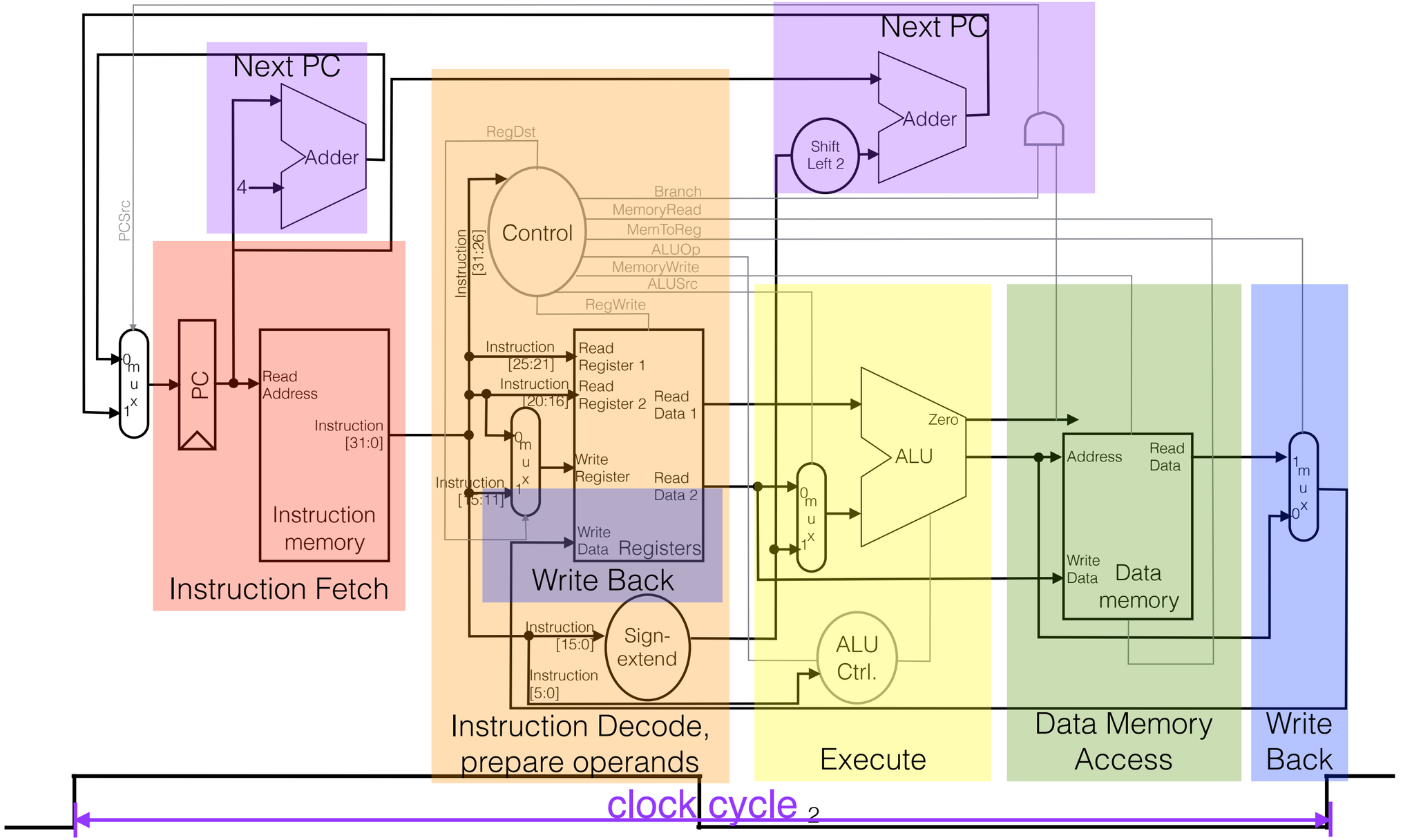
Processor Design (II)

