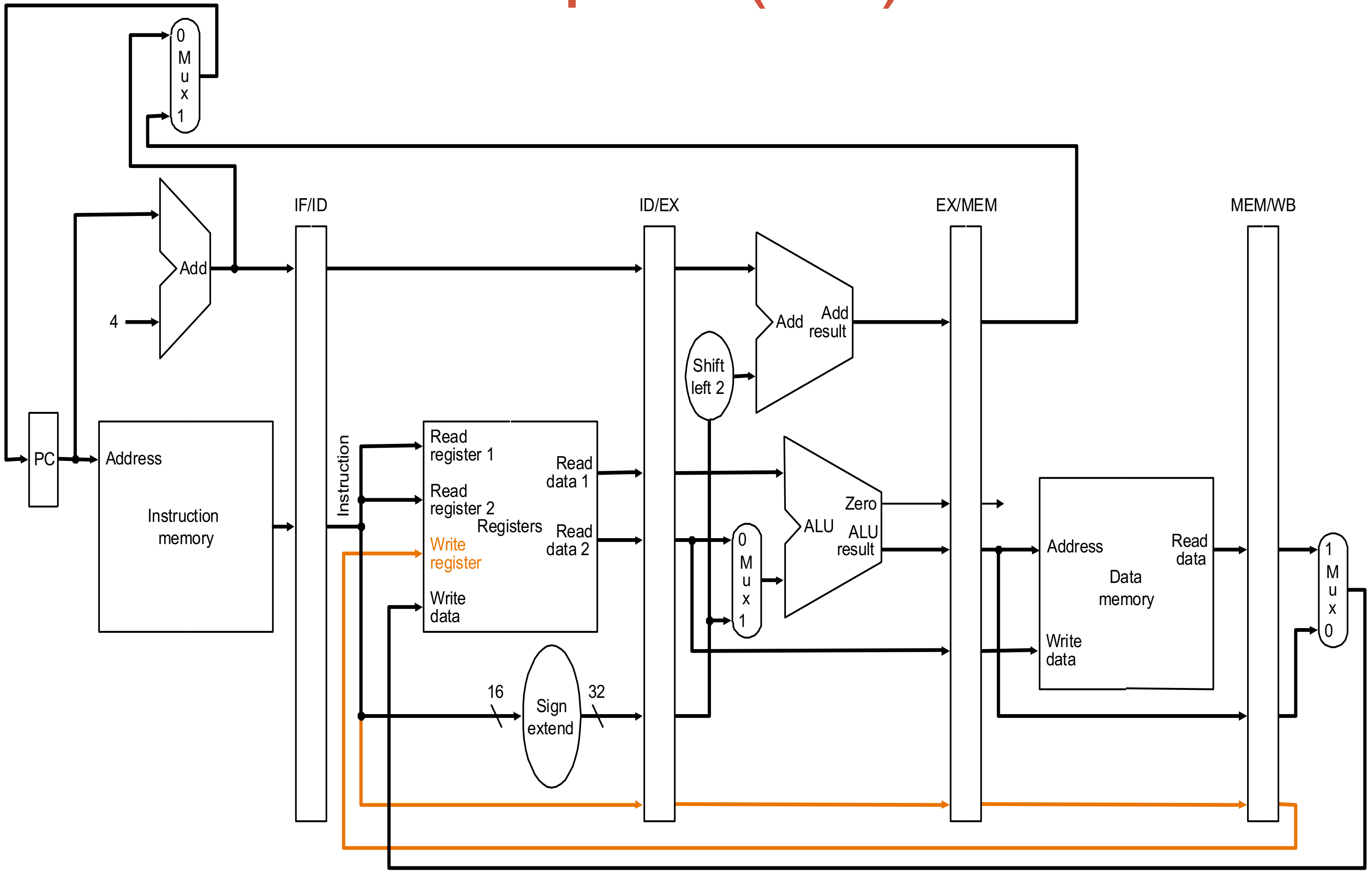
Corrected Datapath (1/2)

- Observe the “Write register” number
  - Supplied by the **`IF/ID`** pipeline register

  ➔ It is NOT the correct write register for the instruction now in **`WB`** stage!

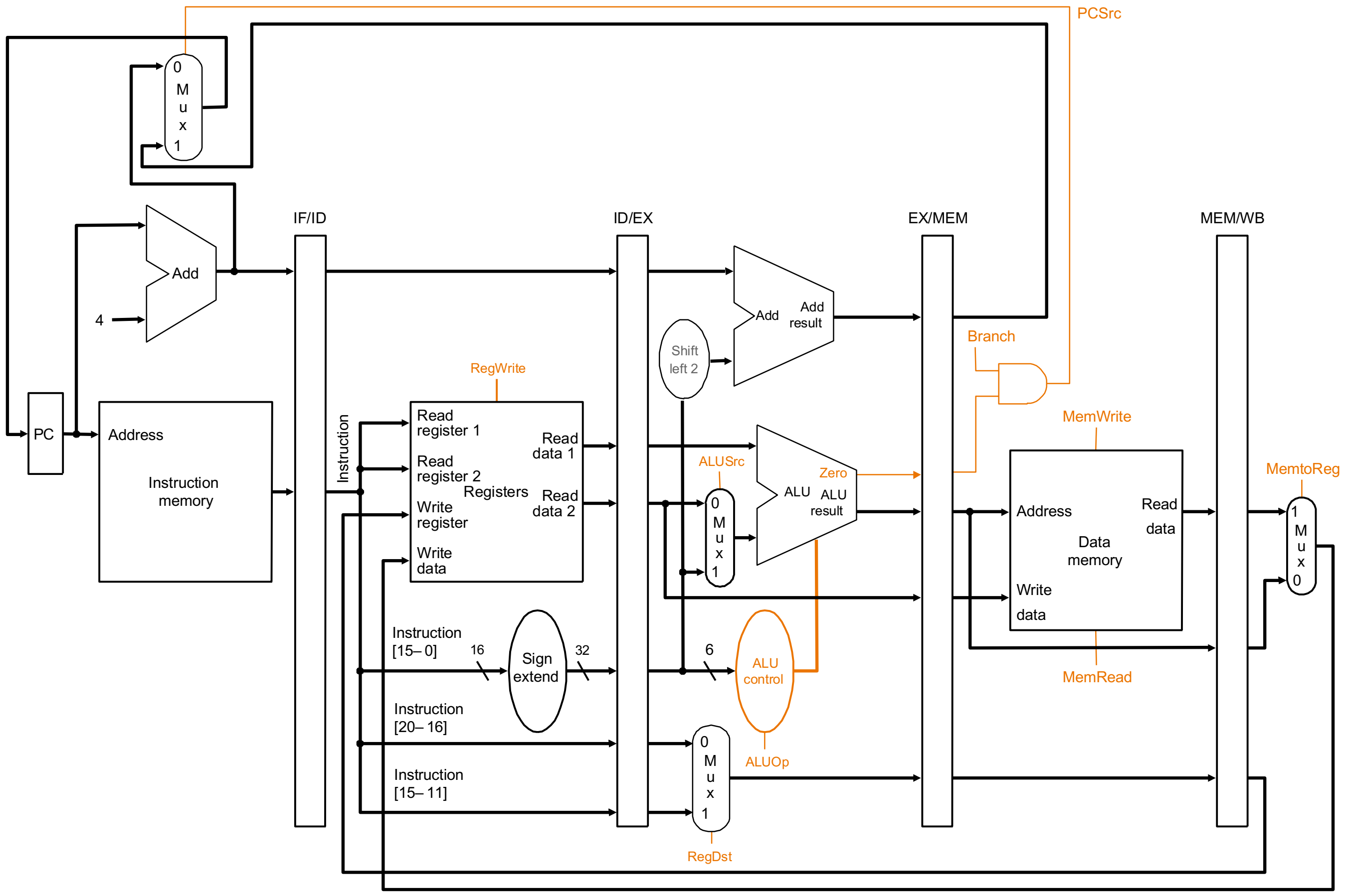
- **Solution:**
  - Pass “Write register” number from **`ID/EX`** through **`EX/MEM`** to **`MEM/WB`** pipeline register for use in **`WB`** stage
  - i.e. let the "Write register" number follows the instruction through the pipeline until it is needed in WB stage

Corrected Datapath (2/2)

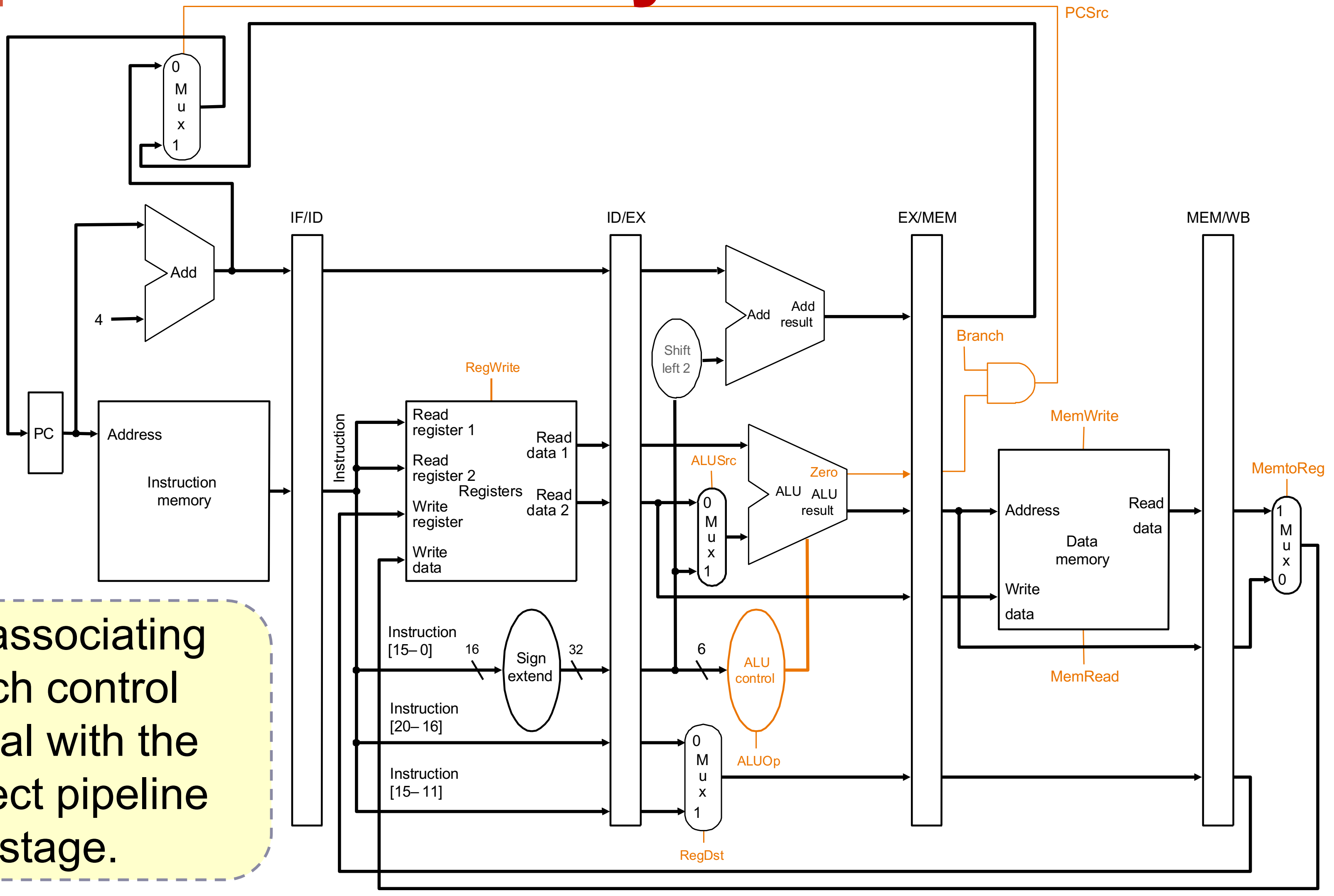


Pipeline Control: **Main Idea**

- Same control signals as single-cycle datapath
- **Difference:** Each control signal belongs to a particular pipeline stage



Pipeline Control: **Try it!**



Try associating each control signal with the correct pipeline stage.

