Processor Design – Pipelined Processor

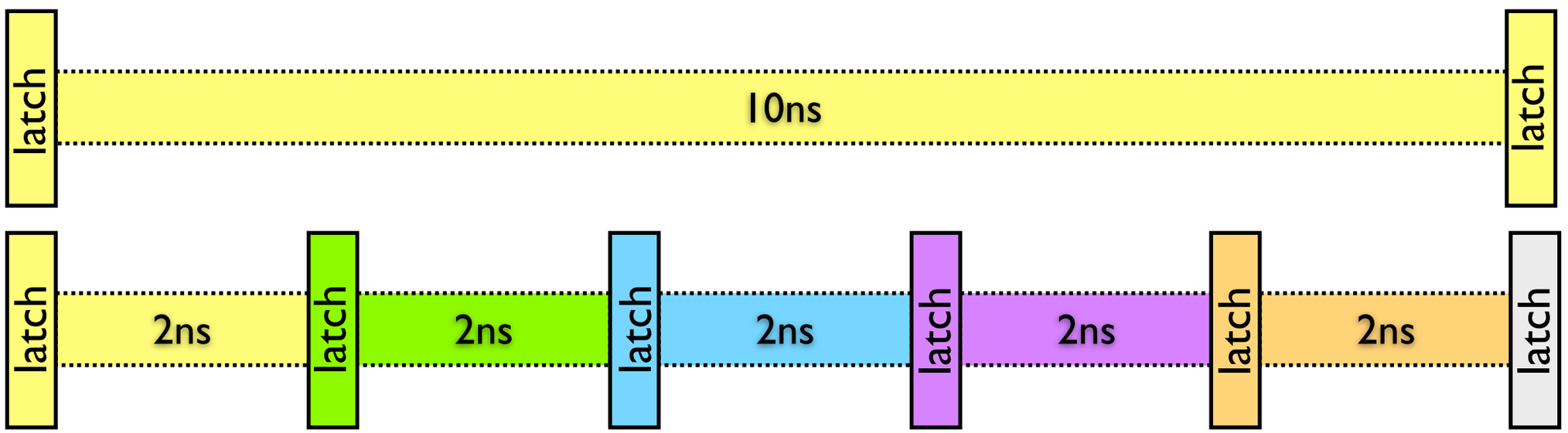
Hung-Wei Tseng

Drawbacks of a single-cycle processor

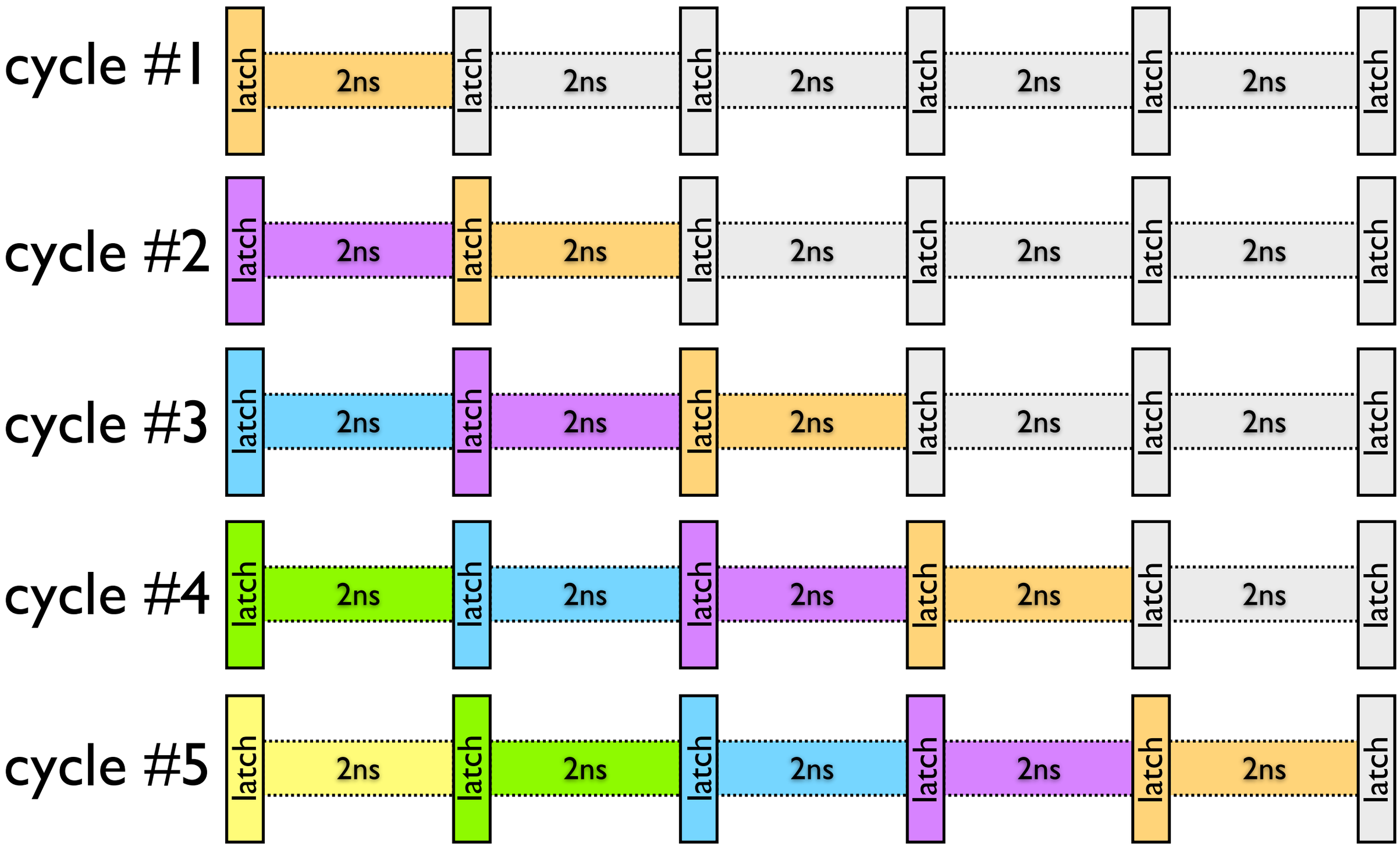
- The cycle time is determined by the longest instruction
  - Could be very long, thinking about fetch data from DRAM
- Hardware is mostly idle

Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
- Each stage can act on different instruction/data
- States/Control Signals of instructions are hold in pipeline registers (latches)



Pipelining

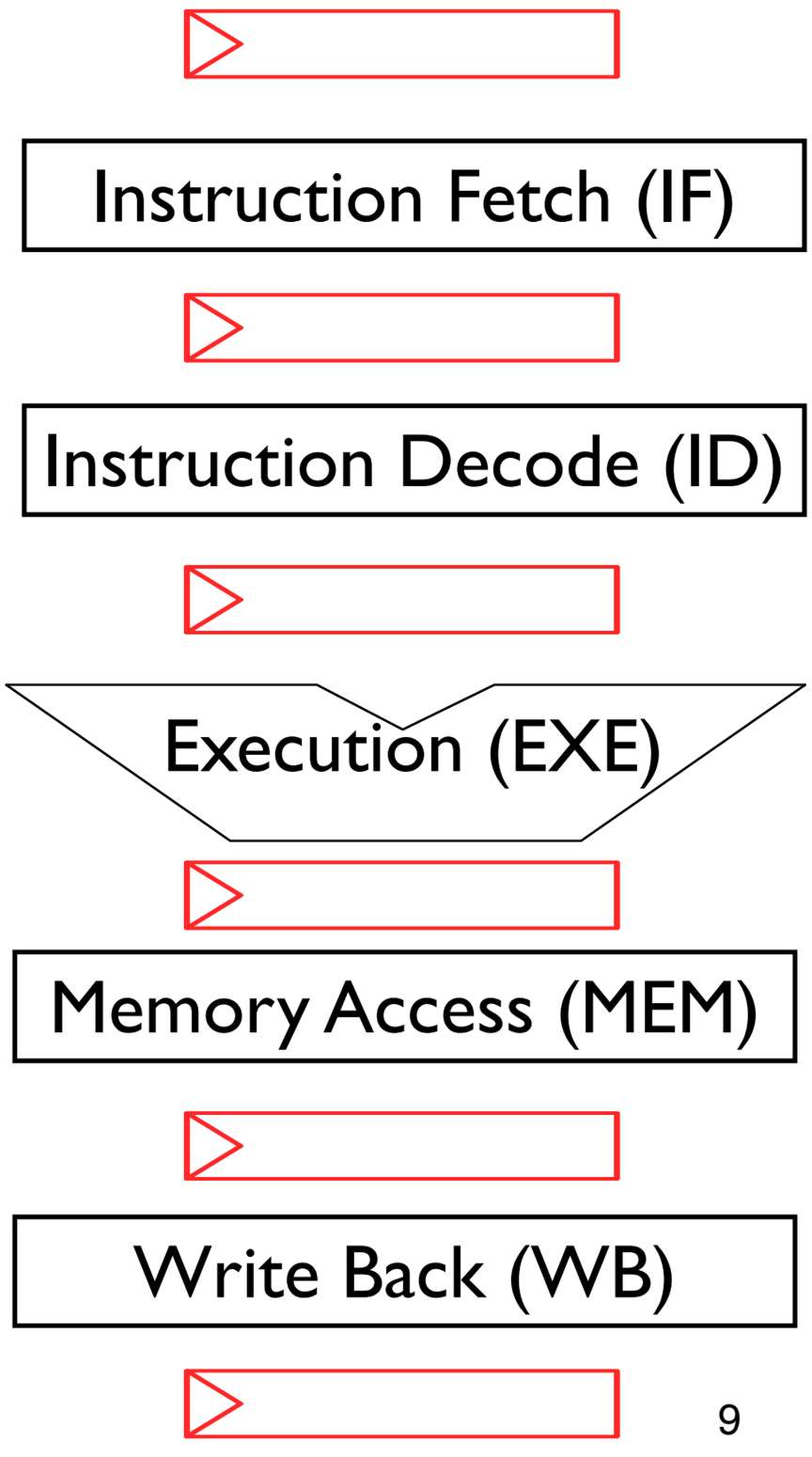

cycle #1
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
cycle #2
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
cycle #3
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
cycle #4
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
cycle #5
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch
2ns
latch

Cycle time of a pipeline processor

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance
- Ideally, all paths are about the same length