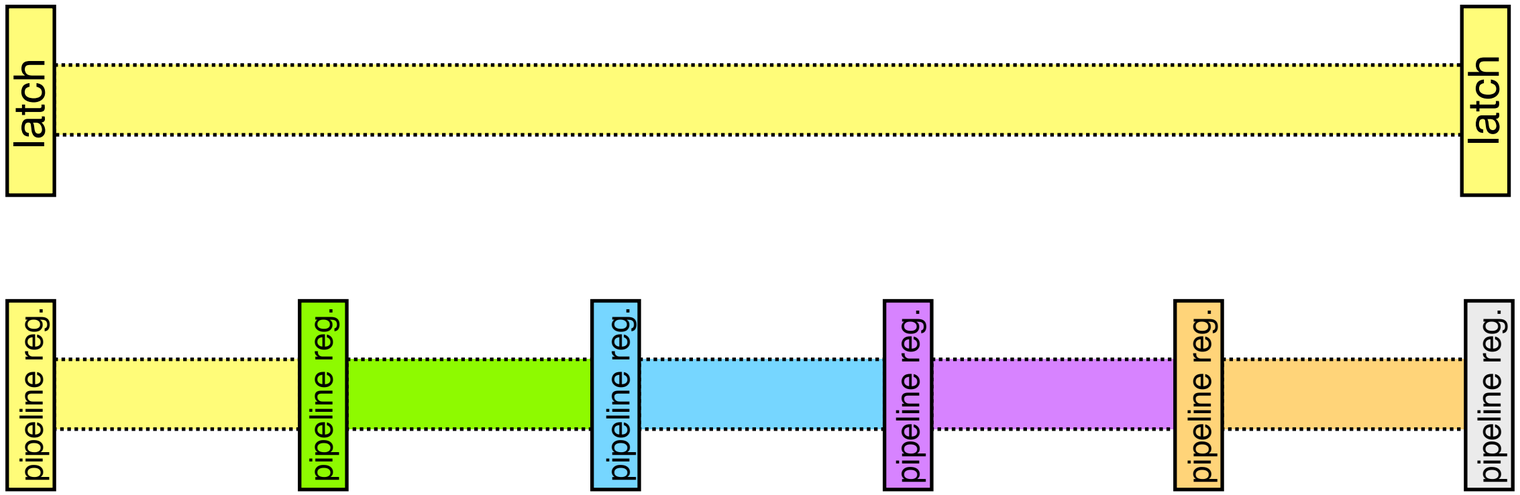
Hung-Wei Tseng

Single cycle processor

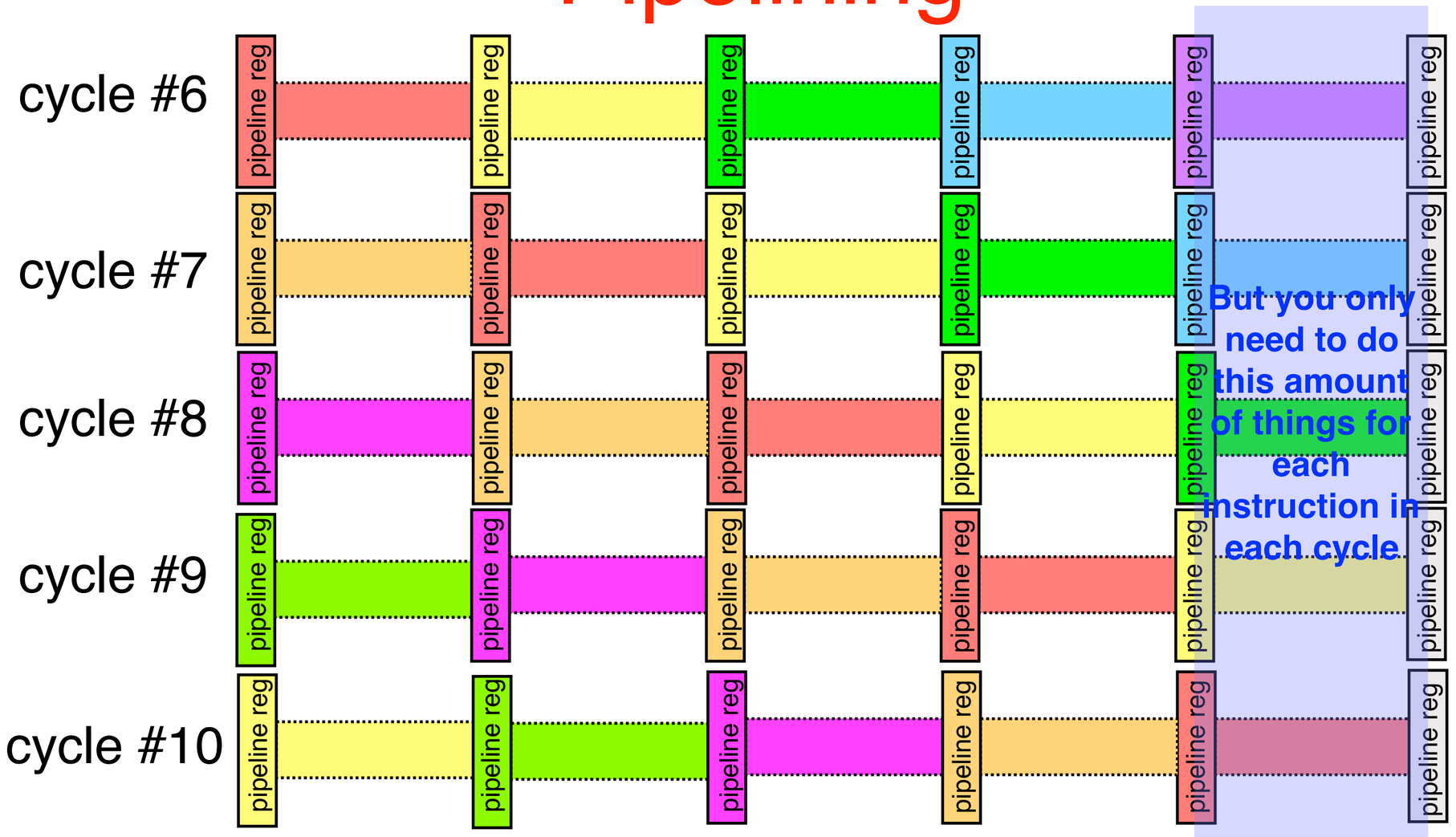


Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
  - These registers only changes their output at the triggered edge cycle
- Each stage can act on different instruction/data
- States/Control signals of instructions are hold in pipeline registers