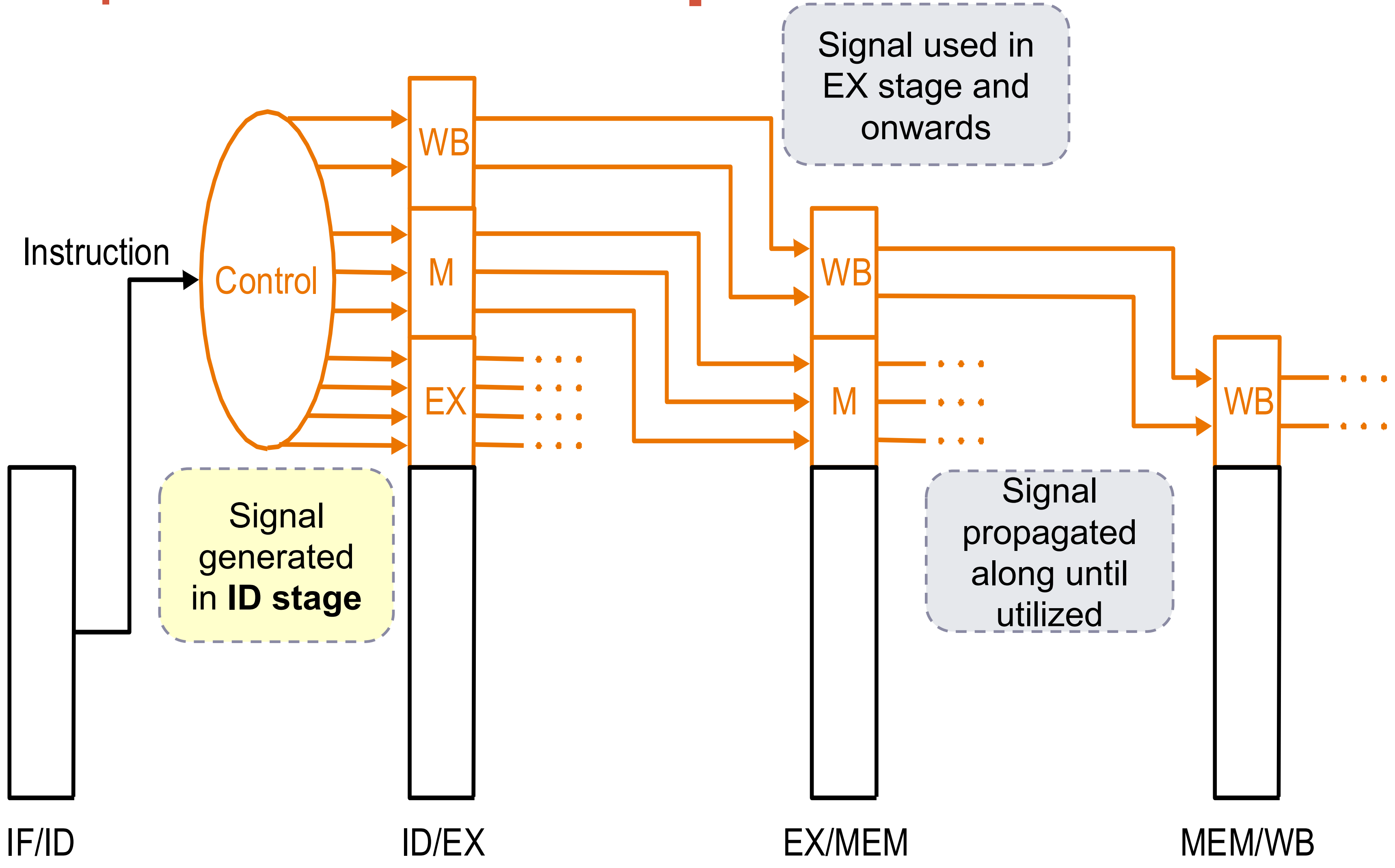
Pipeline Control: **Grouping**

- Group control signals according to pipeline stage

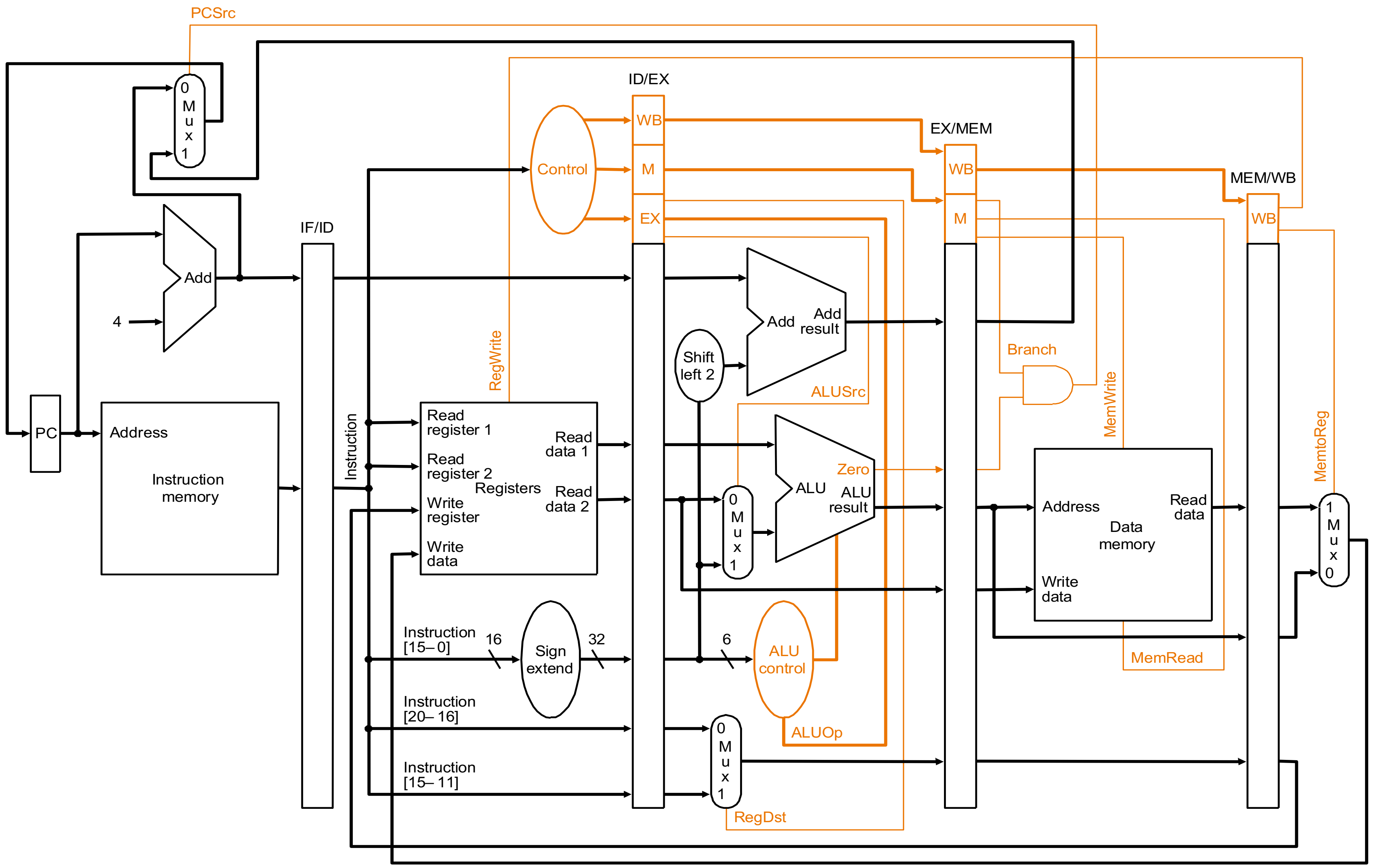
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

Pipeline Control: **Implementation**

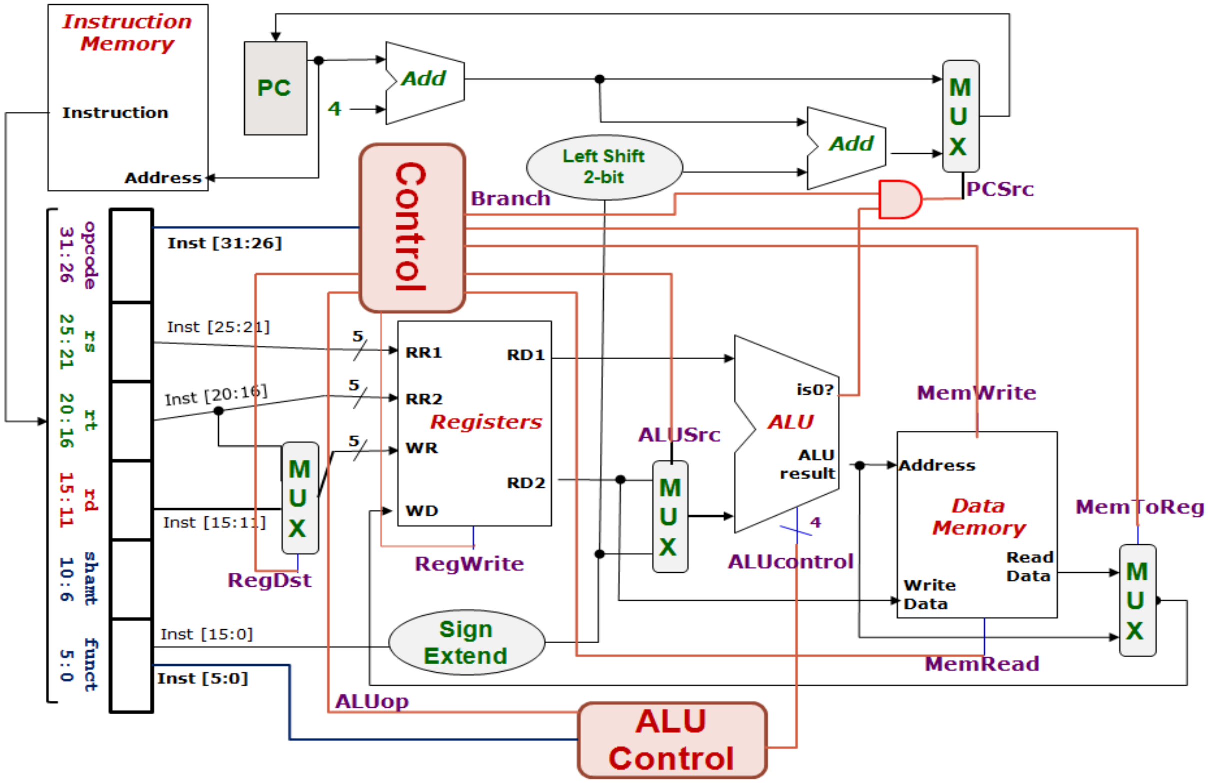

Signal used in EX stage and onwards
Instruction
Control
WB
M
EX
WB
M
WB
Signal generated in **ID stage**
Signal propagated along until utilized
IF/ID
ID/EX
EX/MEM
MEM/WB

MIPS Pipeline: **Datapath and Control**

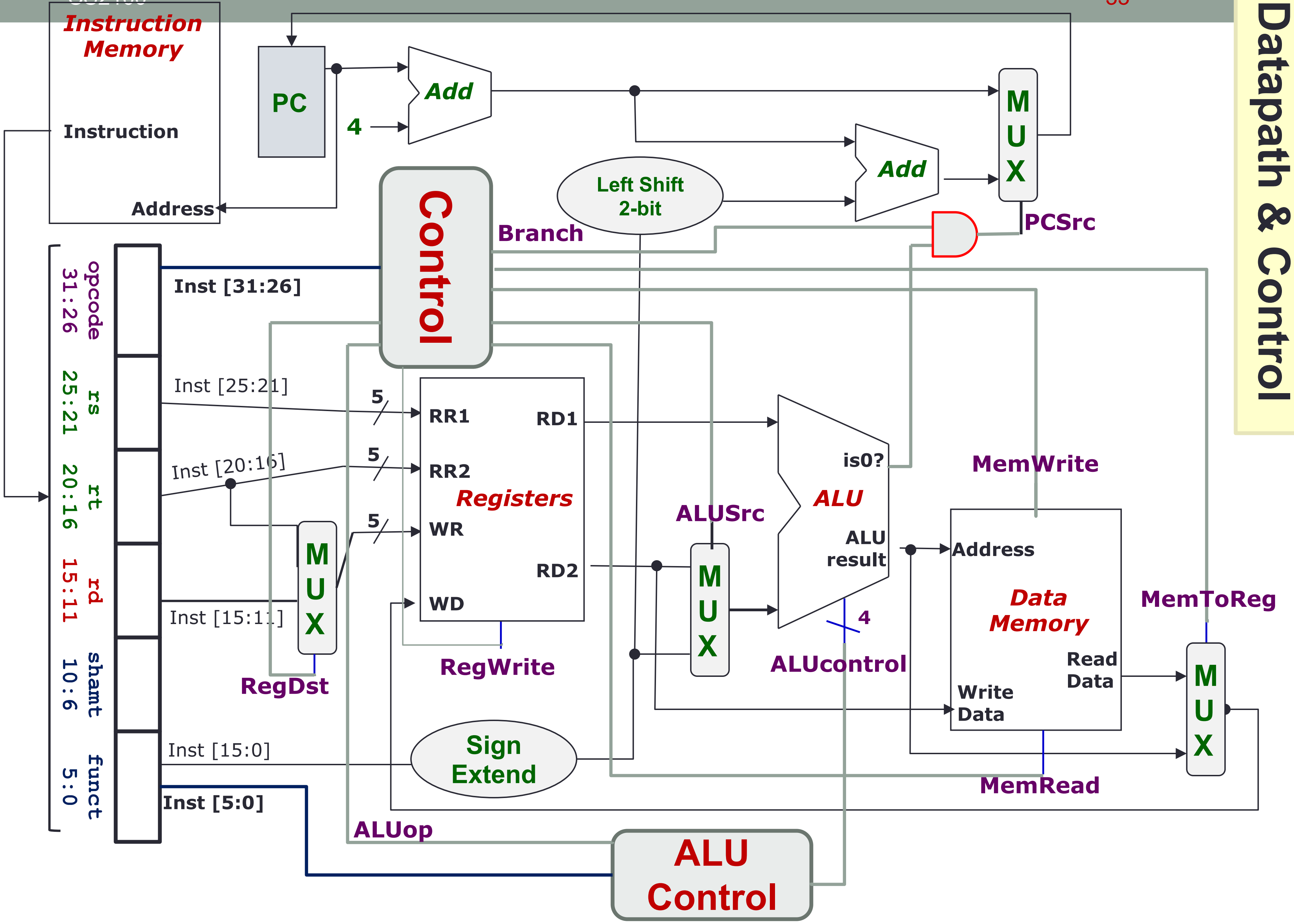


Control Design: **Outputs**

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	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



Datapath & Control



Exercise #1: **Pipeline Control Signals**

- Indicate the control signals **utilized in each cycle** for the following sequence of instructions: `add`, `sub`, `lw`, `sw`

Cycle	RegDst	ALUSrc	ALUOp1	ALUOp0	MemRead	MemWrite	MemToReg	RegWrite
1								
2								
3								
4								
5								
6								
7								
8								

PIPELINE PERFORMANCE

Single Cycle Processor: **Performance**

- Cycle time:
  - $CT_{seq} = \sum_{k=1}^{N} T_k$
  - T_k= Time for operation in stage k
  - N = Number of stages

- Total Execution Time for **I** instructions:
  - $Time_{seq}$ = Cycles $\times CycleTime$

    $= I \times CT_{seq} = \boldsymbol{I} \times \sum_{k=1}^{N} \boldsymbol{T_k}$

Single Cycle Processor: **Example**

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

- Cycle Time
  - Choose the longest total time = **8ns**
- To execute **100 instructions:**
  - **100 x 8ns = 800ns**

Multi-Cycle Processor: **Performance**

- Cycle time:
  - $CT_{multi} = \max(T_k)$
  - $\max(T_k) =$ longest stage duration among the N stages

- Total Execution Time for **I** instructions:
  - $Time_{multi}$ = Cycles $\times CycleTime$

    $= \mathrm{I} \times \boldsymbol{Average\ CPI} \times CT_{multi}$
  - Average CPI is needed because each instruction take different number of cycles to finish

Multi-Cycle Processor: **Example**

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

- Cycle Time:
  - Choose the longest stage time = **2ns**
- To execute **100 instructions**, with a given **average CPI of 4.6**
  - **100 x 4.6 x 2ns = 920ns**

Pipeline Processor: **Performance**

- Cycle Time:
  - $CT_{pipe} = \max(T_k) + T_d$
  - $\max(T_k) =$ longest time among the N stages
  - T_d = Overhead for pipelining, e.g. pipeline register
- Cycles needed for **I** instructions:
  - $\mathrm{I} + \mathrm{N} - 1$
  - $\mathrm{N} - 1$ is the cycles wasted in filling up the pipeline
- Total Time needed for **I** instructions :
  - $Time_{pipe} = Cycle \times CT_{pipe}$

    $= (\mathrm{I} + \mathrm{N} - 1) \times (\max(T_k) + T_d)$

Pipeline Processor: **Example**

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

- Cycle Time
  - assume pipeline register delay of **0.5ns**
  - longest stage time + overhead = **2 + 0.5 = 2.5ns**
- To execute **100 instructions:**
  - **(100 + 5 – 1) x 2.5ns = 260ns**

Pipeline Processor: **Ideal Speedup**

- Assumptions for ideal case:
  - Every stage takes the same amount of time:

    ➔ $\sum_{k=1}^{N} T_k = N \times T_k$

  - No pipeline overhead ➔ $T_d = 0$
  - Number of instructions **I**, is much larger than number of stages, **N**
- Note: The above also shows **how pipeline processor loses performance**