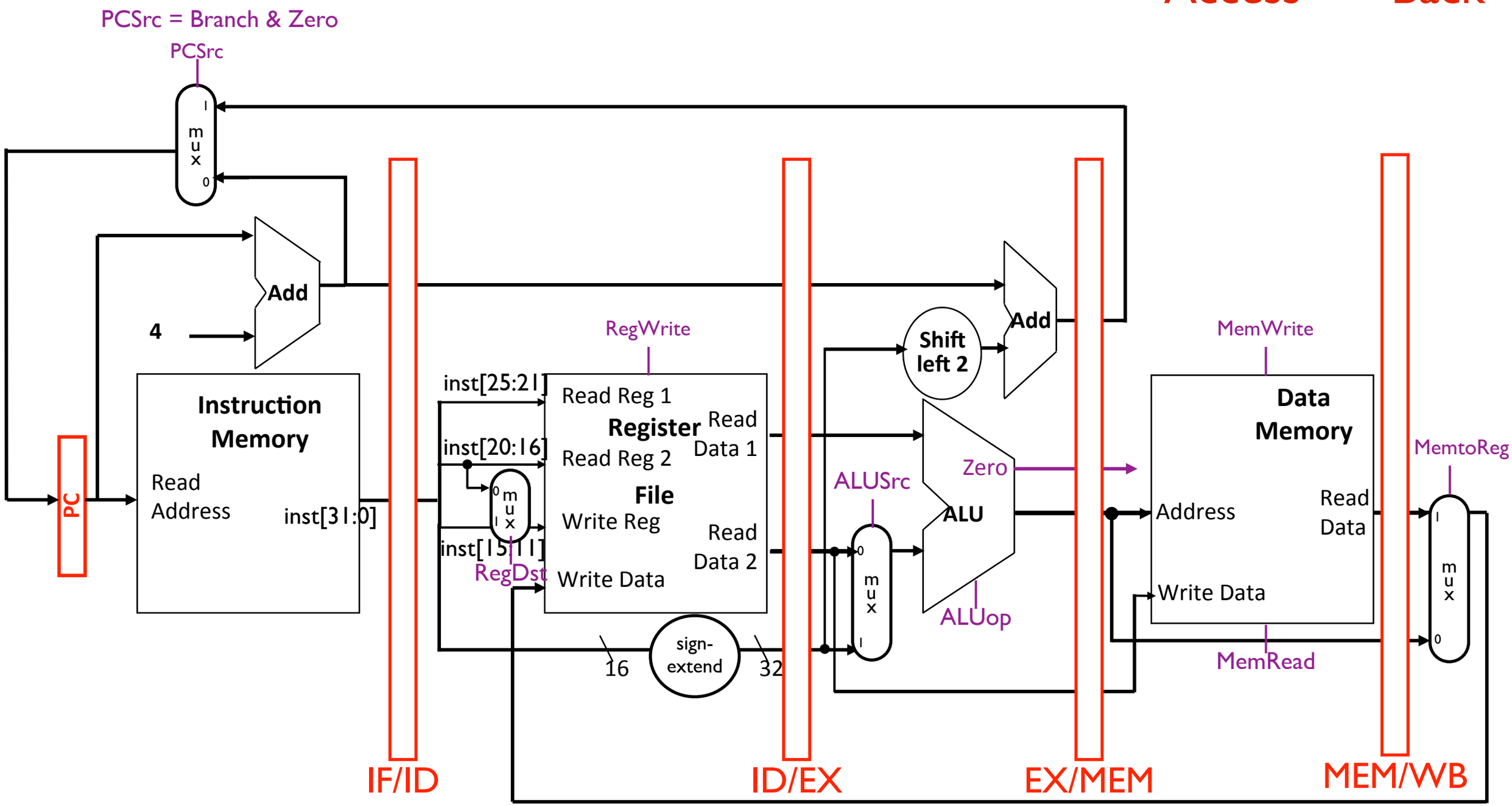
Pipeline a MIPS processor

- Instruction Fetch
  - Read the instruction
- Decode
  - Figure out the incoming instruction?
  - Fetch the operands from the register file
- Execution: ALU
  - Perform ALU functions
- Memory access
  - Read/write data memory
- Write back results to registers
  - Write to register file