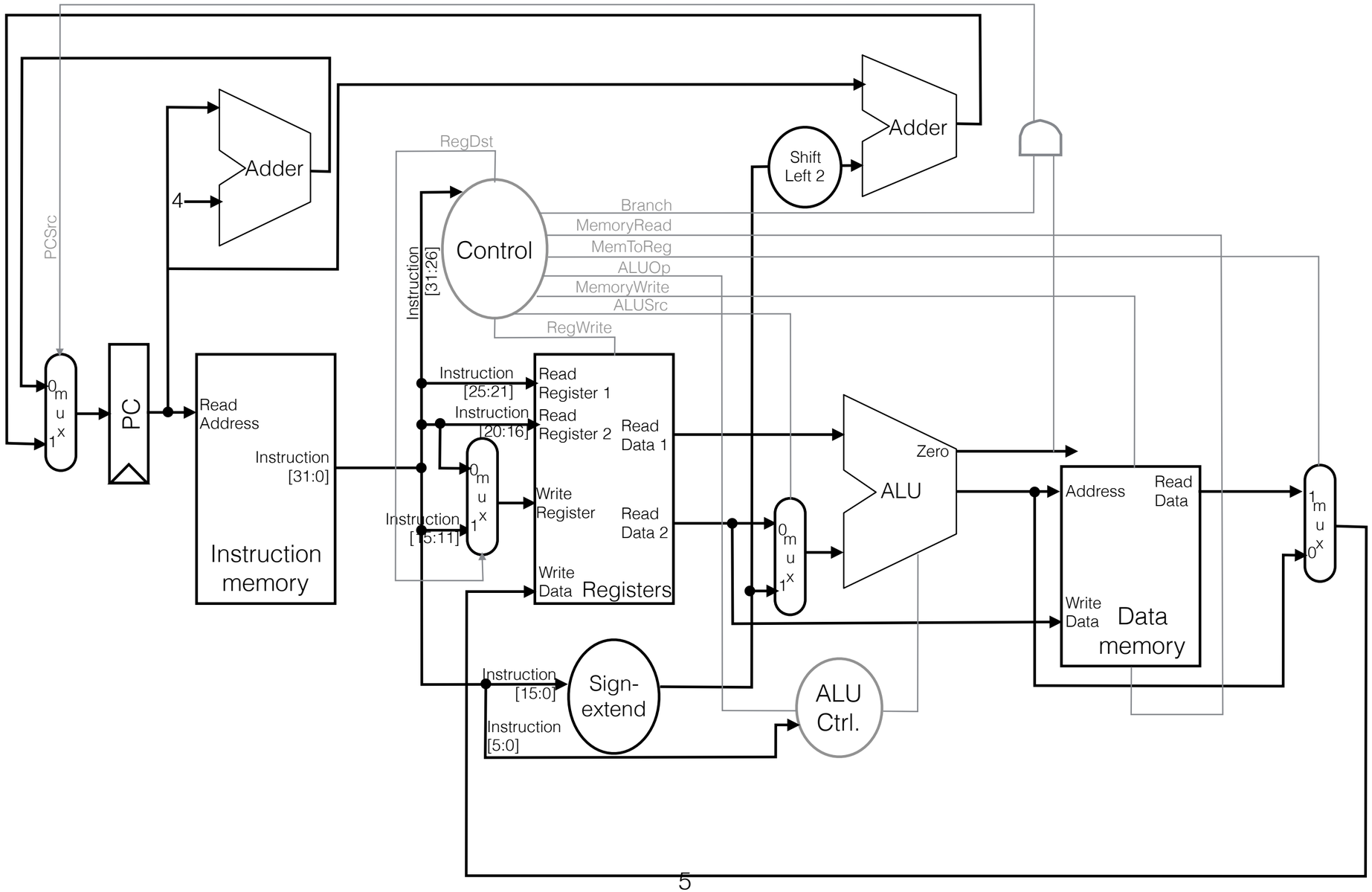
Pipelining



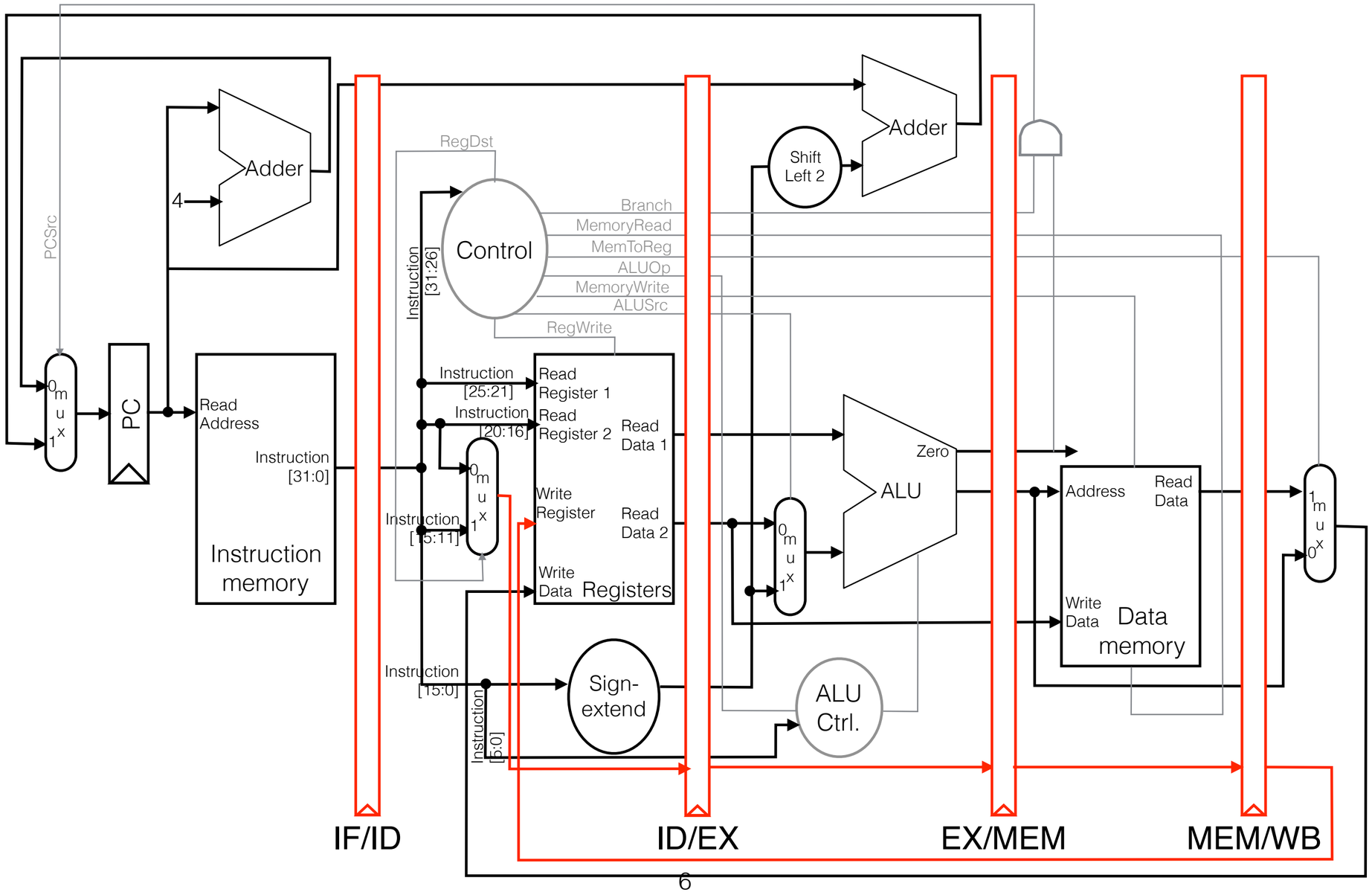
The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!

Single cycle processor



5-stage pipeline processor



Can we get the right result?

- Given the current 5-stage pipeline,

how many of the following MIPS code can work correctly (i.e. generate the same result as a single-cycle processor)?

	I	II	III	IV
a:	`add $1, $2, $3`	`add $1, $2, $3`	`add $1, $2, $3`	`add $1, $2, $3`
b:	`lw $4, 0($1)`	`lw $4, 0($5)`	`lw $4, 0($5)`	`lw $4, 0($5)`
c:	`sub $6, $7, $8`	`sub $6, $7, $8`	`bne $0, $7, L`	`sub $6, $7, $8`
d:	`sub $9,$10,$11`	`sub $9, $1, $10`	`sub $9,$10,$11`	`sub $9,$10,$11`
e:	`sw $1, 0($12)`	`sw $11, 0($12)`	`sw $1, 0($12)`	`sw $1, 0($12)`
	b cannot get $1 produced by a before WB	both a and d are accessing $1 at 5th cycle	We don't know if d & e will be executed or not until c finishes	

A. 0
B. 1
C. 2
D. 3
E. 4

Pipeline hazards

- Even though we perfectly divide pipeline stages, it's still hard to achieve CPI == 1.
- Pipeline hazards:
  - Structural hazard
    - The hardware does not allow two pipeline stages to work concurrently
  - Data hazard
    - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  - Control hazard
    - The processor is not clear about what's the next instruction to fetch

Can we get the right result?

- Given the current 5-stage pipeline,

how many of the following MIPS code can work correctly?

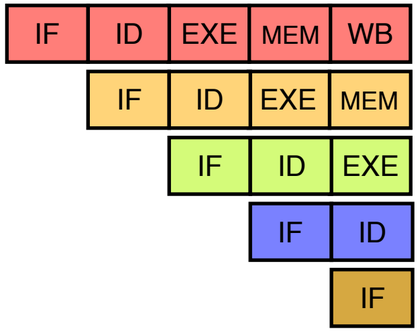
	I	II	III	IV
a:	`add $1, $2, $3`	`add $1, $2, $3`	`add $1, $2, $3`	`add $1, $2, $3`
b:	`lw $4, 0($1)`	`lw $4, 0($5)`	`lw $4, 0($5)`	`lw $4, 0($5)`
c:	`sub $6, $7, $8`	`sub $6, $7, $8`	`bne $0, $7, L`	`sub $6, $7, $8`
d:	`sub $9,$10,$11`	`sub $9, $1, $10`	`sub $9,$10,$11`	`sub $9,$10,$11`
e:	`sw $1, 0($12)`	`sw $11, 0($12)`	`sw $1, 0($12)`	`sw $1, 0($12)`
	b cannot get $1 produced by a before WB	both a and d are accessing $1 at 5th cycle	We don't know if d & e will be executed or not	
	Data hazard	Structural hazard	Control hazard	

Structural hazard

Structural hazard

- What just happened here is problematic if we change one of the source register of the 2nd sub instruction?

```
add $1, $2, $3
 lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw  $1, 0($12)
```

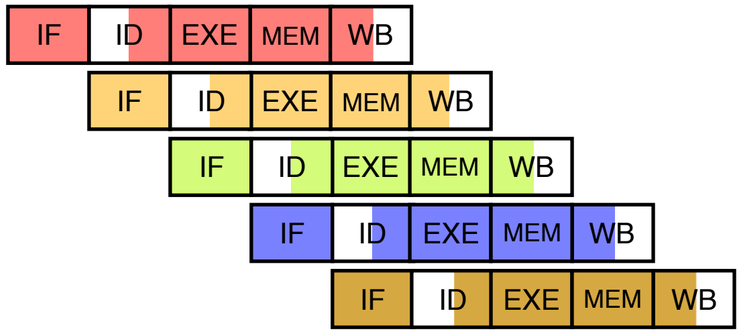


A. The register file is trying to read and write at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it's produced
D. Both A and B
E. Both A and C

Structural hazard

- The hardware cannot support the combination of instructions that we want to execute at the same cycle
- The original pipeline incurs structural hazard when two instructions competing the same register.
- Solution: write early, read late
  - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  - This leaves enough time for outputs to settle for reads
  - The revised register file is the default one from now!