Pipeline Processor: **Ideal Speedup**

- $Speedup_{pipe} = \frac{Time_{seq}}{Time_{pipe}}$

$$= \frac{\mathrm{I} \times \sum_{k=1}^{N} T_k}{(\mathrm{I}+\mathrm{N}-1)\times(\max(T_k)+T_d)}$$

$$= \frac{I\times N\times T_k}{(I+N-1)\times T_k}$$

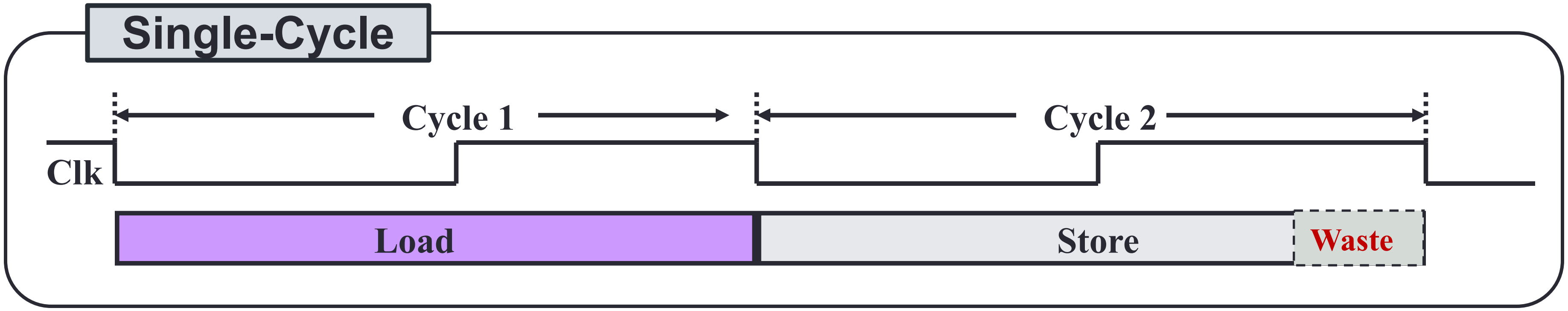
$$\approx \frac{I\times}{I}$$

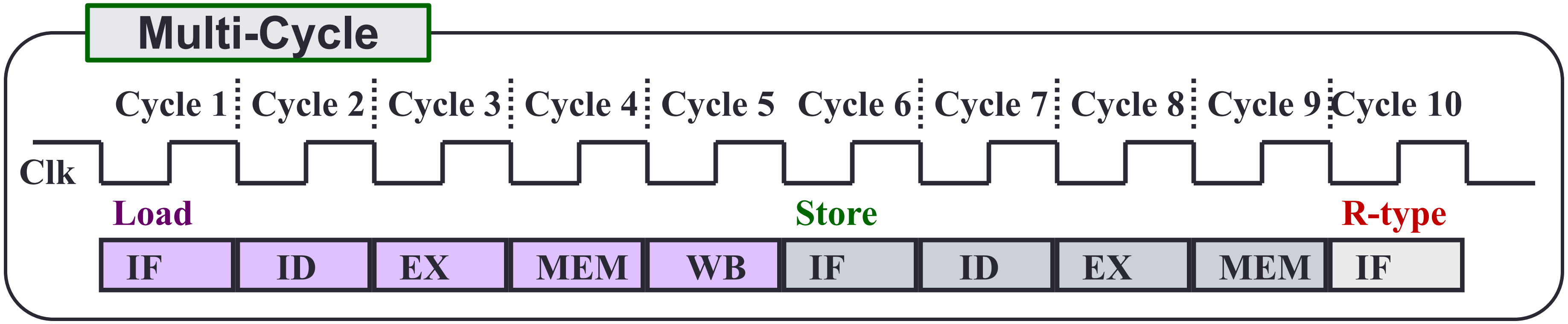
$$\approx$$

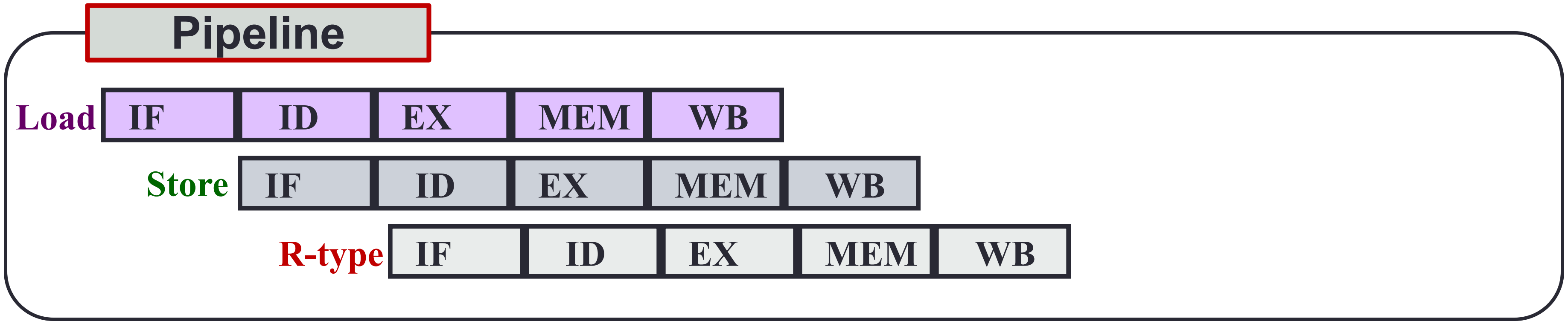
Conclusion:

Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages

Summary for different implementations







Review Question

- Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

a) How many cycles will it take to execute the code on a single-cycle datapath?

b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?

c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?

d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?

PIPELINE HAZARDS

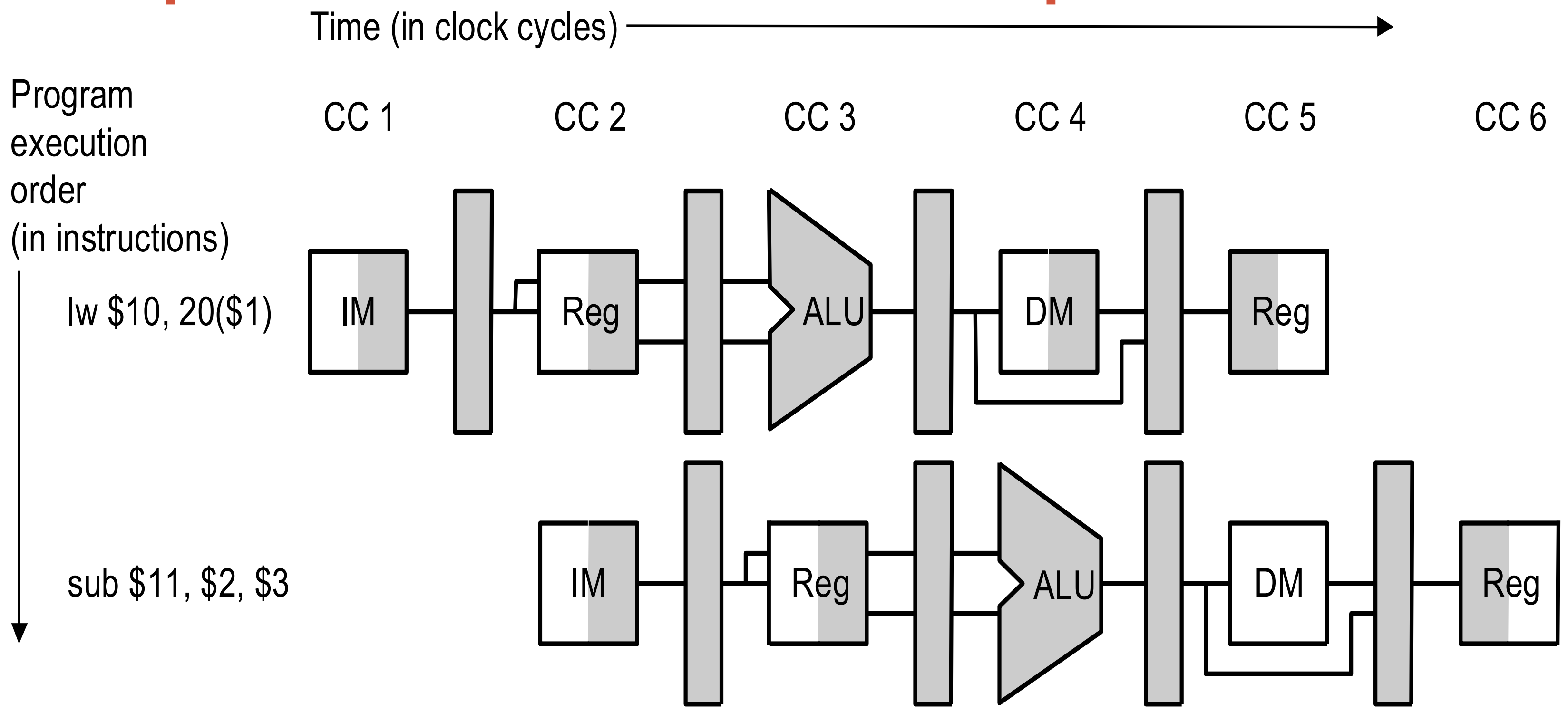
Grinding to a halt…..

Pipeline Hazards

- Speedup from pipeline implementation:
  - Based on the assumption that a new instructions can be “pumped” into pipeline every cycle
- However, there are **pipeline hazards**
  - Problems that prevent next instruction from immediately following previous instruction
  - **Structural hazards:**
    - Simultaneous use of a hardware resource
  - **Data hazards:**
    - Data dependencies between instructions
  - **Control hazards:**
    - Change in program flow

Instruction Dependencies

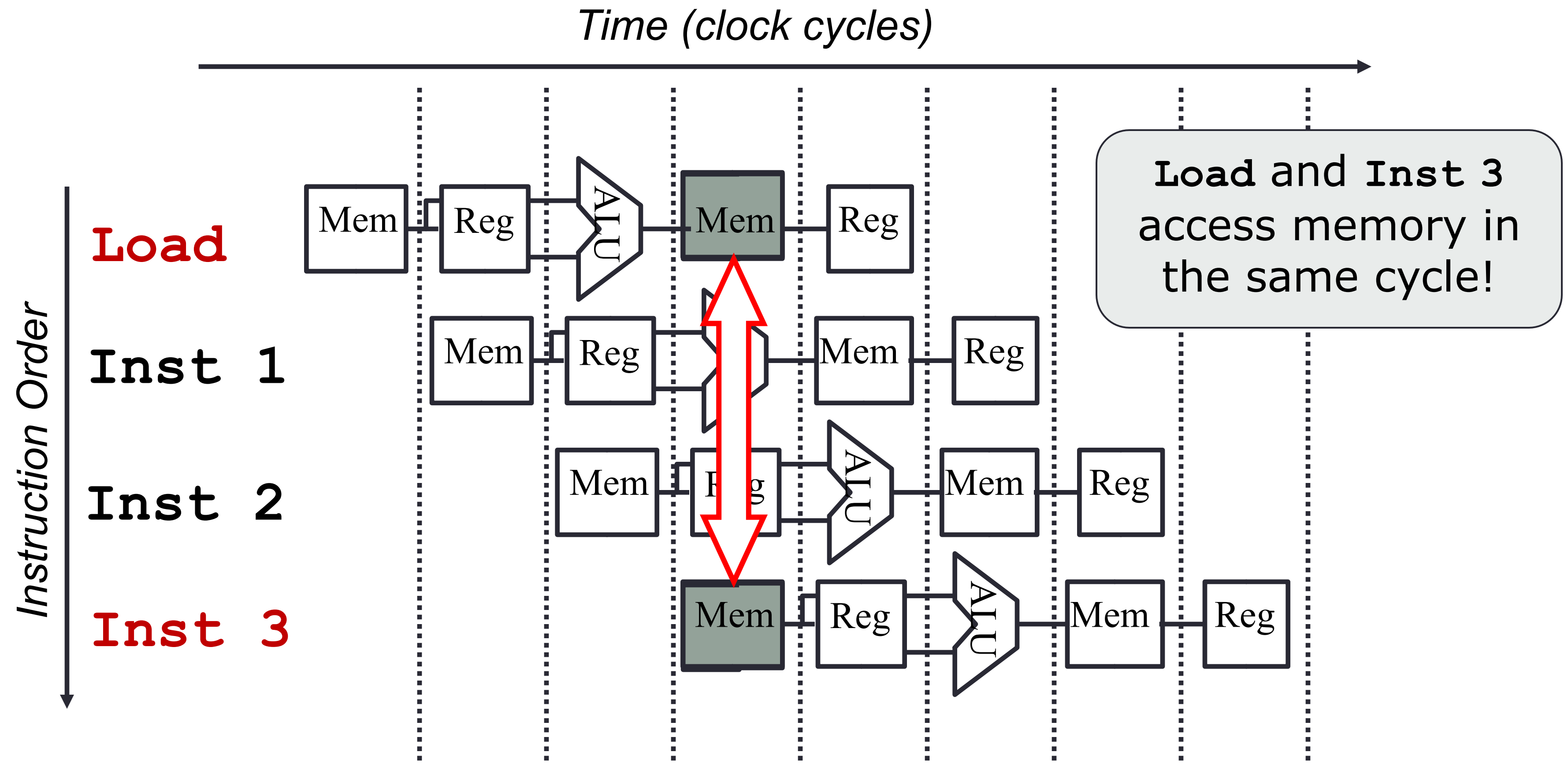
Graphical Notation for Pipeline



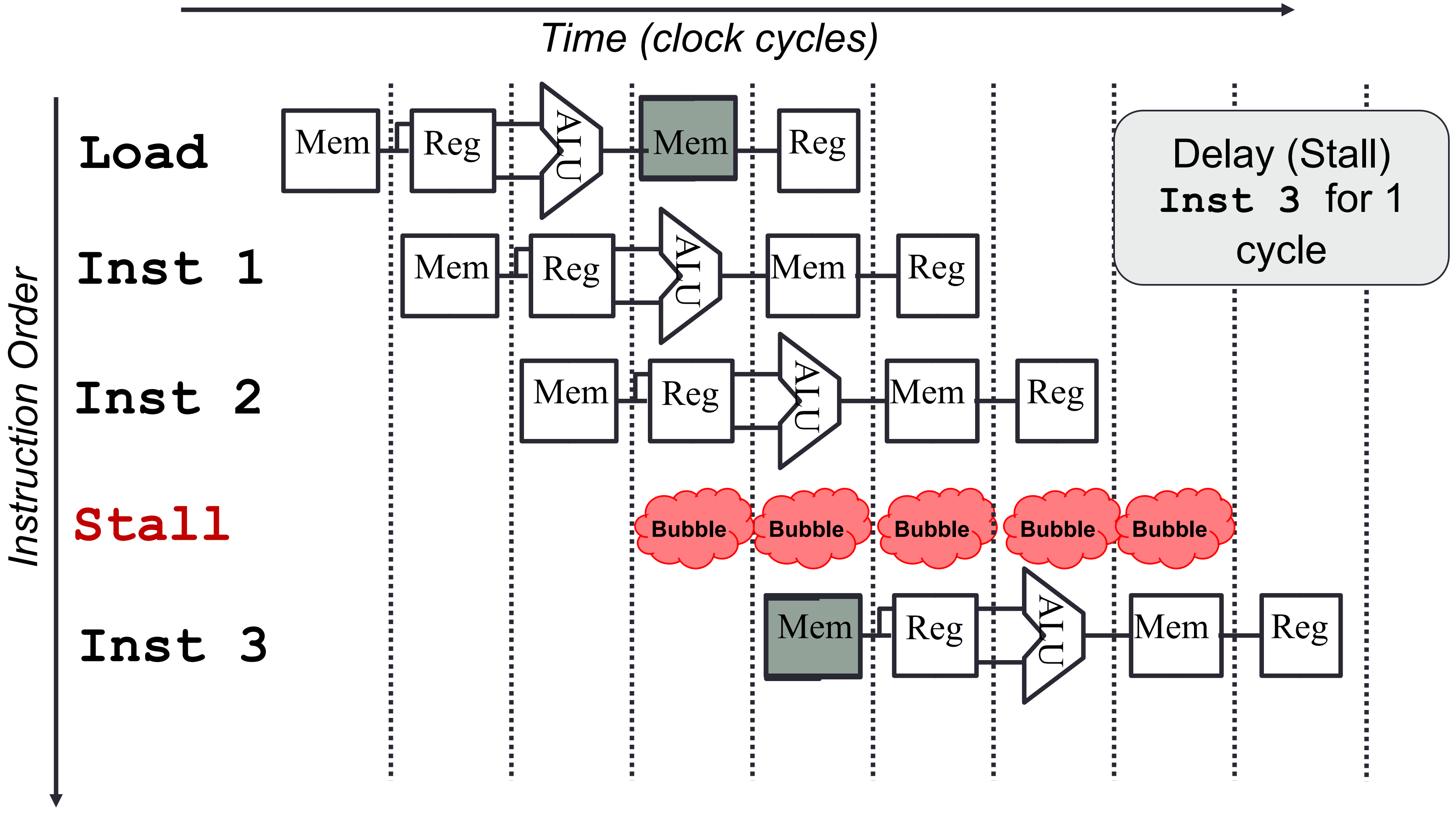
- Horizontal = the stages of an instruction
- Vertical = the instruction**s** in different pipeline stages