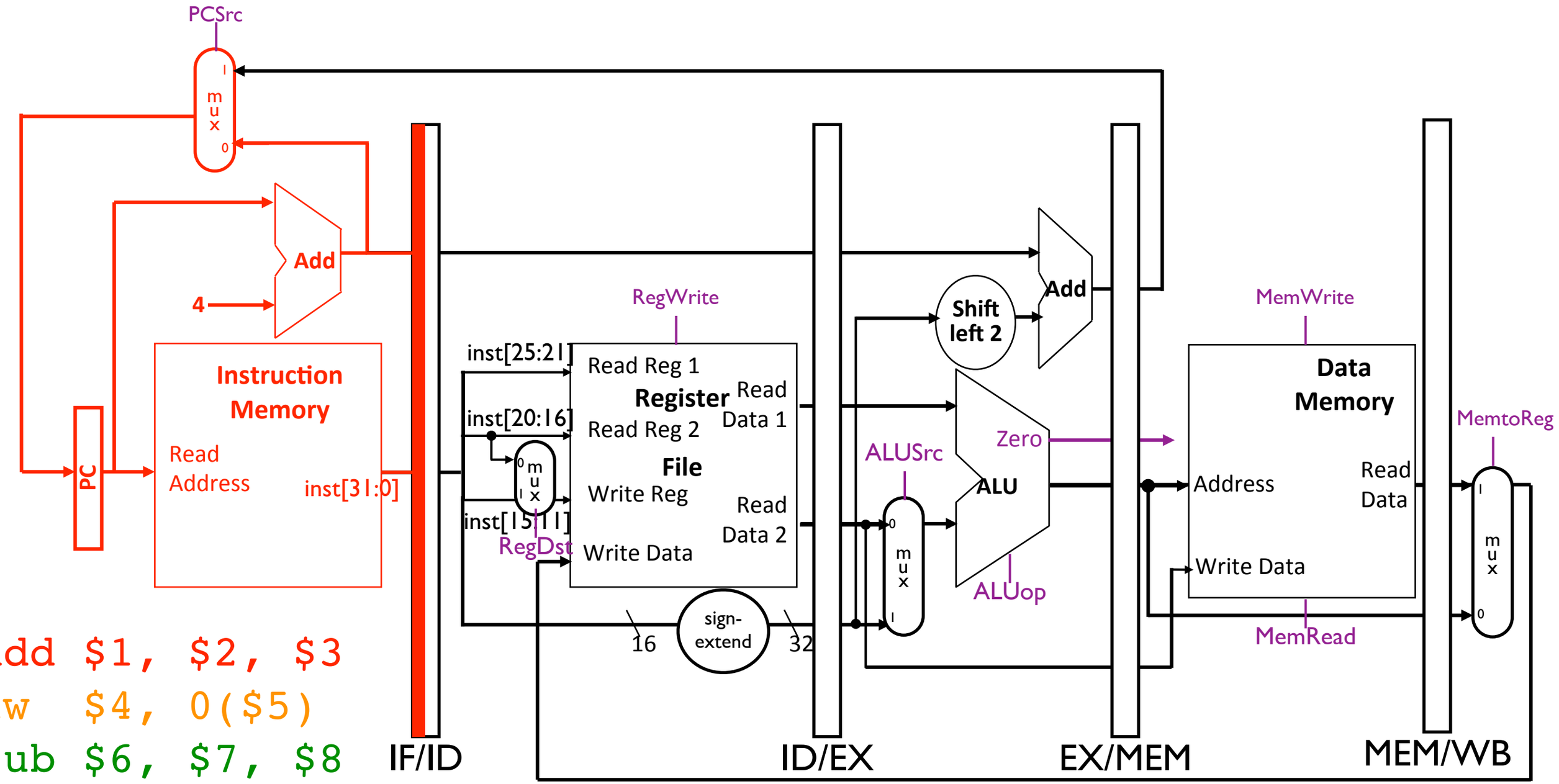
Pipelined datapath

Instruction Fetch | Instruction Decode | Execution | Memory Access | Write Back



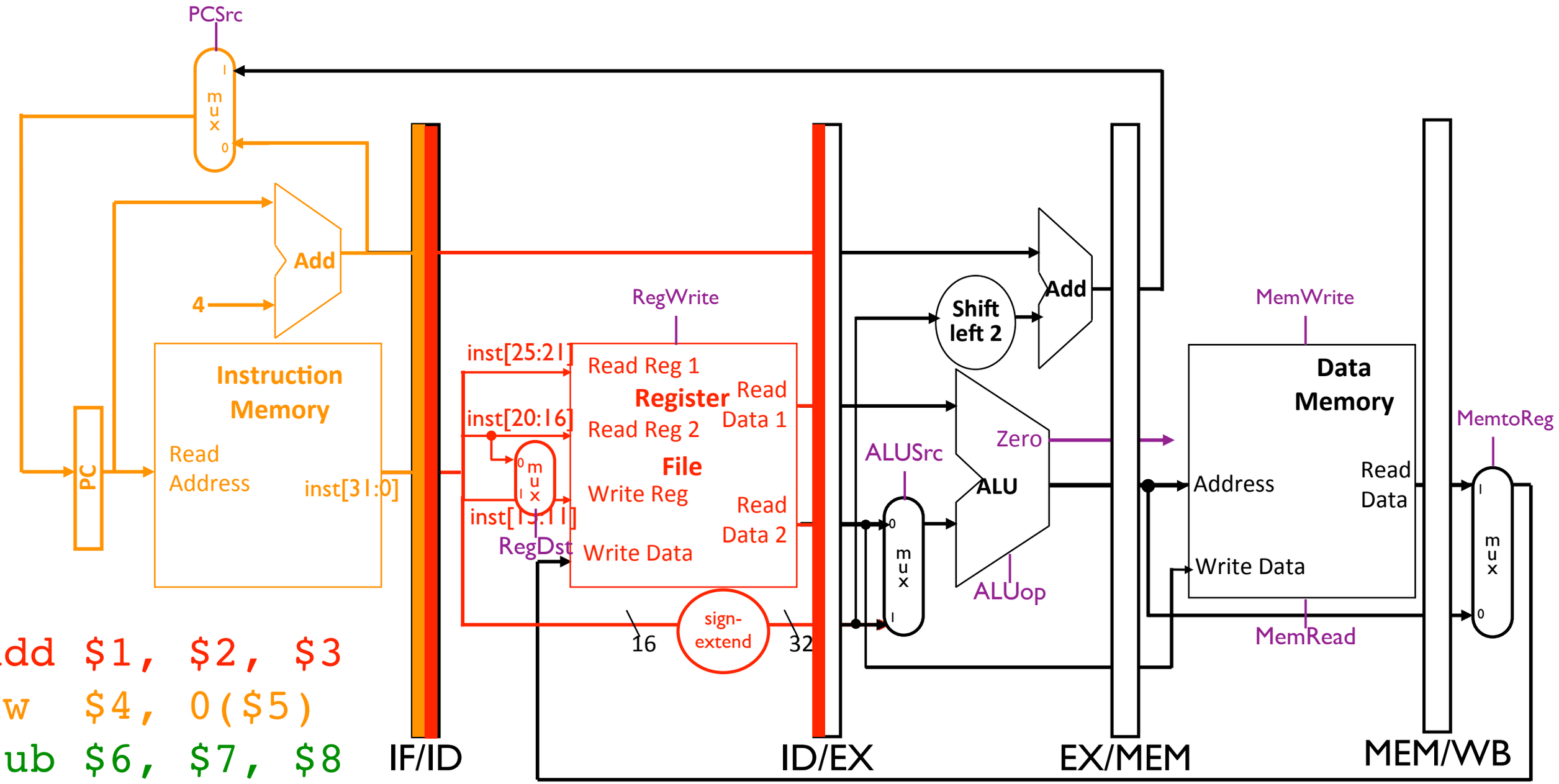
Will this work?