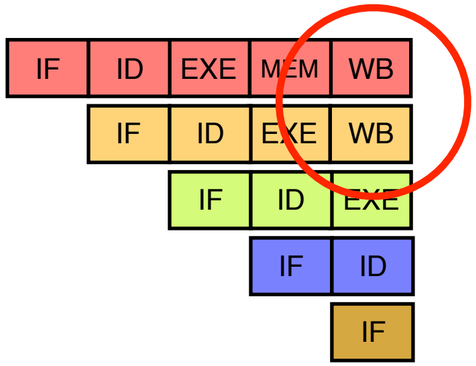
```
add $1, $2, $3
 lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw  $1, 0($12)
```



Structural hazard

- What pair of instructions will be problematic if we allow R-type instructions to skip the “MEM” stage?

```
a: lw  $1, 0($2)
b: add $3, $4, $5
c: sub $6, $7, $8
d: sub $9,$10,$11
e: sw  $1, 0($12)
```



A. a & b
B. a & c
C. b & e
D. c & e
E. None

Structural hazard

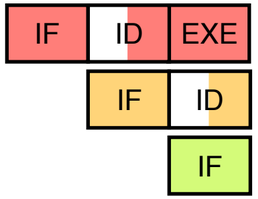
- The design of hardware causes structural hazard
- We need to modify the hardware design to avoid structural hazard

Data hazard

Data hazard

- What just happened here is problematic for the following instructions in our current pipeline?

```
add $1, $2, $3
 lw  $4, 0($1)
sub $6, $7, $8
sub $9,$10, $1
sw  $1, 0($12)
```



A. The register file and memory are both active at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it's produced
D. Both A and B
E. Both A and C

Data hazard

- When an instruction in the pipeline needs a value that is not available
- Data dependences
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline

Data dependences

- How many pairs of data dependences are there in the following code?

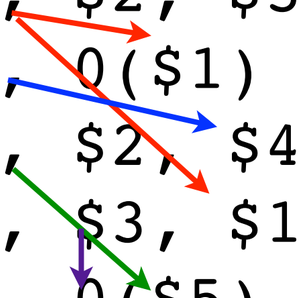
```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

A. 1
B. 2
C. 3
D. 4
E. 5

Data dependences

- How many pairs of data dependences are there in the following code?

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



A. 1

B. 2

C. 3

D. 4

E. 5

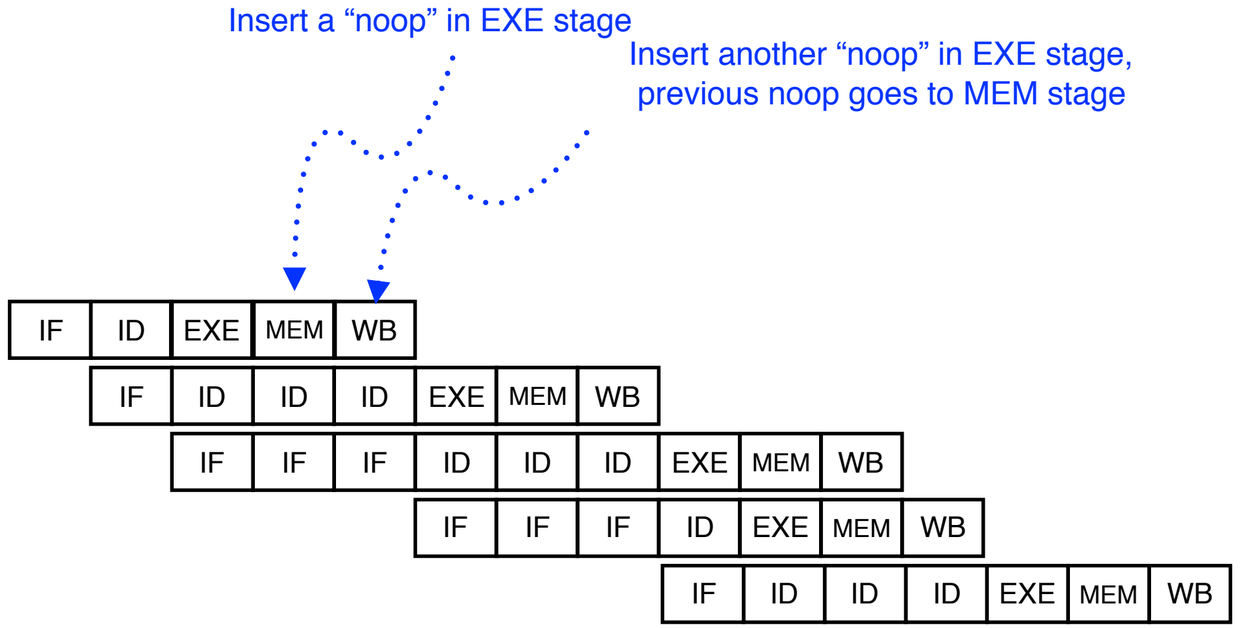
No every “data dependency” will lead to “data hazards”.

Sol. of data hazard I: Stall

- When the source operand of an instruction is not ready, stall the pipeline
  - Suspend the instruction and the following instruction
  - Allow the previous instructions to proceed
  - This introduces a pipeline bubble: a bubble does nothing, propagate through the pipeline like a nop instruction
- How to stall the pipeline?
  - Disable the PC update
  - Disable the pipeline registers on the earlier pipeline stages
  - When the stall is over, re-enable the pipeline registers, PC updates

Performance of stall

```
add $1, $2, $3
 lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
 sw  $1, 0($5)
```



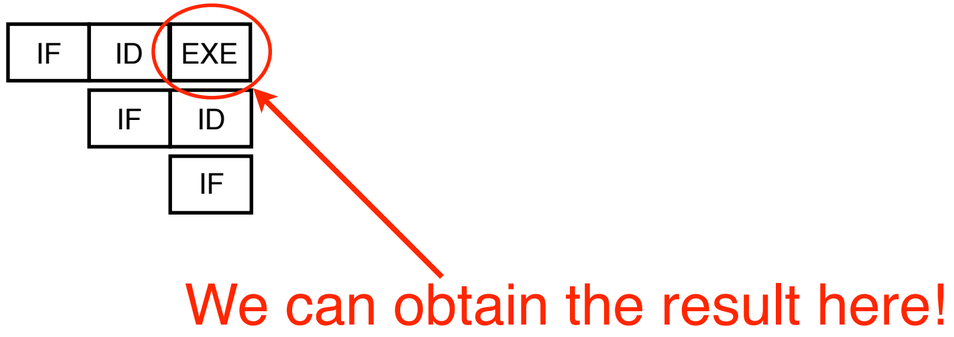
15 cycles! CPI == 3

(If there is no stall, CPI should be just 1!)

Sol. of data hazard II: Forwarding

- The result is available after EXE and MEM stage, but publicized in WB!
- The data is already there, we should use it right away!
- Also called bypassing

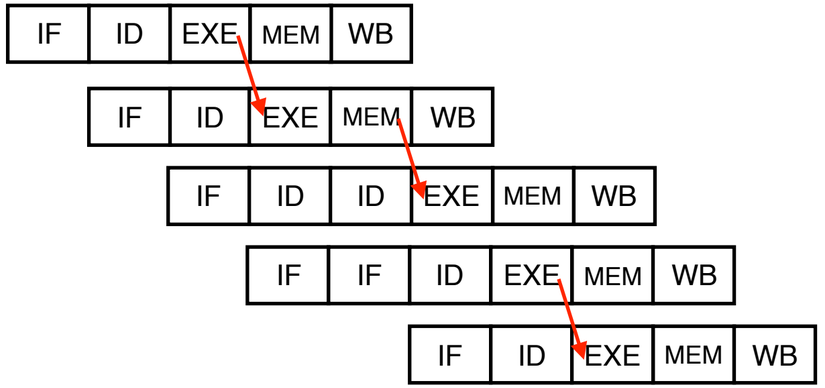
```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



Sol. of data hazard II: Forwarding

- Take the values, where ever they are!