Structure Hazard: Example

- If there is only a **single memory module:**

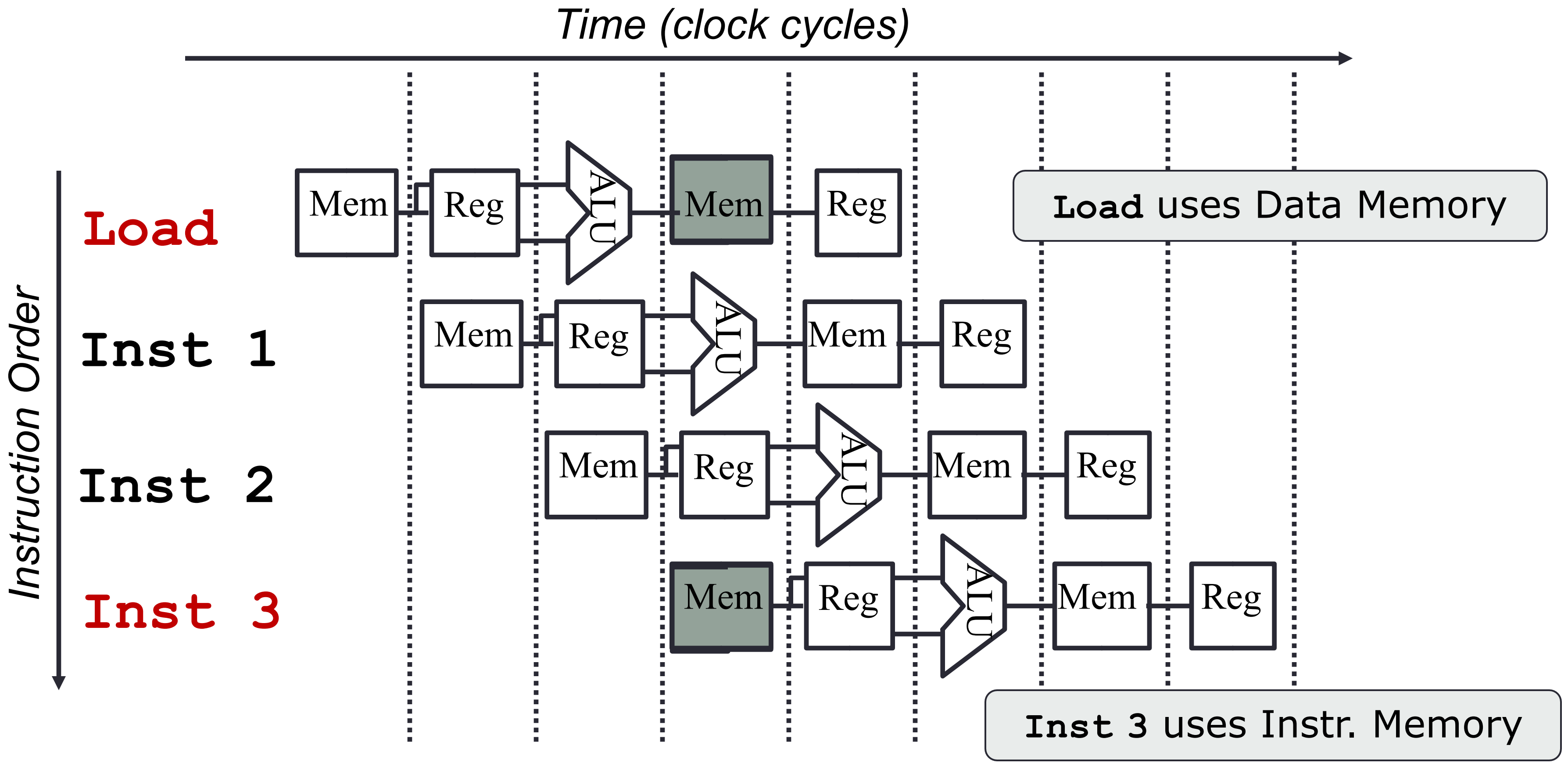


Solution 1: **Stall the Pipeline**

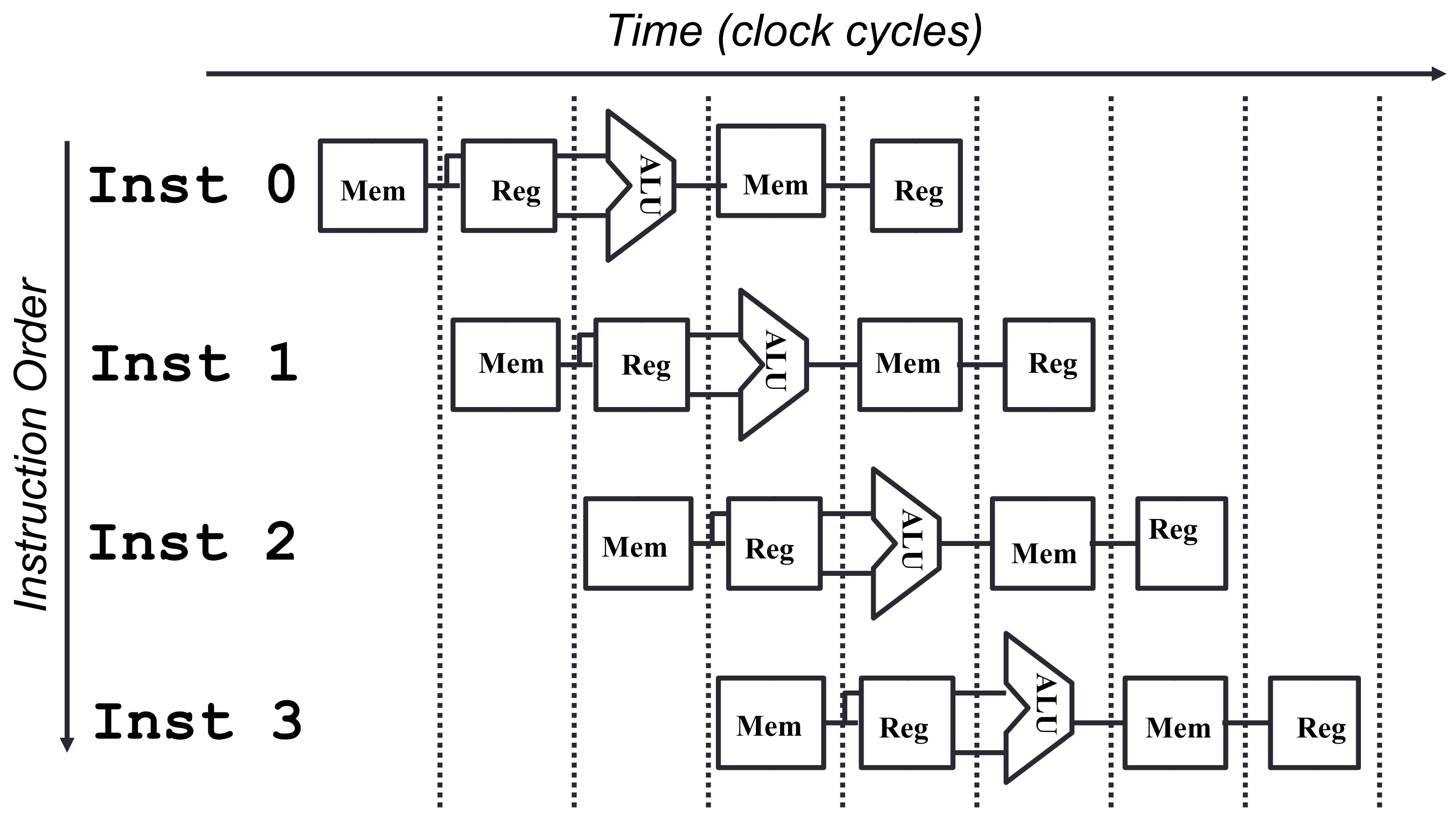


Solution 2: Separate Memory

- Split memory into:
  - **Data** and **Instruction** memory



Quiz: **Is there another conflict?**



Instruction Dependencies

- Instructions can have relationship that prevent pipeline execution:
  - Although a partial overlap maybe possible in some cases
- When different instructions accesses (read/write) the same register
  - Register contention is the cause of dependency
  - Known as **data dependency**
- When the execution of an instruction depends on another instruction
  - Control flow is the cause of dependency
  - Known as **control dependency**
- Failure to handle dependencies can affect **program correctness**!

Data Dependency: **RAW**

- "**R**ead-**A**fter-**W**rite" **Definition:**
  - Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
  - Also known as **true data dependency**

```
i1: add $1, $2, $3 #writes to $1
i2: sub $4, $1, $5 #reads from $1
```

- Effect of incorrect execution:
  - If `i2` reads register `$1` before `i1` can write back the result, `i2` will get a ***stale result (old result)***

Other Data Dependencies

- Similarly, we have:
  - **WAR**: **W**rite-**a**fter-**R**ead dependency
  - **WAW**: **W**rite-**a**fter-**W**rite dependency
- Fortunately, these dependencies **do not cause any pipeline hazards**
- They affects the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor

RAW Dependency: **Hazards?**

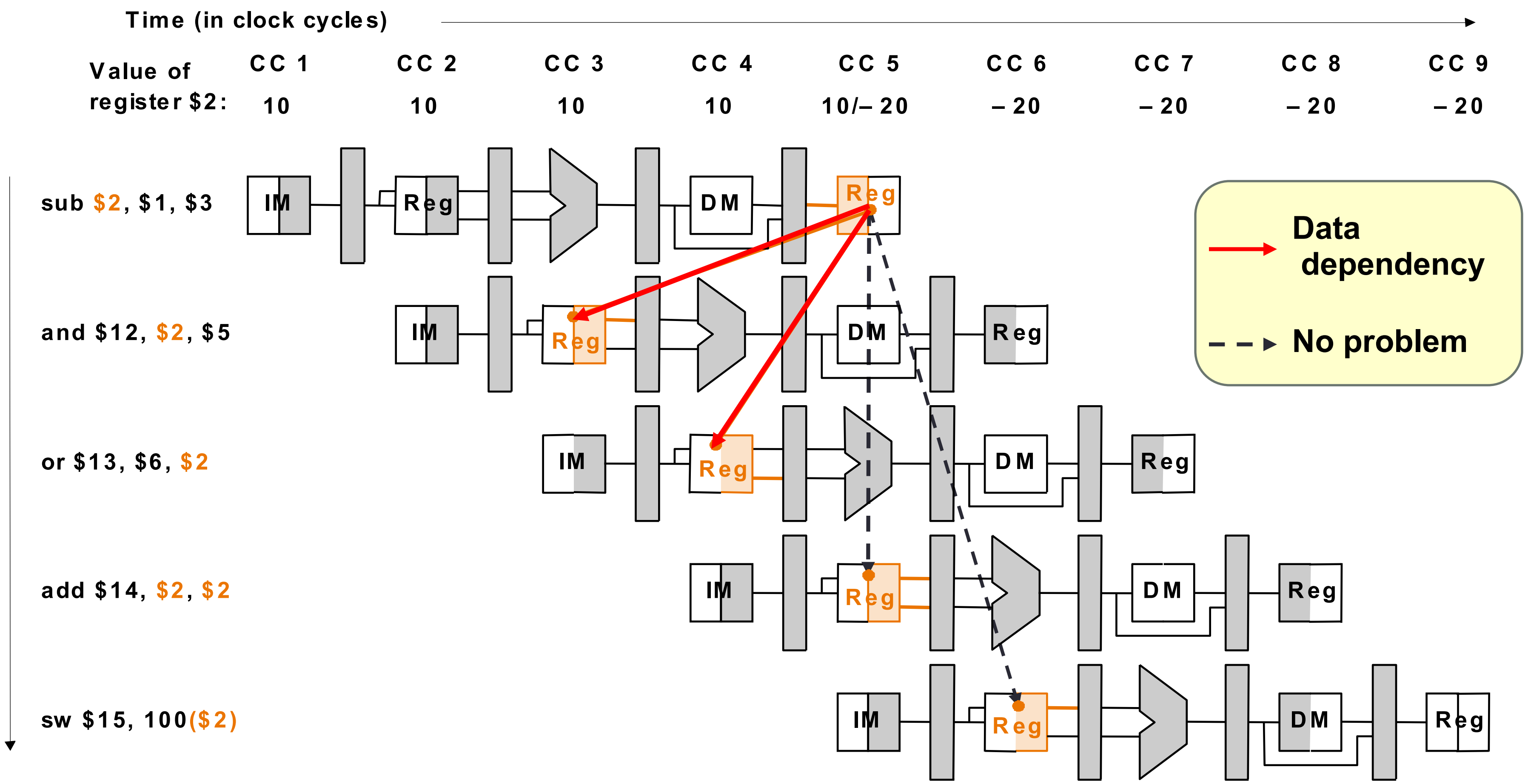
- Suppose we are executing the following code fragment:

```
sub $2, $1, $3        #i1
and $12, $2, $5       #i2
or  $13, $6, $2       #i3
add $14, $2, $2       #i4
sw  $15, 100($2)      #i5
```

- Note the multiple uses of register **$2**
- Question:
  - Which are the instructions require special handling?

RAW Data Hazards: **Illustration**

- Value from prior instruction is needed before write back



Observations

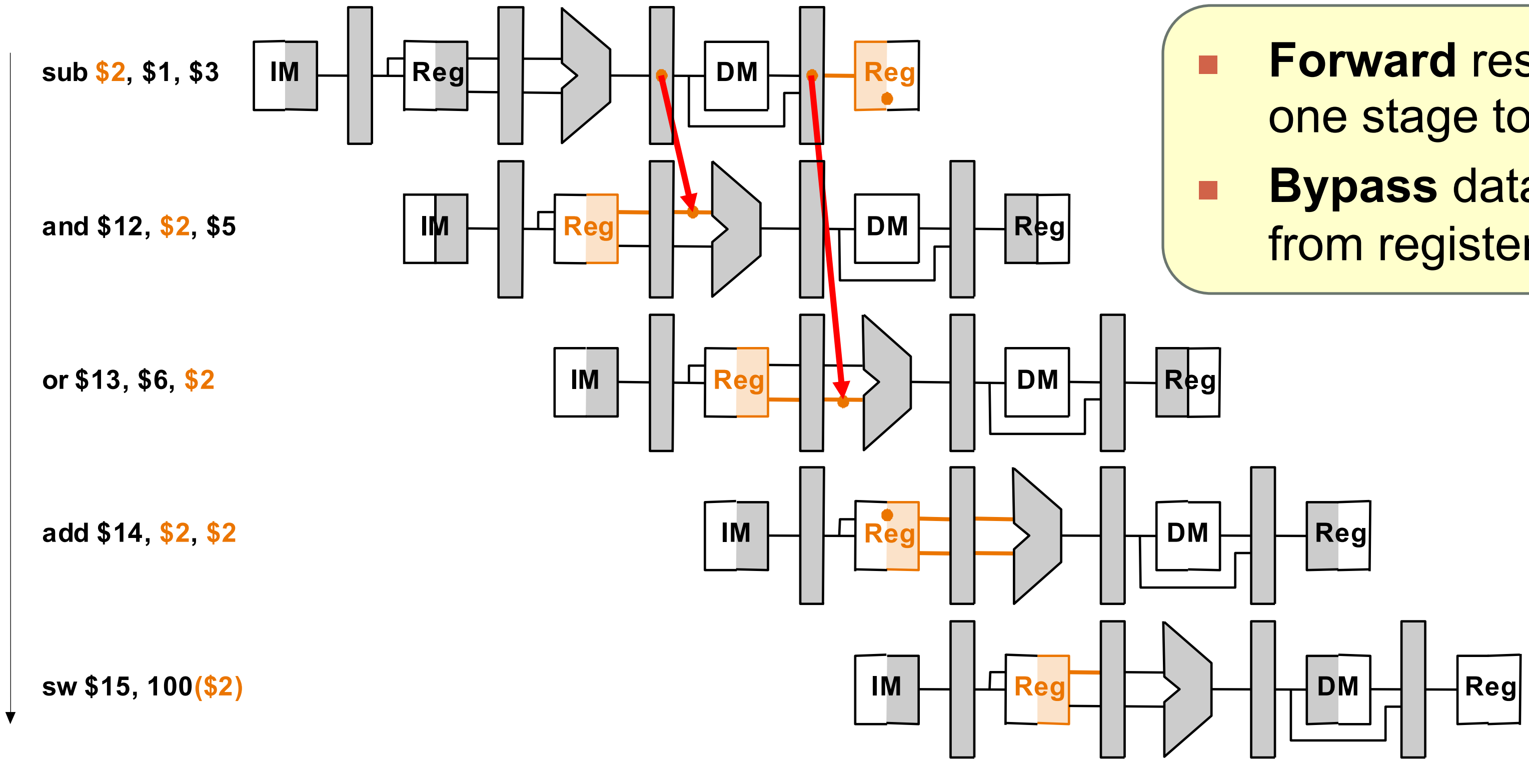
- Questions:
  - When is the result from `sub` instruction actually produced?
    - End of EX stage for `sub` or clock cycle 3
  - When is the data actually needed by `and`?
    - Beginning of `and`'s EX stage or clock cycle 4
  - When is the data actually needed by `or`?
    - Beginning of `or`'s EX stage or clock cycle 5
- **Solution:**
  - **Forward** the result to any trailing (later) instructions before it is reflected in register file

  ➔ **Bypass (replace)** the data read from register file

Solution: **Forwarding**

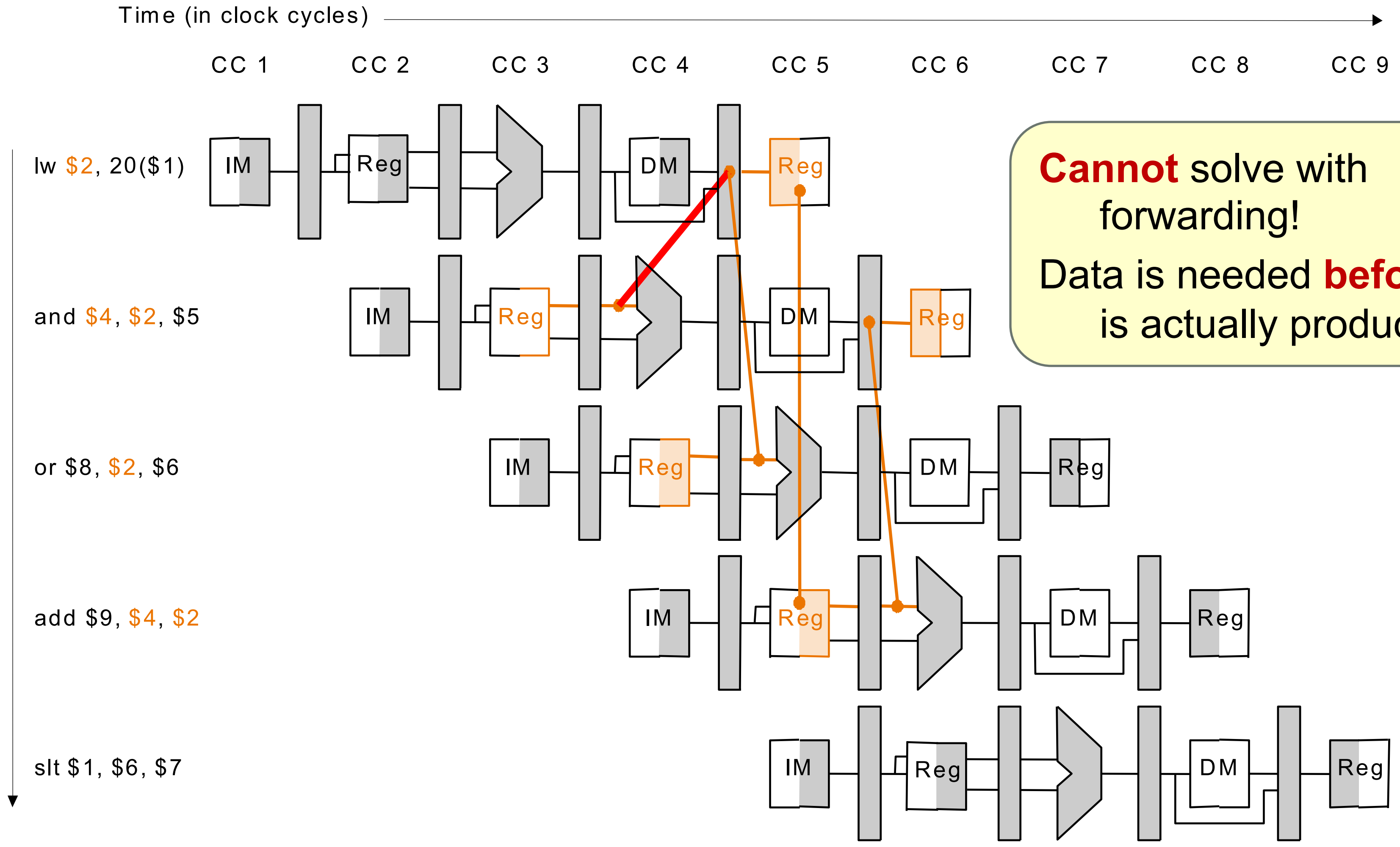
Time (in clock cycles) →

	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



- **Forward** results from one stage to another
- **Bypass** data read from register file

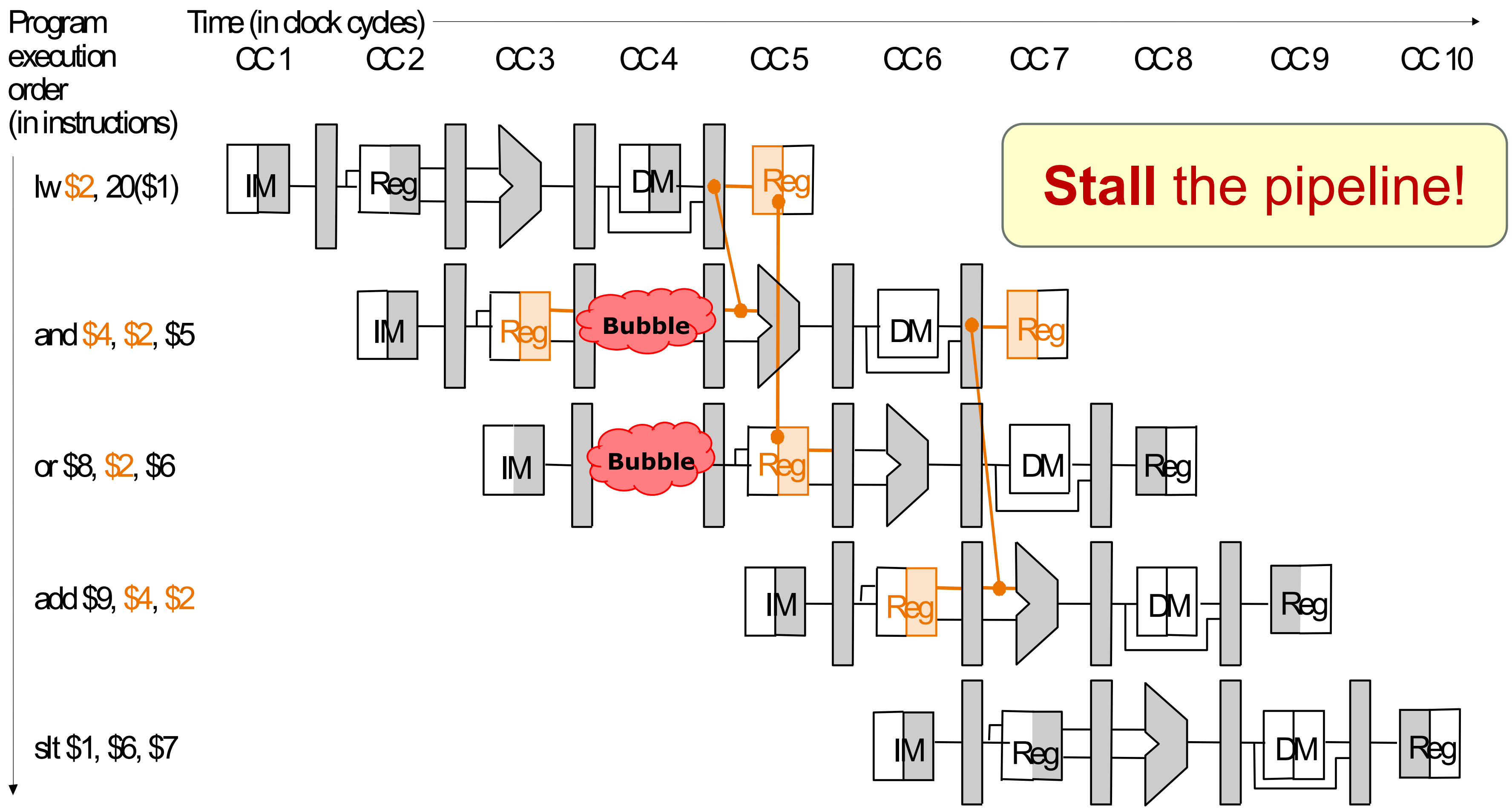
Data Hazard: **LOAD** Instruction



Cannot solve with forwarding!

Data is needed **before** it is actually produced!

Data Hazard: **LOAD** Instruction Solution



Stall the pipeline!

Exercise #2: Without Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline **without** forwarding?

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

Exercise #2: Without Forwarding

	1	2	3	4	5	6	7	8	9	10	11
`sub`	IF	ID	EX	MEM	WB						
`and`											
`or`											
`add`											
`sw`											

Exercise #3: Without Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline **without** forwarding?

```
    lw  $2,  20($3)
    and $12, $2, $5
    or  $13, $6, $2
    add $14, $2, $2
    sw  $15, 100($2)
```

Exercise #3: Without Forwarding

	1	2	3	4	5	6	7	8	9	10	11
`lw`	IF	ID	EX	MEM	WB						
`and`											
`or`											
`add`											
`sw`											

Exercise #4: With Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding?

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

Exercise #4: With Forwarding

	1	2	3	4	5	6	7	8	9	10	11
`sub`	IF	ID	EX	MEM	WB						
`and`											
`or`											
`add`											
`sw`											

Exercise #5: With Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding?

```
lw   $2,   20($3)
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

Exercise #5: With Forwarding

	1	2	3	4	5	6	7	8	9	10	11
`lw`	IF	ID	EX	MEM	WB						
`and`											
`or`											
`add`											
`sw`											

CONTROL DEPENDENCY

Control Dependency

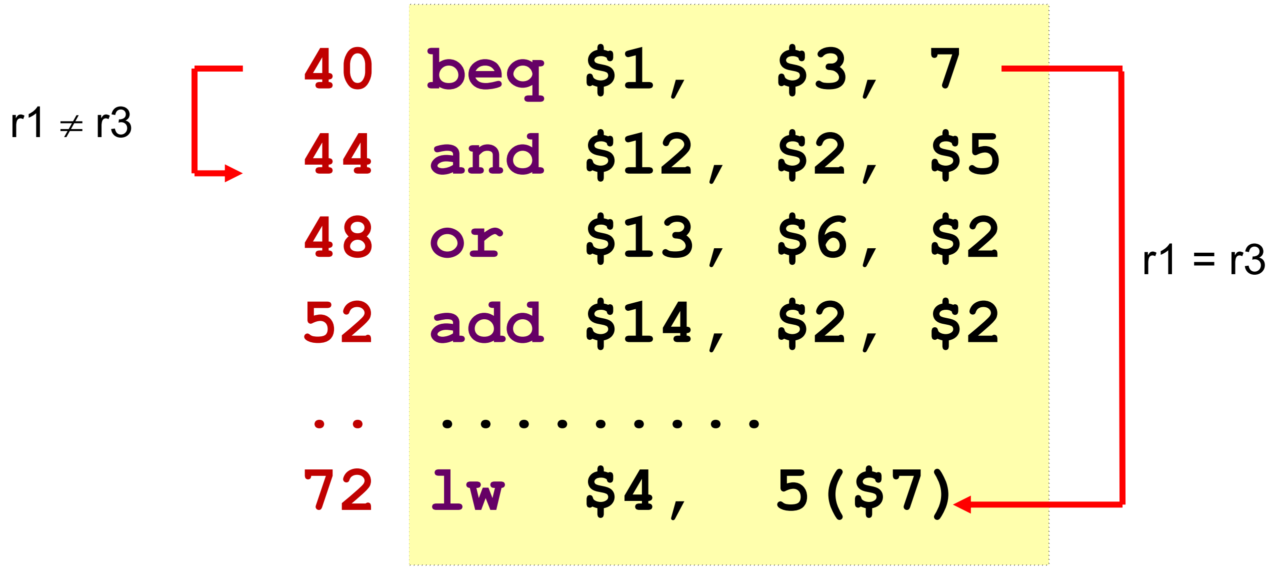
- **Definition:**
  - An instruction `j` is control dependent on `i` if `i` controls whether or not `j` executes
  - Typically `i` would be a branch instruction
- Example:

```
i1: beq $3, $5, label    # Branch
i2: add $1, $2, $4       # depends on i1
...    ...    ...
```

- Effect of incorrect execution:
  - If `i3` is allowed to execute before `i2` is determined, register `$1` maybe incorrectly changed!

Control Dependency: **Example**

- Let us turn to a code fragment with a conditional branch:

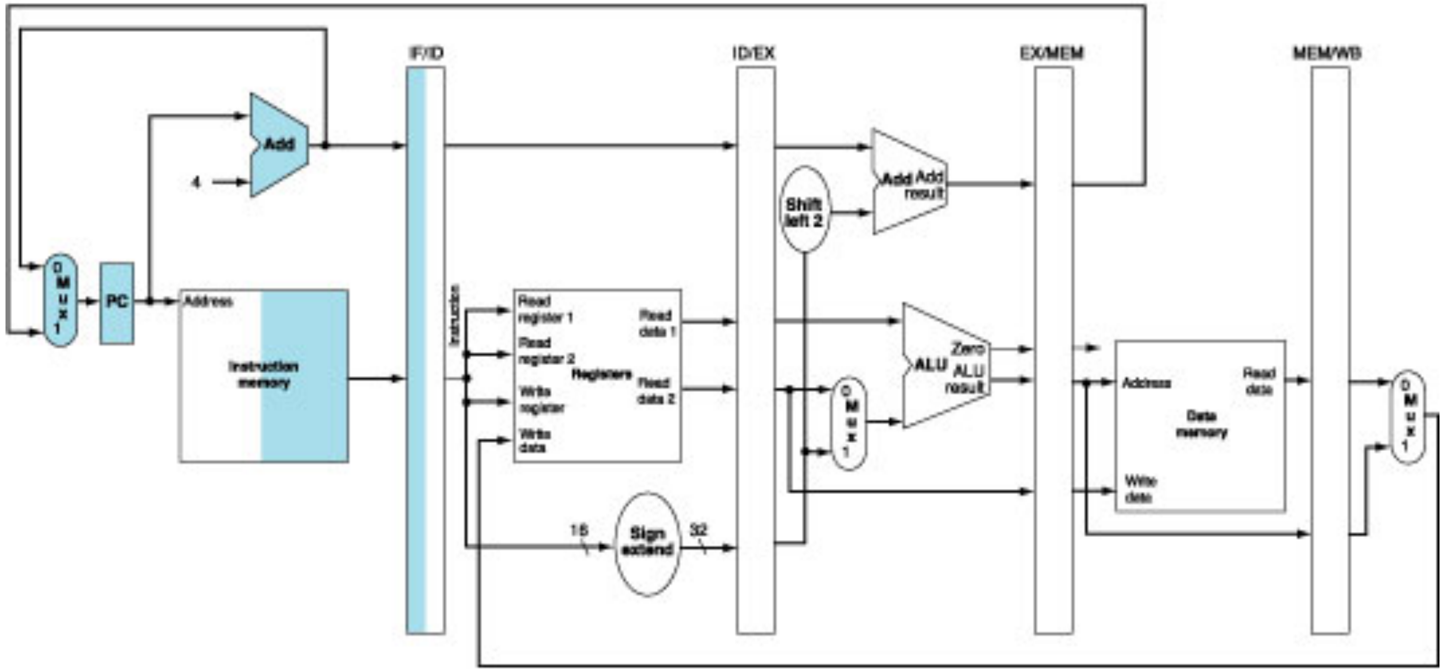