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

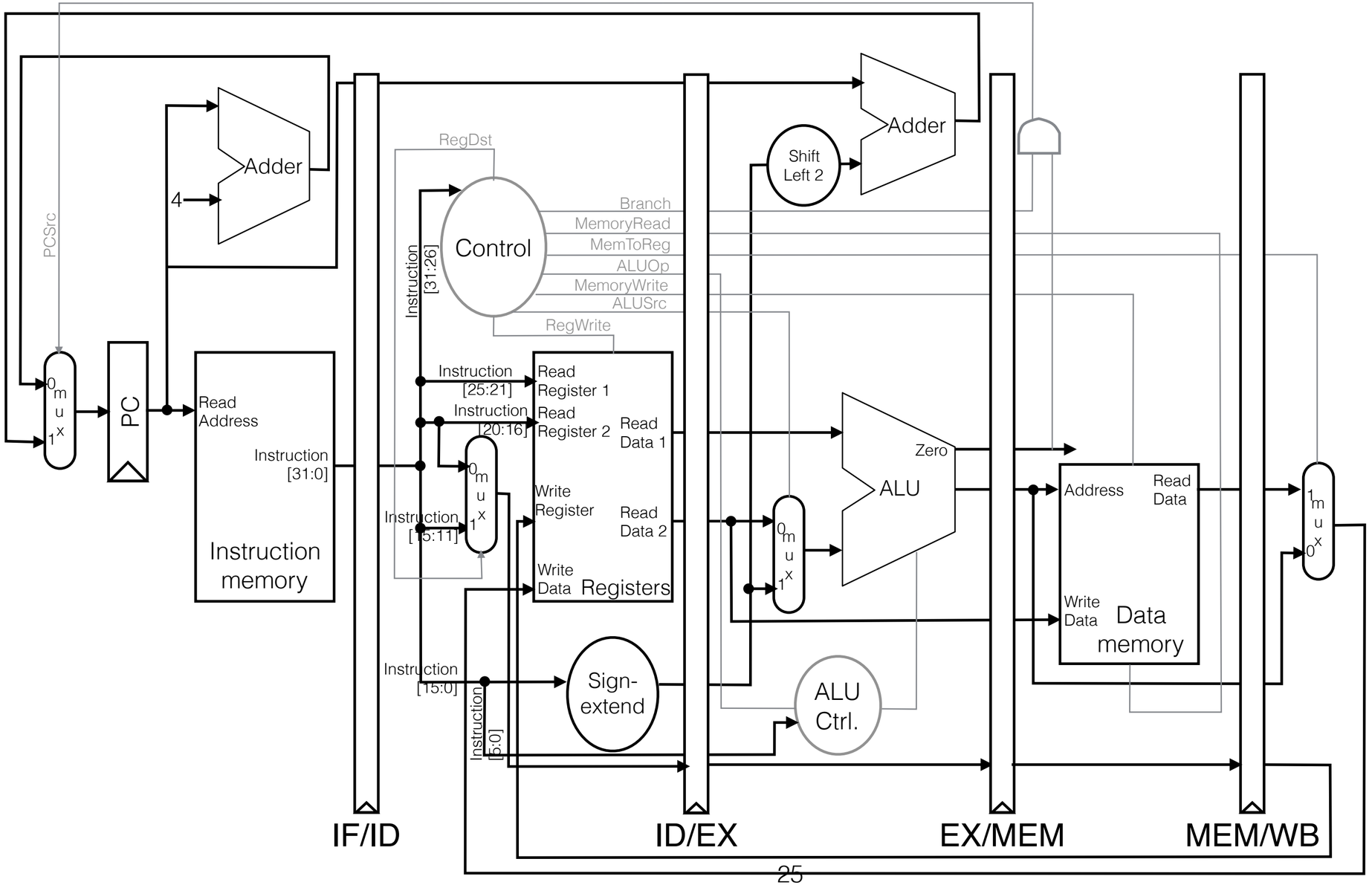


10 cycles! CPI == 2 (Not optimal, but much better!)

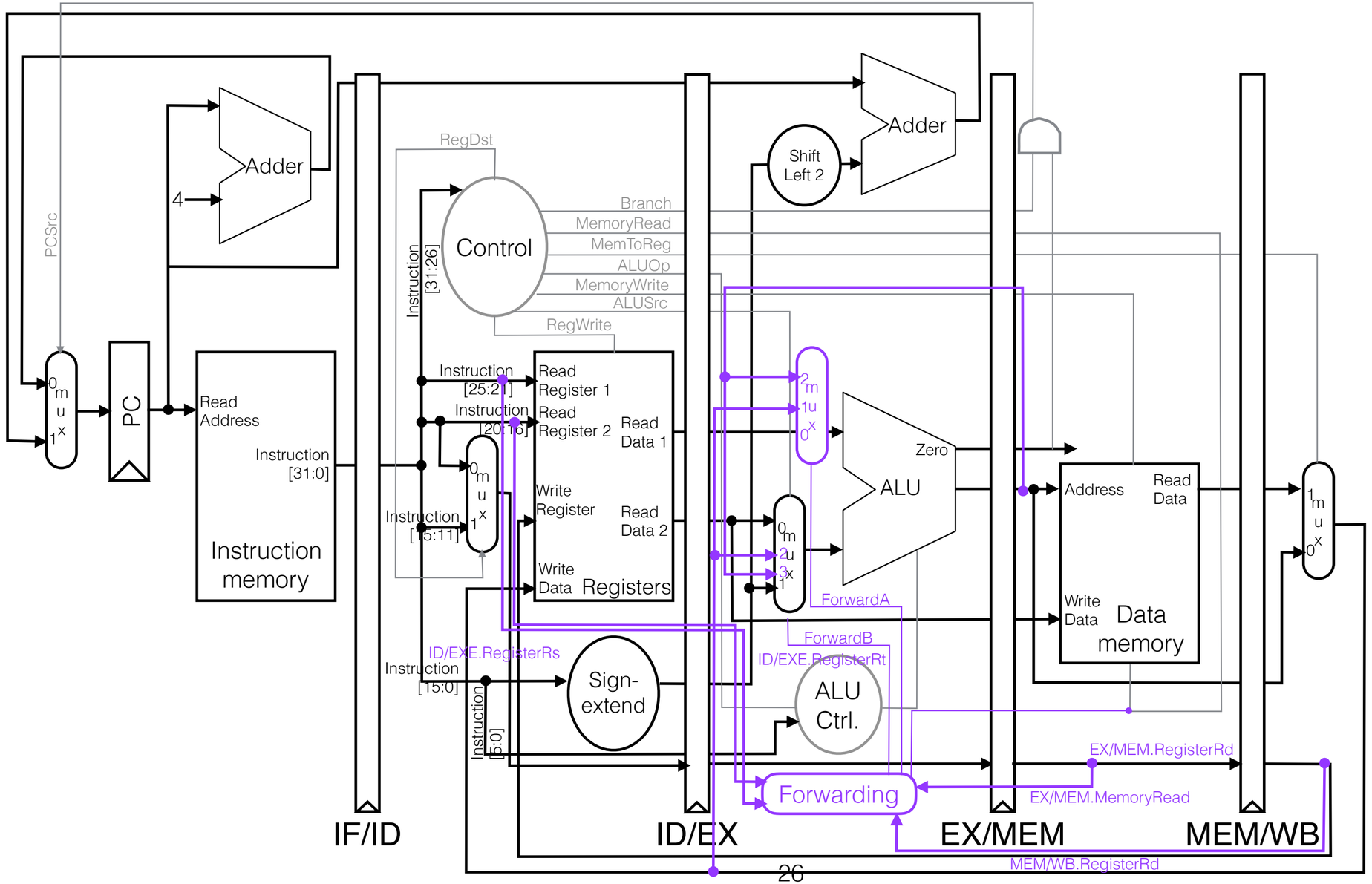
When can/should we forward data?

- If the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file

5-stage pipeline processor

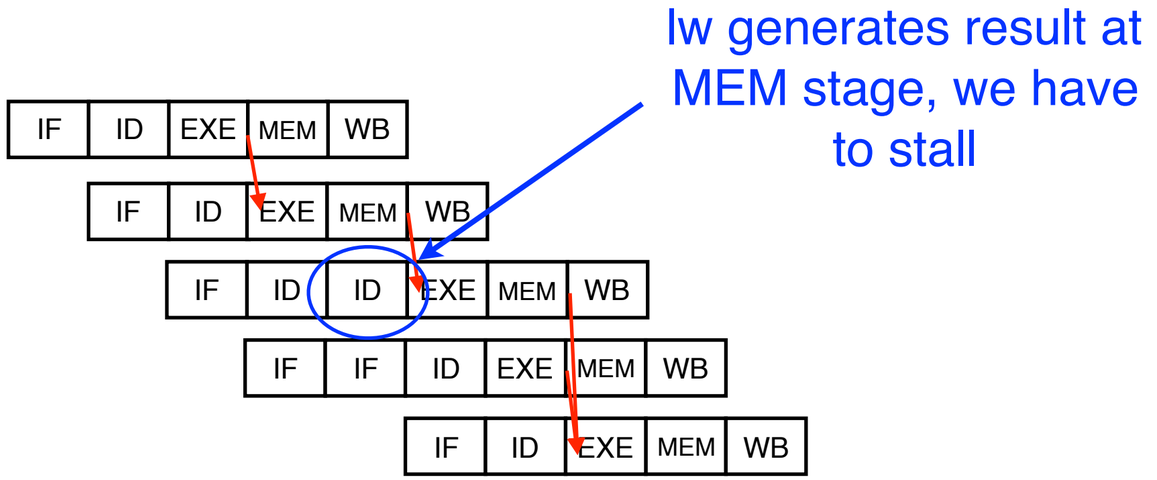


5-stage pipeline processor with forwarding



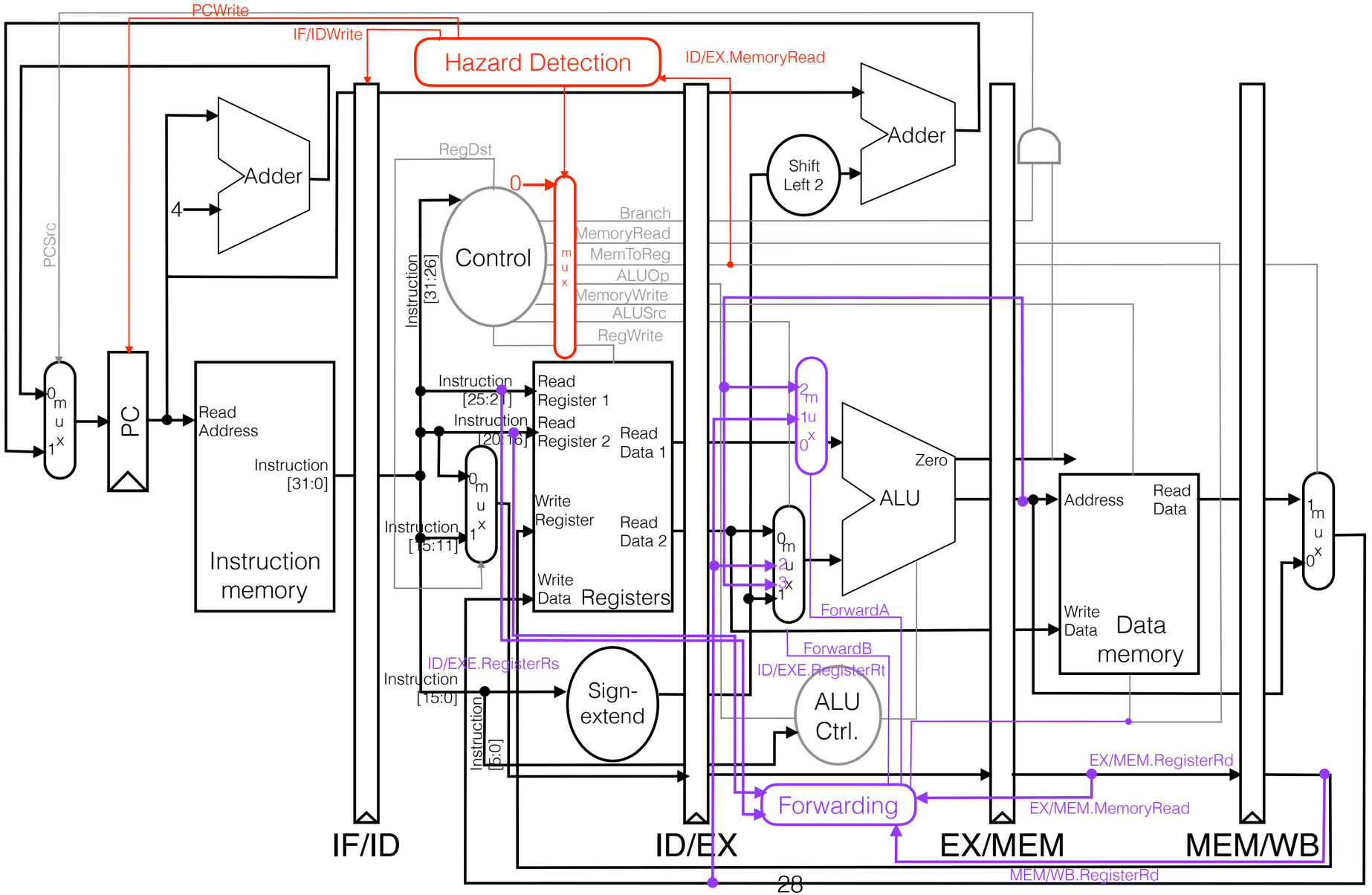
There is still a case that we have to stall...

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!

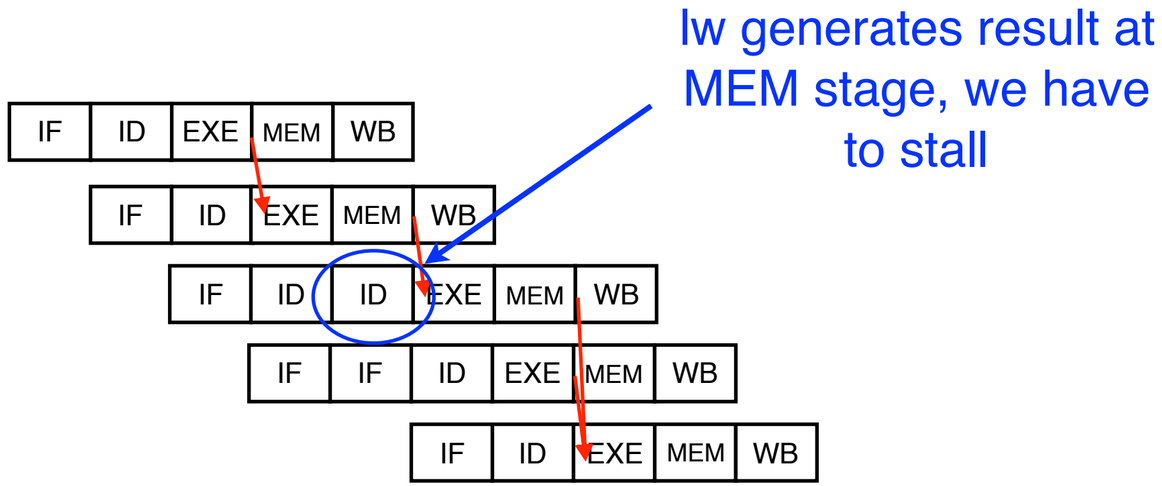
5-stage pipeline processor



There is still a case that we have to stall...