```
40  beq $1,  $3, 7
44  and $12, $2, $5
48  or  $13, $6, $2
52  add $14, $2, $2
..  .........
72  lw  $4,  5($7)
```

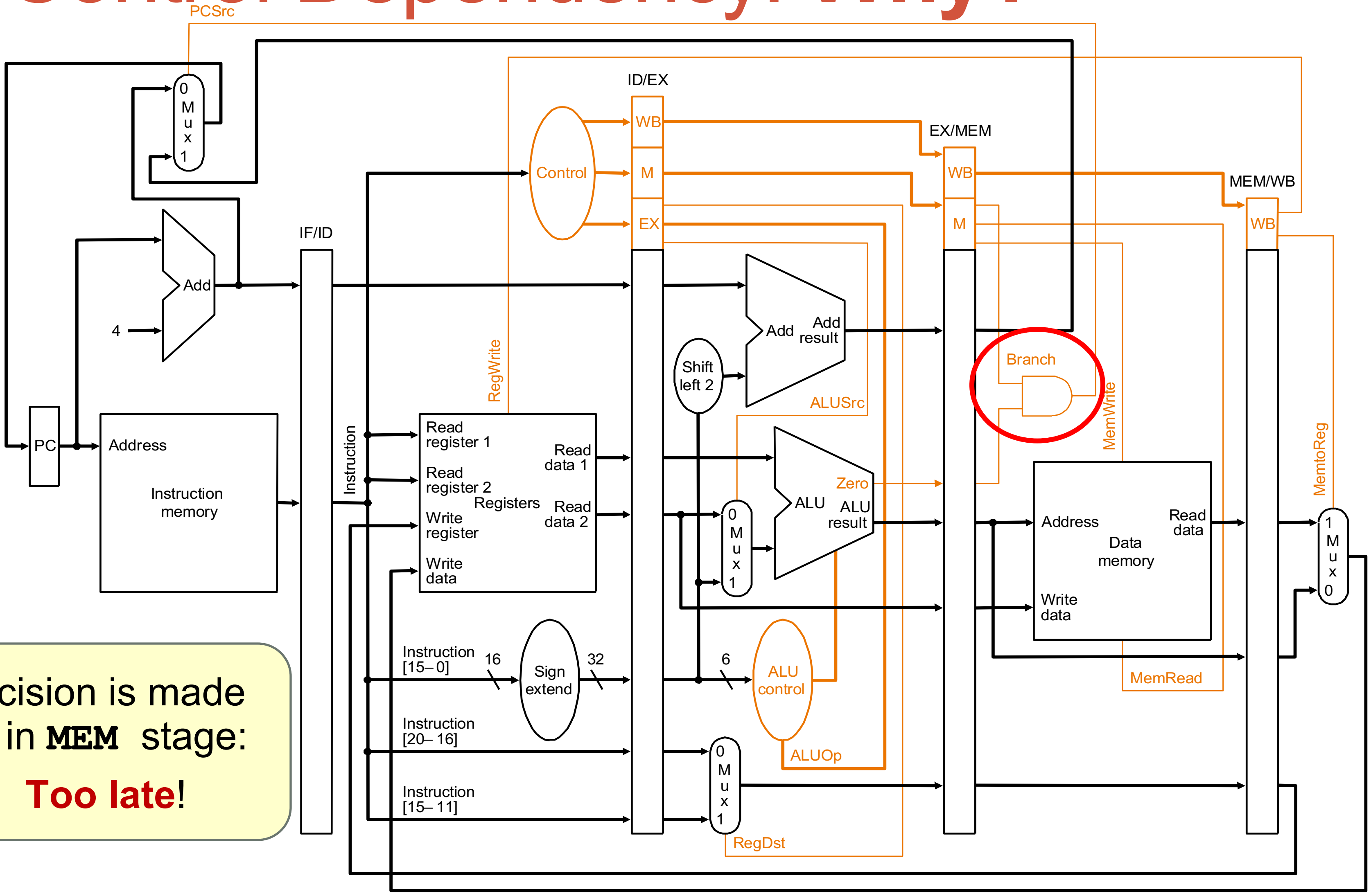
- How does the code affect a pipeline processor?

Pipeline Execution: `IF` Stage

- Read instruction from memory using the address in PC and put it in `IF/ID` register
- PC address is incremented by 4 and then written back to the PC for next instruction



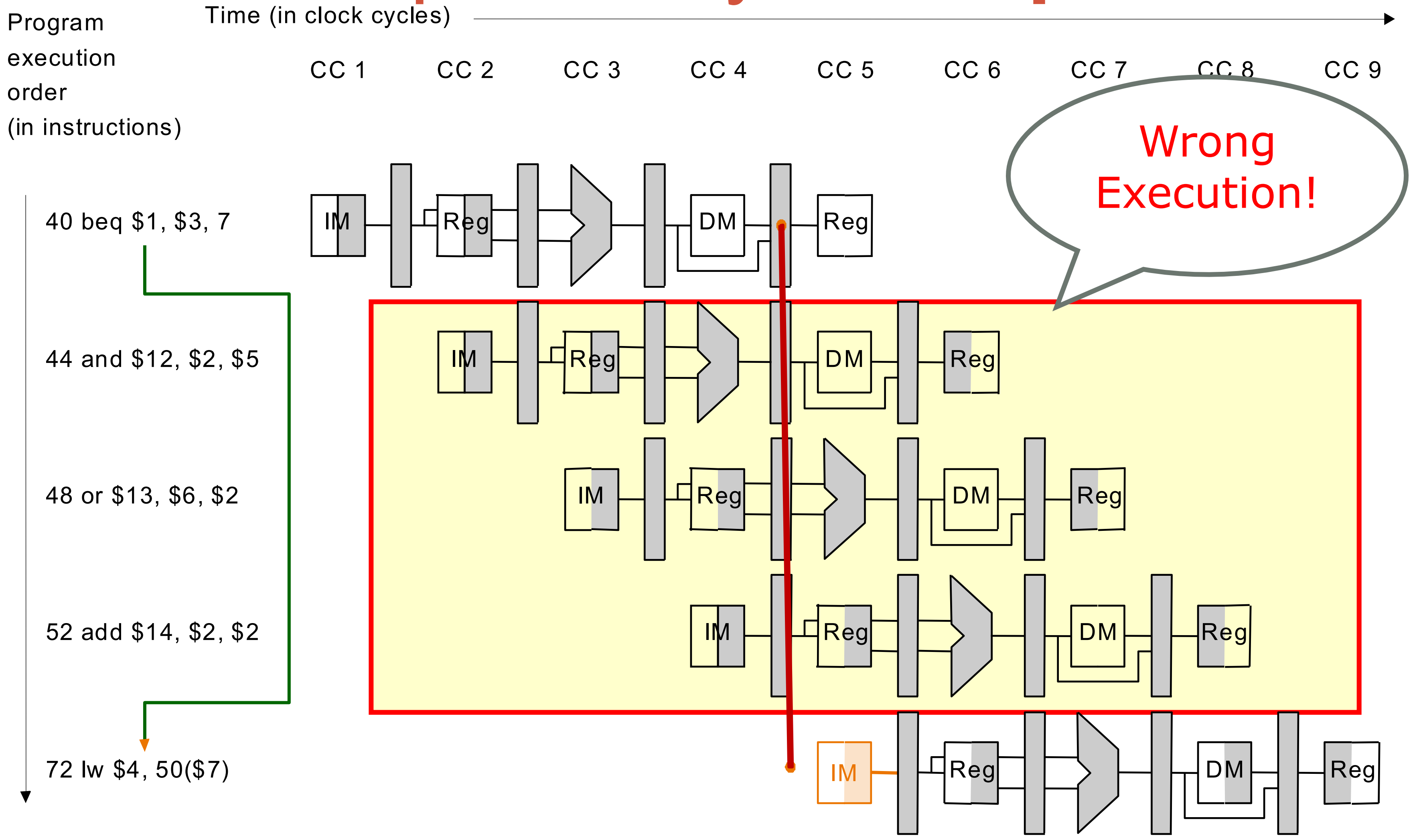
Control Dependency: **Why?**



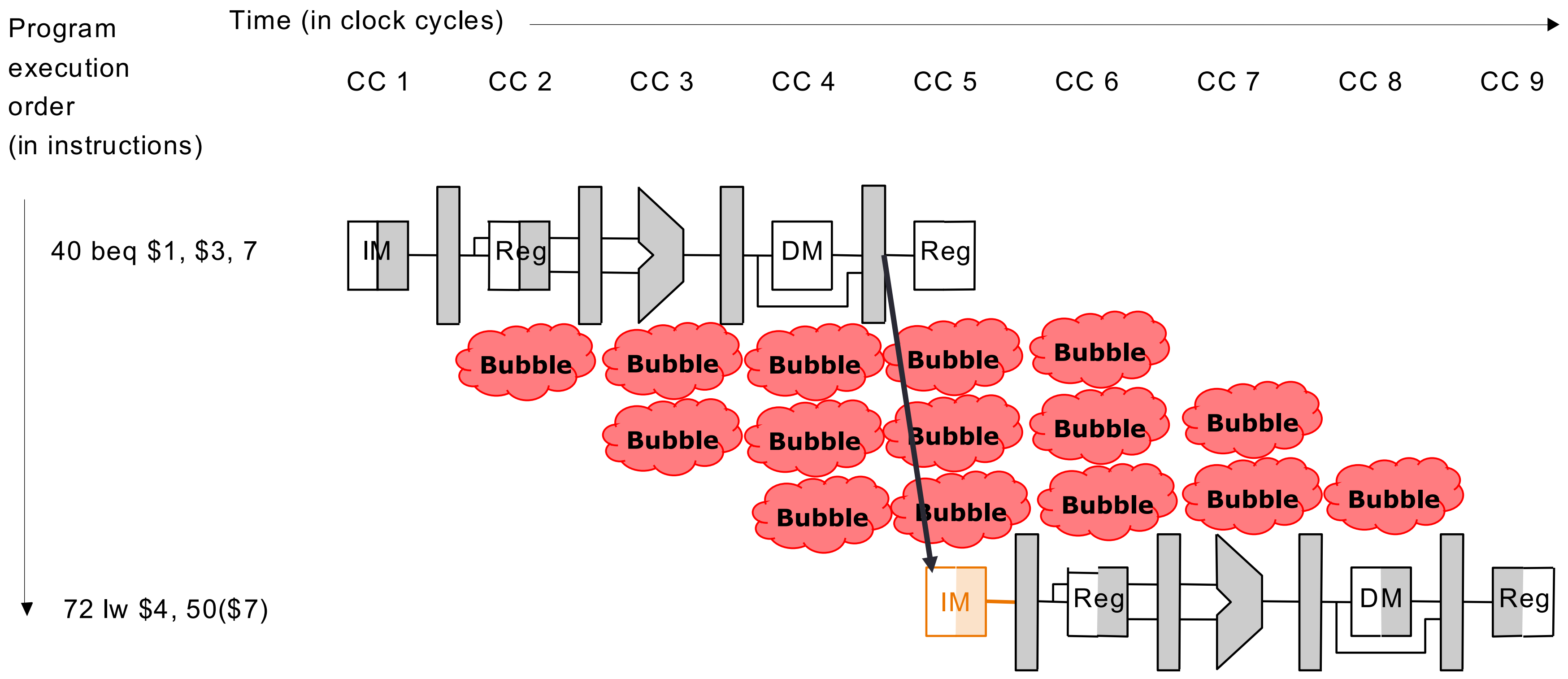
Decision is made in **MEM** stage:

Too late!

Control Dependency: **Example**



Control Hazards: **Stall Pipeline?**



- Wait until the branch outcome is known and then fetch the correct instructions

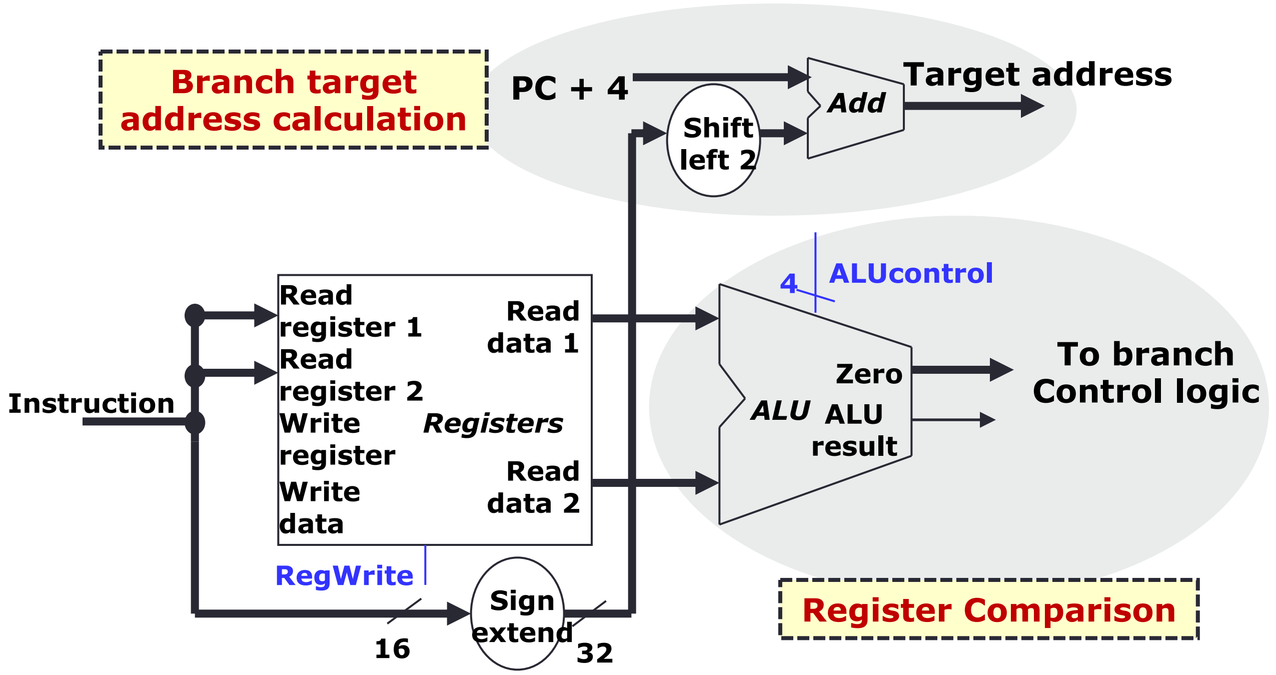
➔ Introduces **3 clock cycles delay**

Control Hazards: **Reducing the Penalty**

- Branching is very common in code:
  - A 3 cycles stall penalty is too heavy!
- Many techniques invented to reduce the control hazard penalty:
  - Move branch decision calculation to earlier pipeline stage
    - **Early Branch Resolution**
  - Guess the outcome before it is produced
    - **Branch Prediction**
  - Do something useful while waiting for the outcome
    - **Delayed Branching**

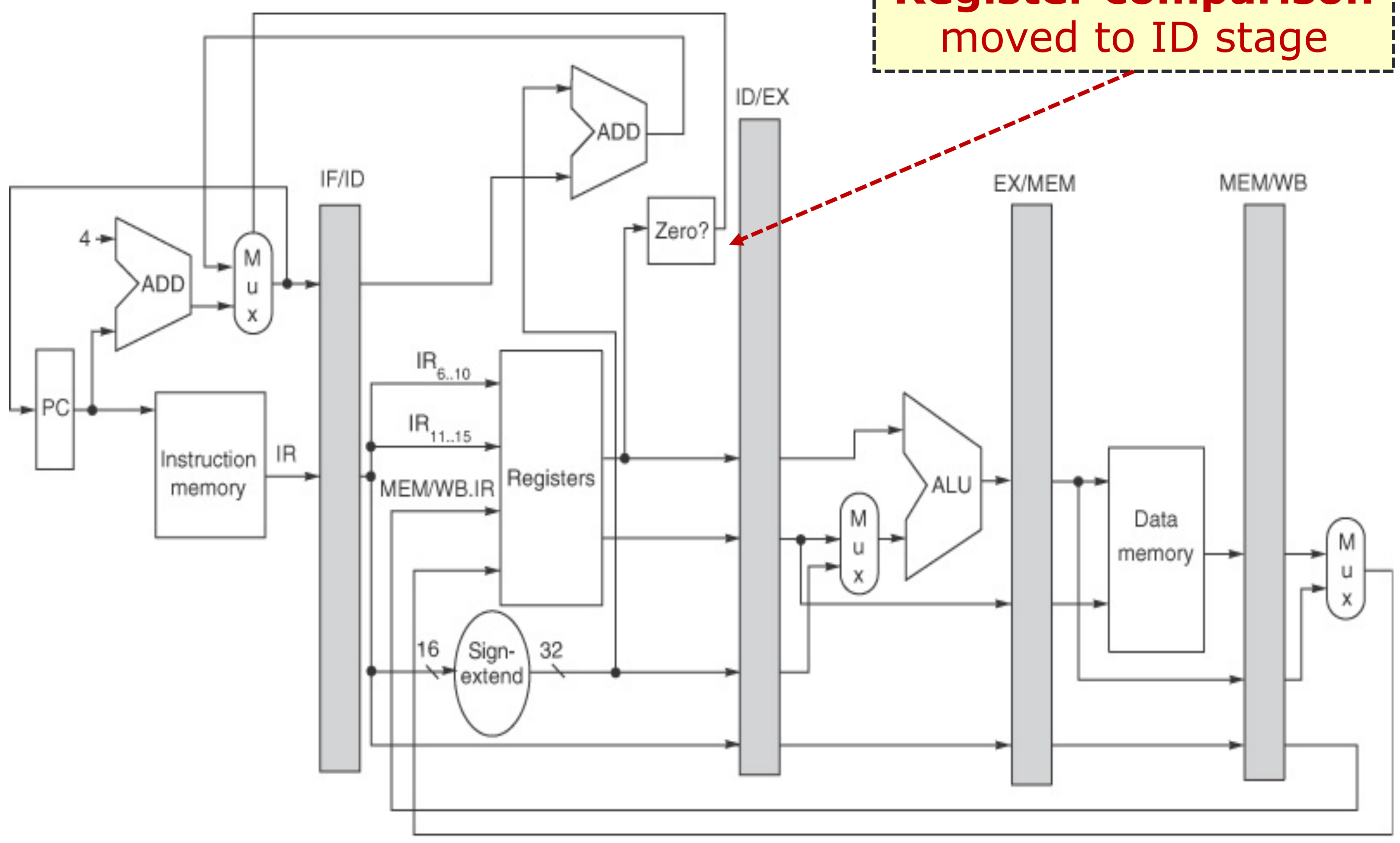
Reduce Stalls: Early Branch(1/3)

- Make decision in **ID** stage instead of **MEM**
  - Move branch target address calculation
  - Move register comparison ➔ cannot use ALU for register comparison any more

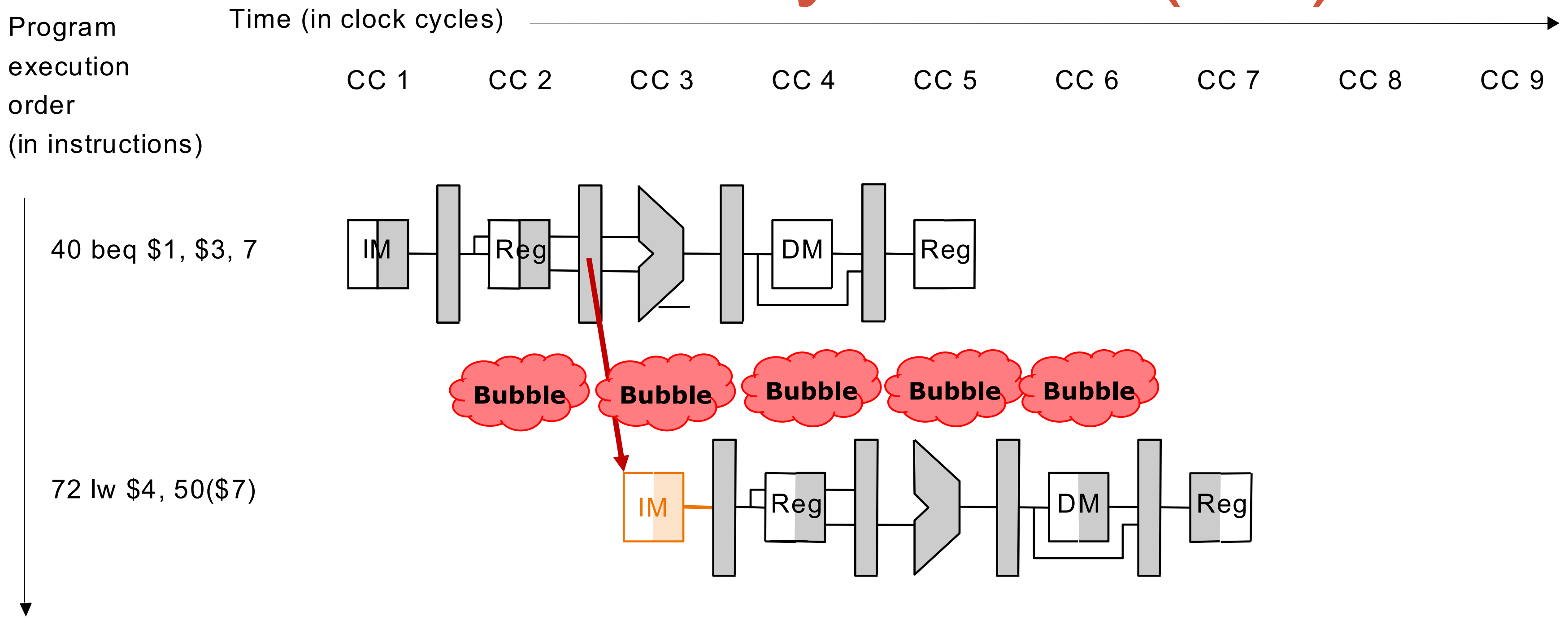


Reduce Stalls: Early Branch(2/3)

Register comparison moved to ID stage



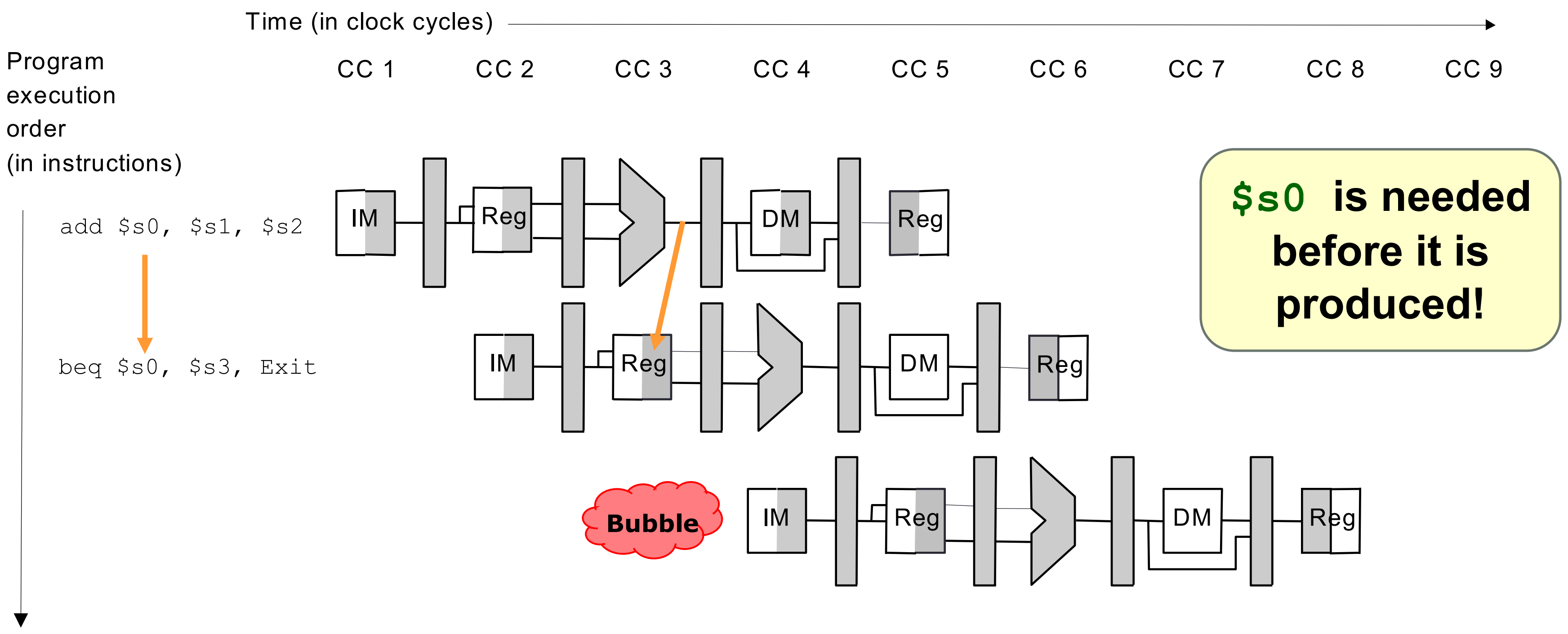
Reduce Stalls: Early Branch(3/3)



- Wait until the branch decision is known:
  - Then fetch the correct instructions