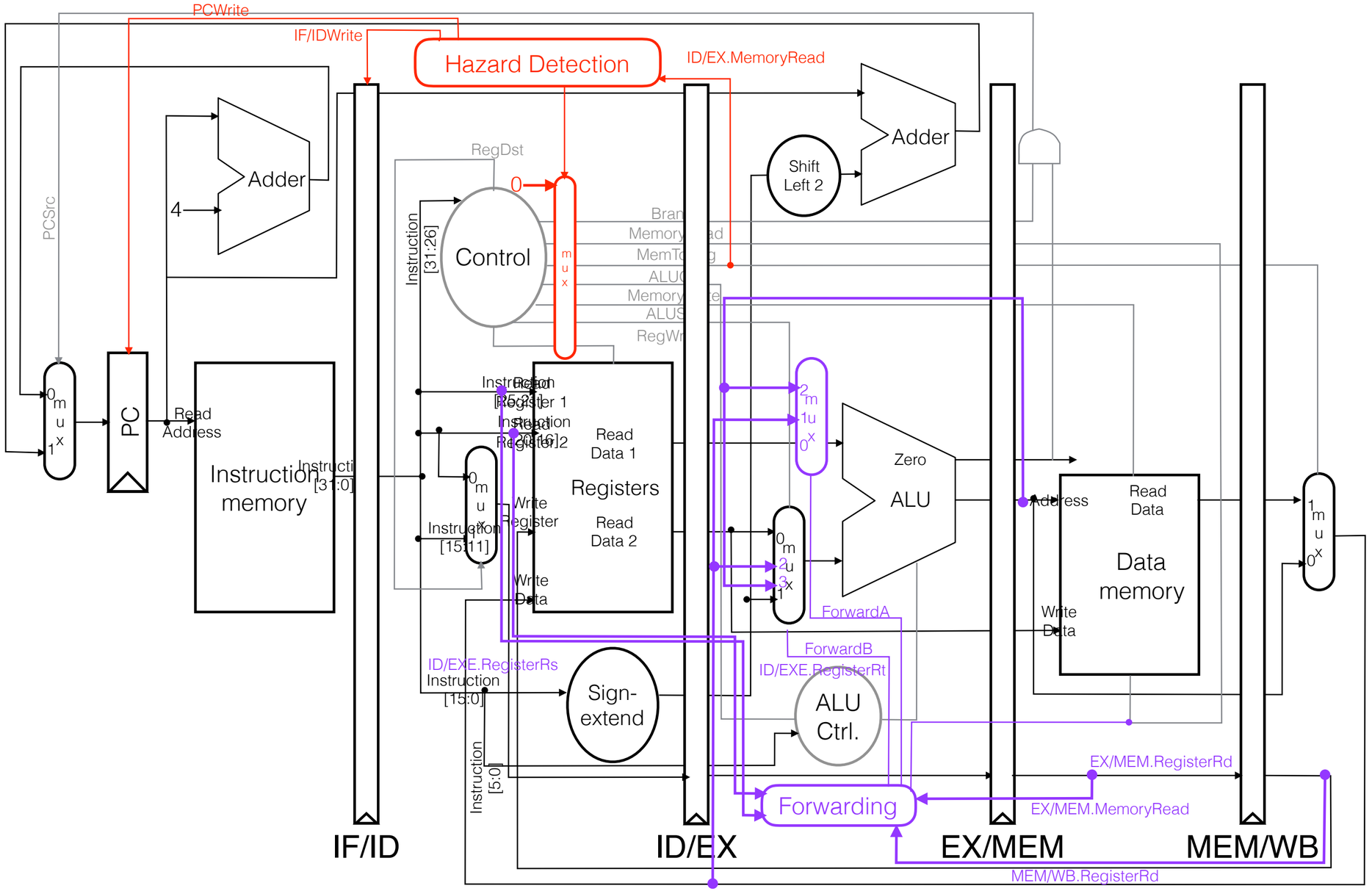
- Revisit the following code:

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!

5-stage pipeline processor



The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

① `add $1, $2, $3`
② `lw  $4, 0($1)`
③ `sub $5, $2, $4`
④ `sub $1, $3, $1`
⑤ `sw  $1, 0($5)`

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. (3) & (5)

Demo

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

Control hazard

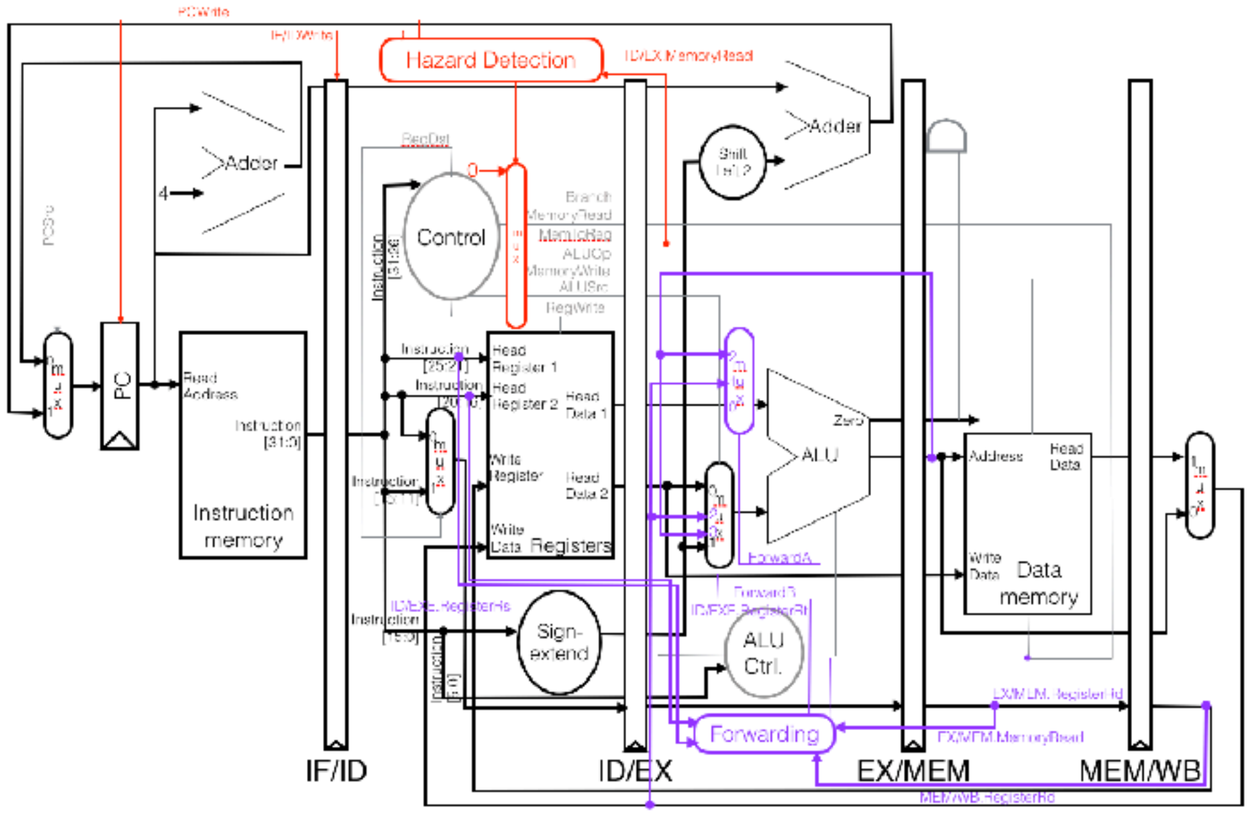
Control hazard

- Consider the following code and the pipeline we designed

```
LOOP: lw   $t3, 0($s0)
      addi $t0, $t0, 1
      add  $v0, $v0, $t3
      addi $s0, $s0, 4
      bne  $t1, $t0, LOOP
      sw   $v0, 0($s1)
```

How many cycles does the processor need to stall before we figure out the next instruction after "`bne`"?