- Reduced to **1 clock cycle delay**

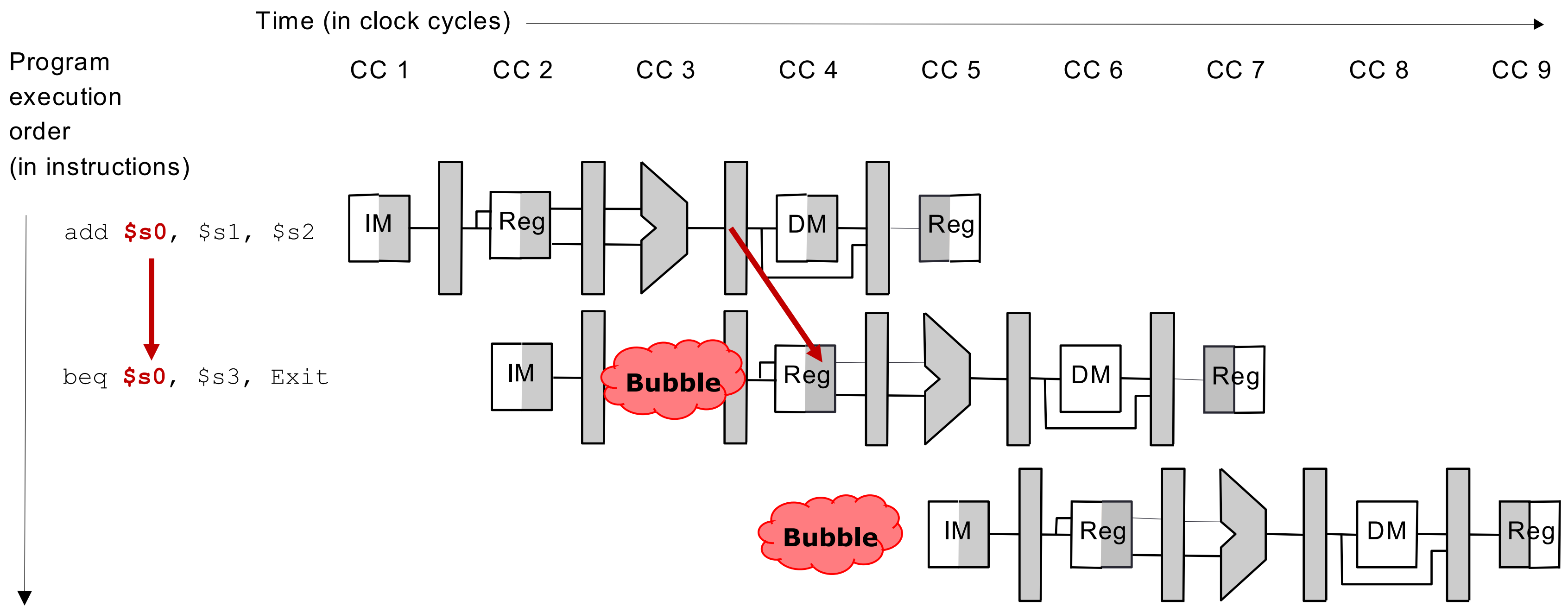
Early Branch: **Problems**(1/3)

- However, if the register(s) involved in the comparison is produced by preceding instruction:
  - Further stall is still needed!



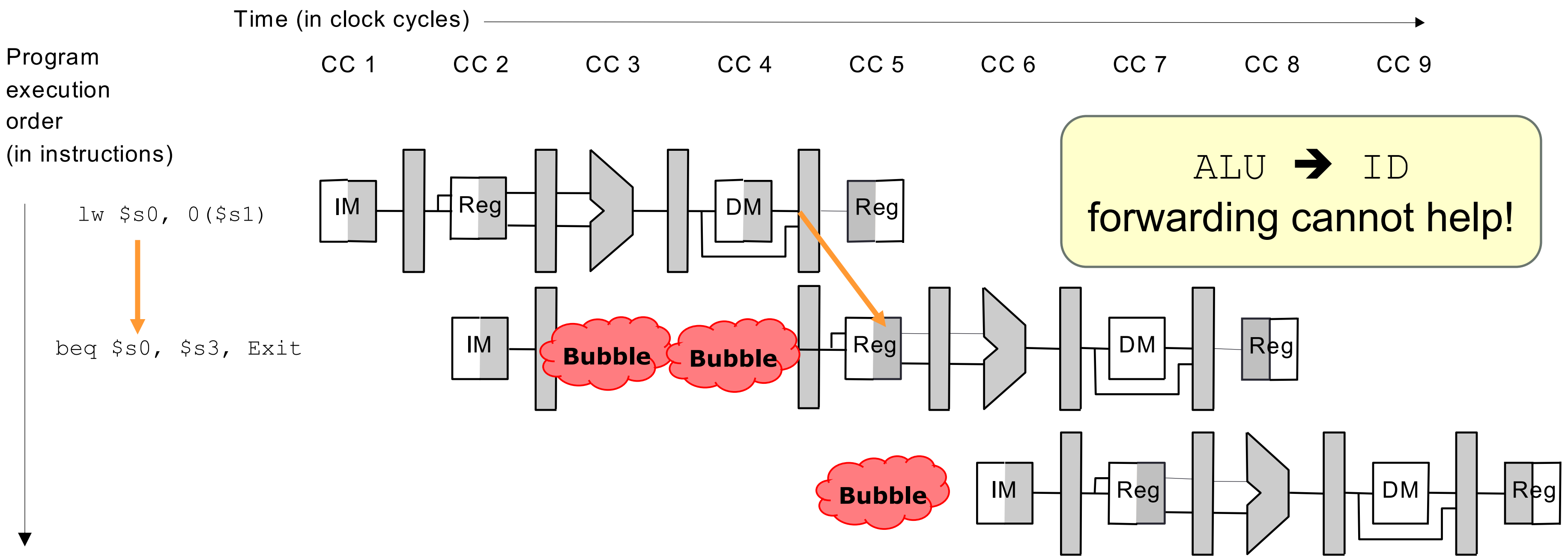
Early Branch: **Problems**(2/3)

- **Solution:**
  - Add forwarding path from ALU to ID stage
  - **One clock cycle delay** is still needed



Early Branch: **Problems**(3/3)

- Problem is worse with **load** followed by **branch**
- **Solution:**
    - MEM to ID forwarding and 2 more stall cycles!
    - In this case, we ended up with 3 total stall cycles ➔ no improvement!



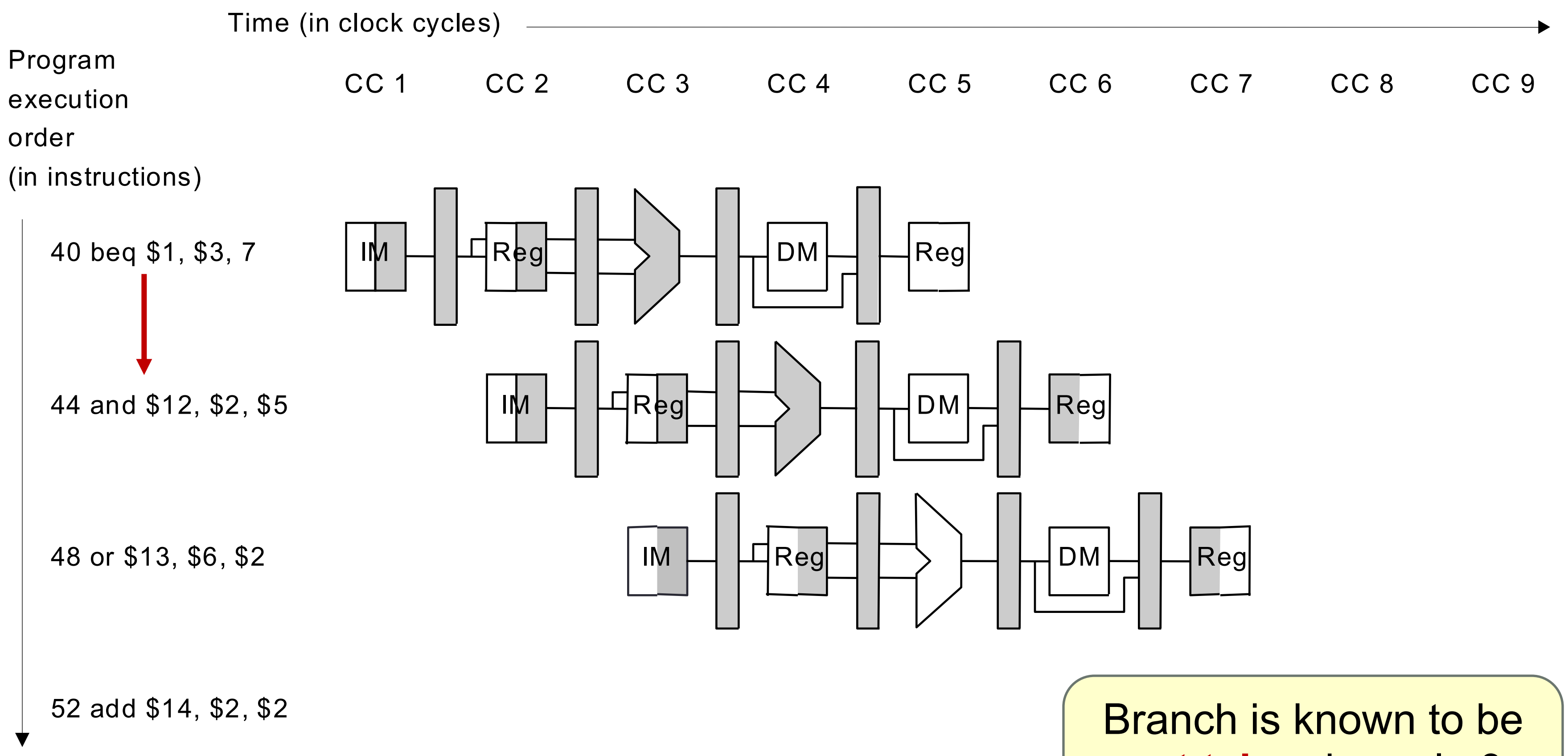
Reduce Stalls: **Branch Prediction**

- There are many branch prediction schemes
  - We only cover the simplest in this course ☺

- Simple prediction:
  - All branches are assumed to be **not taken**
  - ➔ Fetch the successor instruction and start pumping it through the pipeline stages

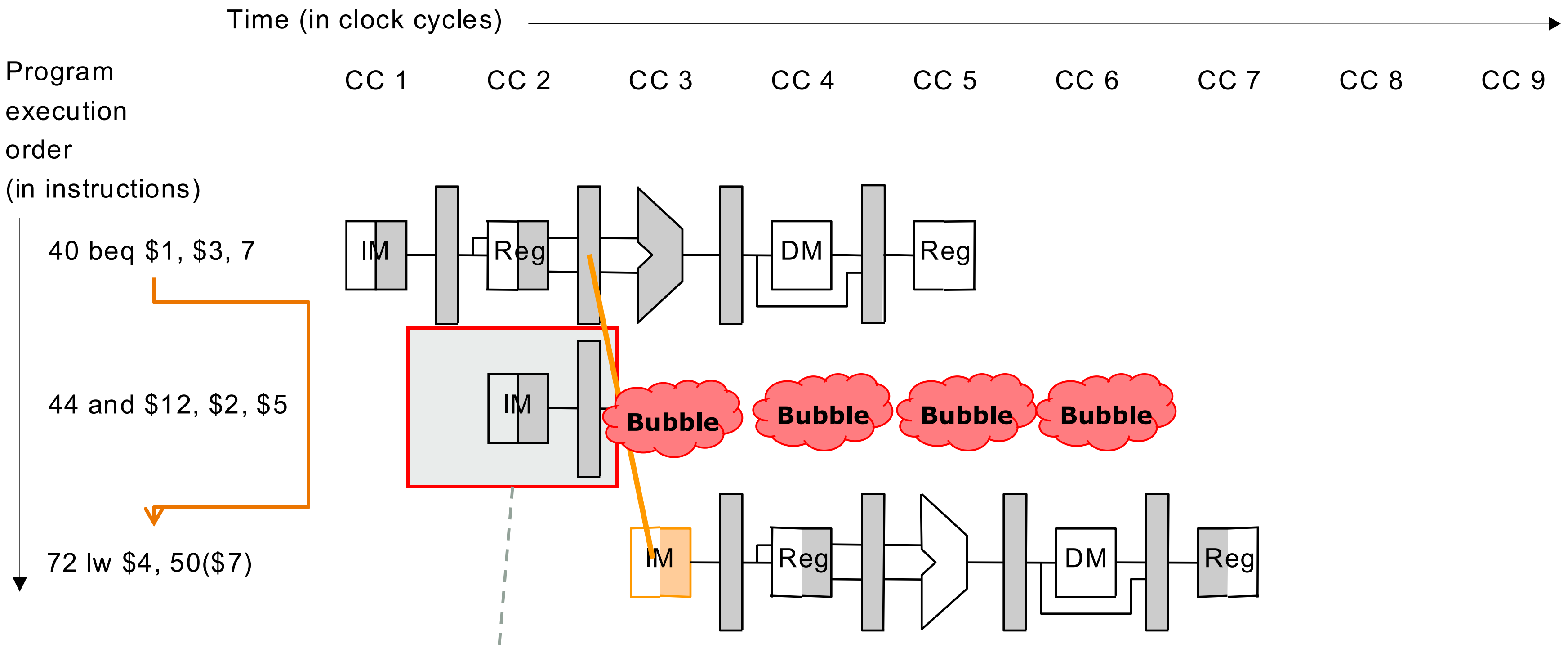
- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly ➔ No pipeline stall
  - **Taken**: Guessed wrongly ➔ Wrong instructions in the pipeline ➔ **Flush** successor instruction from the pipeline

Branch Prediction: **Correct Prediction**



Branch is known to be **not taken** in cycle 3
➔ no stall needed!

Branch Prediction: Wrong Prediction



Branch is known to be **taken** in cycle 3
➔"**`and`**" instruction should not be executed
➔ Flushed from pipeline

Exercise #6: **No Branch Prediction**

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding but **no** branch prediction?
  - Decision making moved to **ID** stage

- Total instructions = 1 + 10×2 + 1 = 22
- **Ideal** pipeline = 4 + 22×1 = 26 cycles