A. 0
B. 1
C. 2
D. 3
E. 4



Why do we need to stall for branch instructions

- How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  - ① The target address when branch is taken is not available for instruction fetch stage of the next cycle
  - ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  - ③ The branch outcome cannot be decided until the comparison result of ALU is out
  - ④ The next instruction needs the branch instruction to write back its result

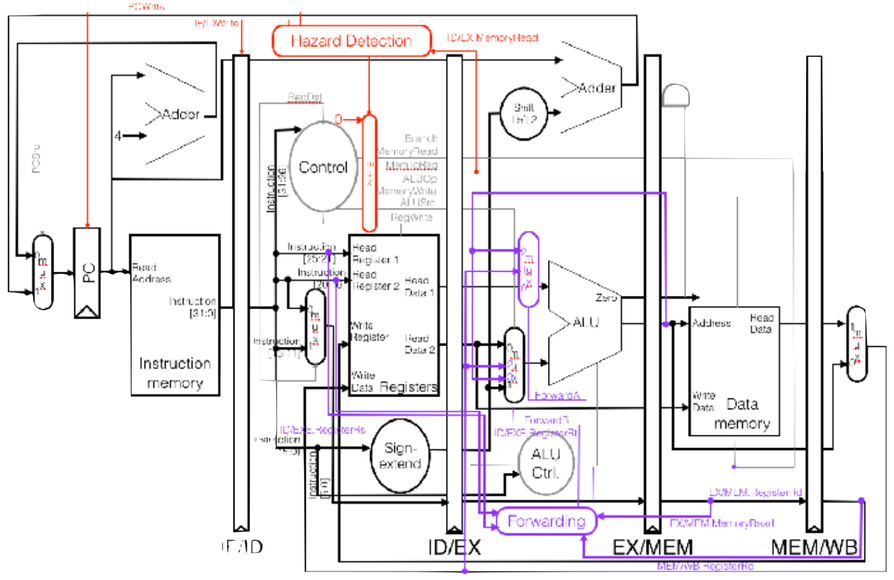
A. 0

B. 1

C. 2

D. 3

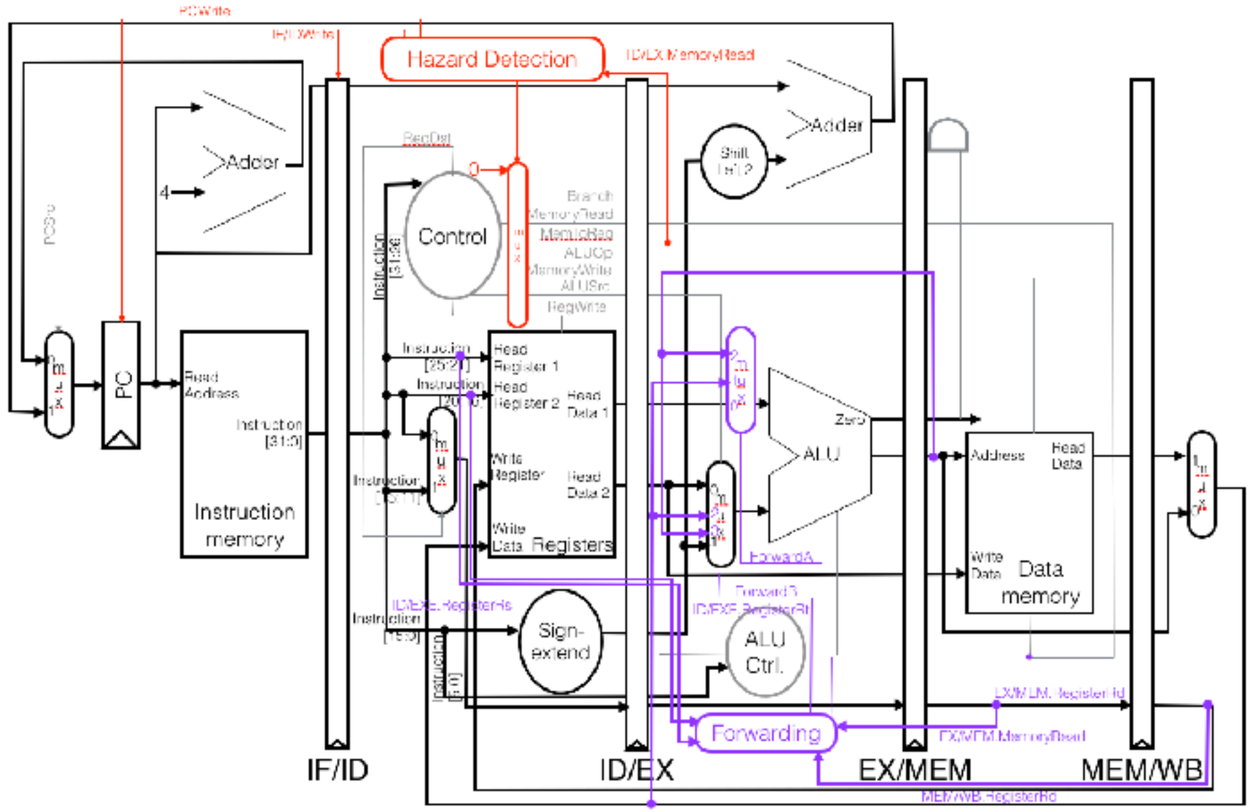
E. 4



Control hazard

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards, what's the average CPI if we execute this program on the 5-stage MIPS pipeline?

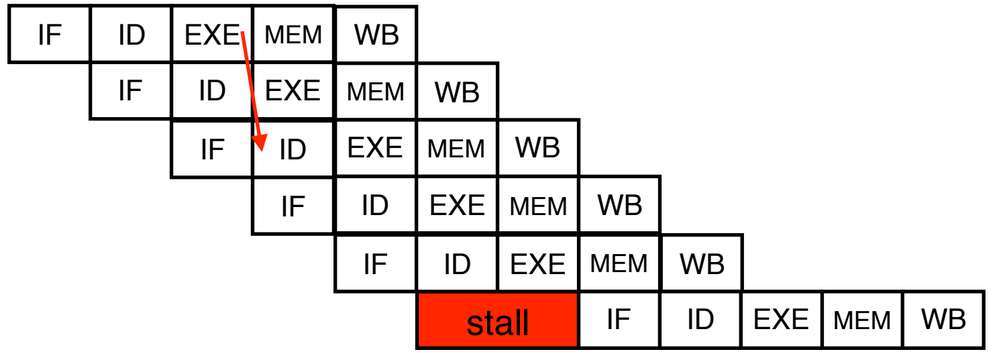
A. 1
B. 1.2
C. 1.4
D. 1.6
E. 1.8



Control hazard

- The processor cannot determine the next PC to fetch

```
LOOP: lw   $t3, 0($s0)
      addi $t0, $t0, 1
      add  $v0, $v0, $t3
      addi $s0, $s0, 4
      bne  $t1, $t0, LOOP
      sw   $v0, 0($s1)
```



7 cycles per loop