```
      addi $s0, $zero, 10
Loop: addi $s0, $s0, -1
      bne  $s0, $zero, Loop
      sub  $t0, $t1, $t2
```

Exercise #6: **No Branch Prediction**

	1	2	3	4	5	6	7	8	9	10	11
addi^1	IF	ID	EX	MEM	WB						
addi^2		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
addi^2						IF	ID	EX	MEM	WB	

Data dependency between (addi $s0, $s0, -1) and bne incurs 1 cycle of delay.
There are 10 iterations, hence 10 cycles of delay.
Every bne incurs a cycle of delay to execute the next instruction. There are 10 iterations, hence 10 cycles of delay.
Total number of cycles of delay = 20.
Total execution cycles = 26 + 20 = **46 cycles.**

Exercise #7: **Branch Prediction**

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding and branch prediction?

```
            addi  $s0, $zero, 10
Loop:       addi  $s0, $s0, -1
            bne   $s0, $zero, Loop
            sub   $t0, $t1, $t2
```

- Total instructions = $1 + 10 \times 2 + 1 = 22$
- **Ideal pipeline** = $4 + 22 \times 1 = 26$ cycles

Exercise #7: **Branch Prediction**

	1	2	3	4	5	6	7	8	9	10	11
addi^1	IF	ID	EX	MEM	WB						
addi^2		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
sub					IF						
addi^2						IF	ID	EX	MEM	WB	

Predict not taken.
The data dependency remains, hence 10 cycles of delay for 10 iterations.
In the first 9 iterations, the branch prediction is wrong, hence 1 cycle of delay.
In the last iteration, the branch prediction is correct, hence saving 1 cycle of delay.
Total number of cycles of delay = 19.
Total execution cycles = 26 + 19 = **45 cycles.**

Reduce Stalls: **Delayed Branch**

- **Observation:**
  - Branch outcome takes **X** number of cycles to be known
  - ➔ **X** cycles stall

- **Idea:**
  - Move **non-control dependent instructions** into the X slots following a branch
    - Known as the **branch-delay slot**
  - ➔ These instructions are executed **regardless of the branch outcome**

- In our MIPS processor:
  - Branch-Delay slot = **1** (with the early branch)

Delayed Branch: **Example**

Nondelayed branch

```
or   $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:
```

Delayed branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or   $8, $9, $10
xor $10, $1, $11

Exit:
```

- The "`or`" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome

  ➔ Same behavior as the original code!

Delayed Branch: **Observation**

- **Best case scenario**
  - There is an instruction **preceding the branch** which **can be moved** into delayed slot
    - Program correctness must be preserved!
- **Worst case scenario**
  - Such instruction cannot be found

  ➔ Add a no-op (**nop**) instruction in the branch-delay slot
- Re-ordering instructions is a common method of program optimization
  - Compiler must be smart enough to do this
  - Usually can find such an instruction at least 50% of the time

EXPLORATION

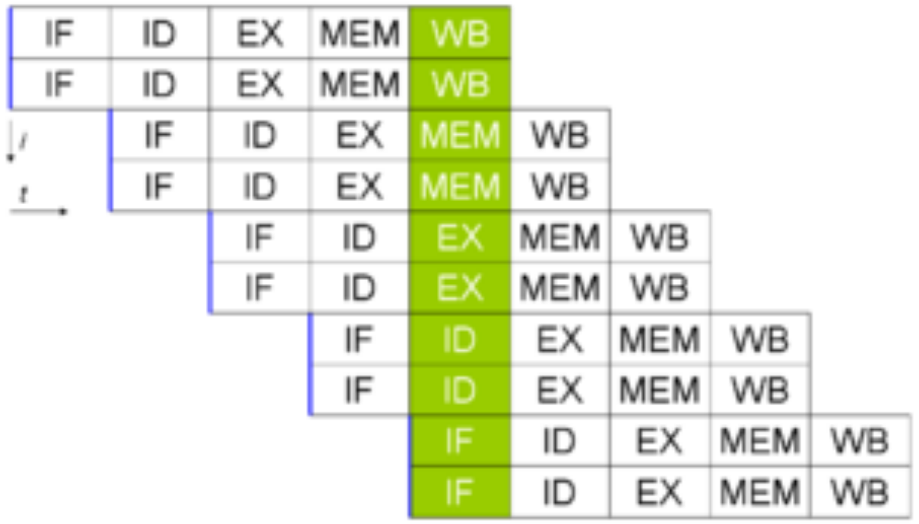
For your reference

Multiple Issue Processors(1/2)

- Multiple Issue processors
    - **Multiple instructions** in every pipeline stage
    - 4 washer, 4 dryer…
- **Static multiple issue**:
    - EPIC (Explicitly Parallel Instruction Computer) or VLIW (Very Long Instruction Word), e.g. IA64
    - Compiler specifies the set of instructions that execute together in a given clock cycle
    - Simple hardware, complex compiler
- **Dynamic multiple issue:**
    - Superscalar processor: Dominant design of modern processors
    - Hardware decides which instructions to execute together
    - Complex hardware, simpler compiler

Multiple Issue Processors(2/2)

- A 2-wide superscalar pipeline:
  - By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed.



Summary

- Pipelining is a fundamental concept in computer systems!
  - Multiple instructions in flight
  - Limited by length of the longest stage
  - Hazards create trouble by stalling pipeline

- Pentium 4 has 22 pipeline stages!

Reading Assignment

- 3rd edition
  - Sections 6.1 – 6.3
  - Sections 6.4 – 6.6 (data hazards and control hazards in details; read for interest; not in syllabus)
- 4th edition
  - Sections 4.5 – 4.6
  - Sections 4.7 – 4.8 (data hazards and control hazards in details; read for interest; not in syllabus)

Q&A