Pipelined datapath



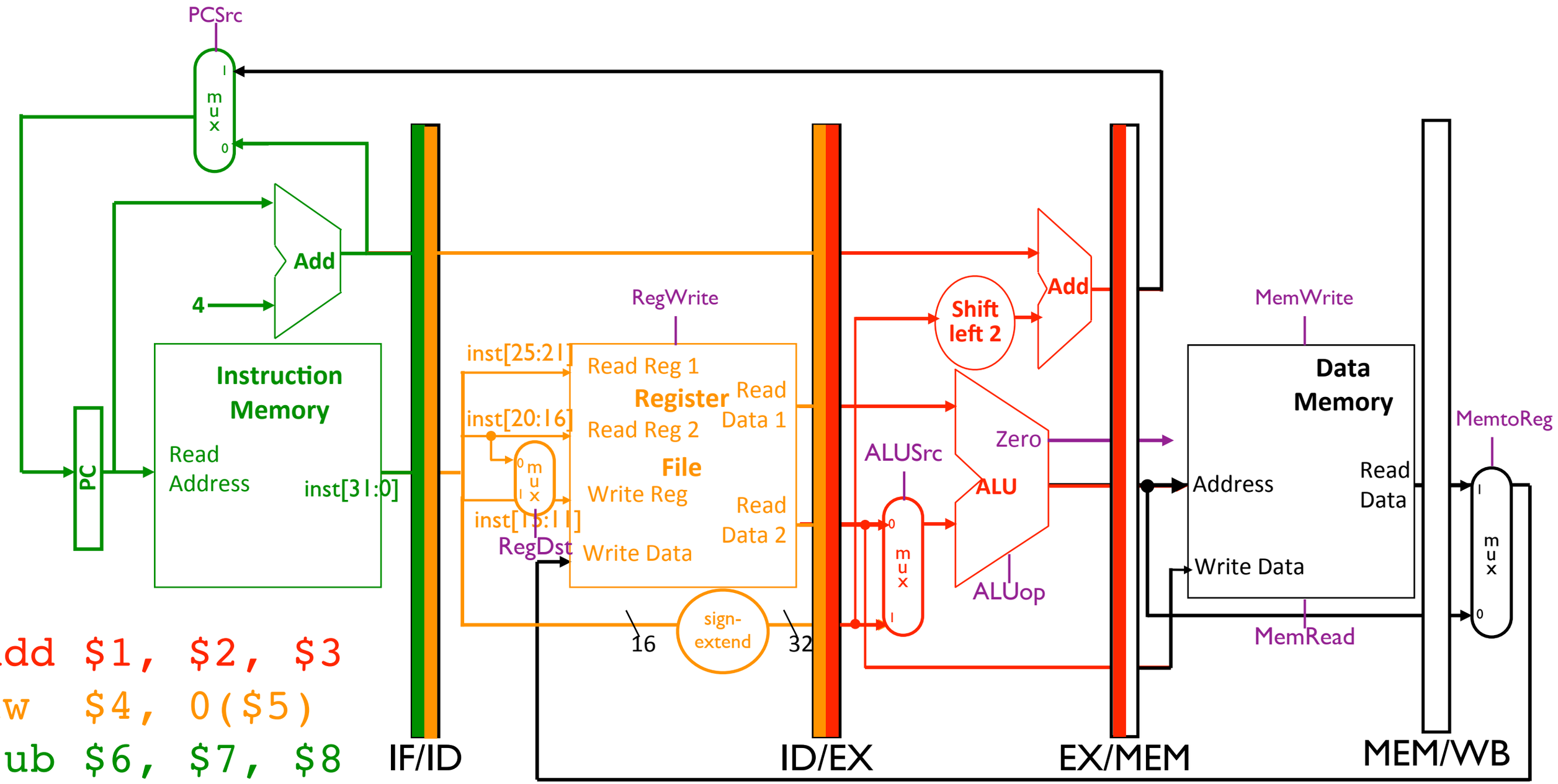

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```

Pipelined datapath



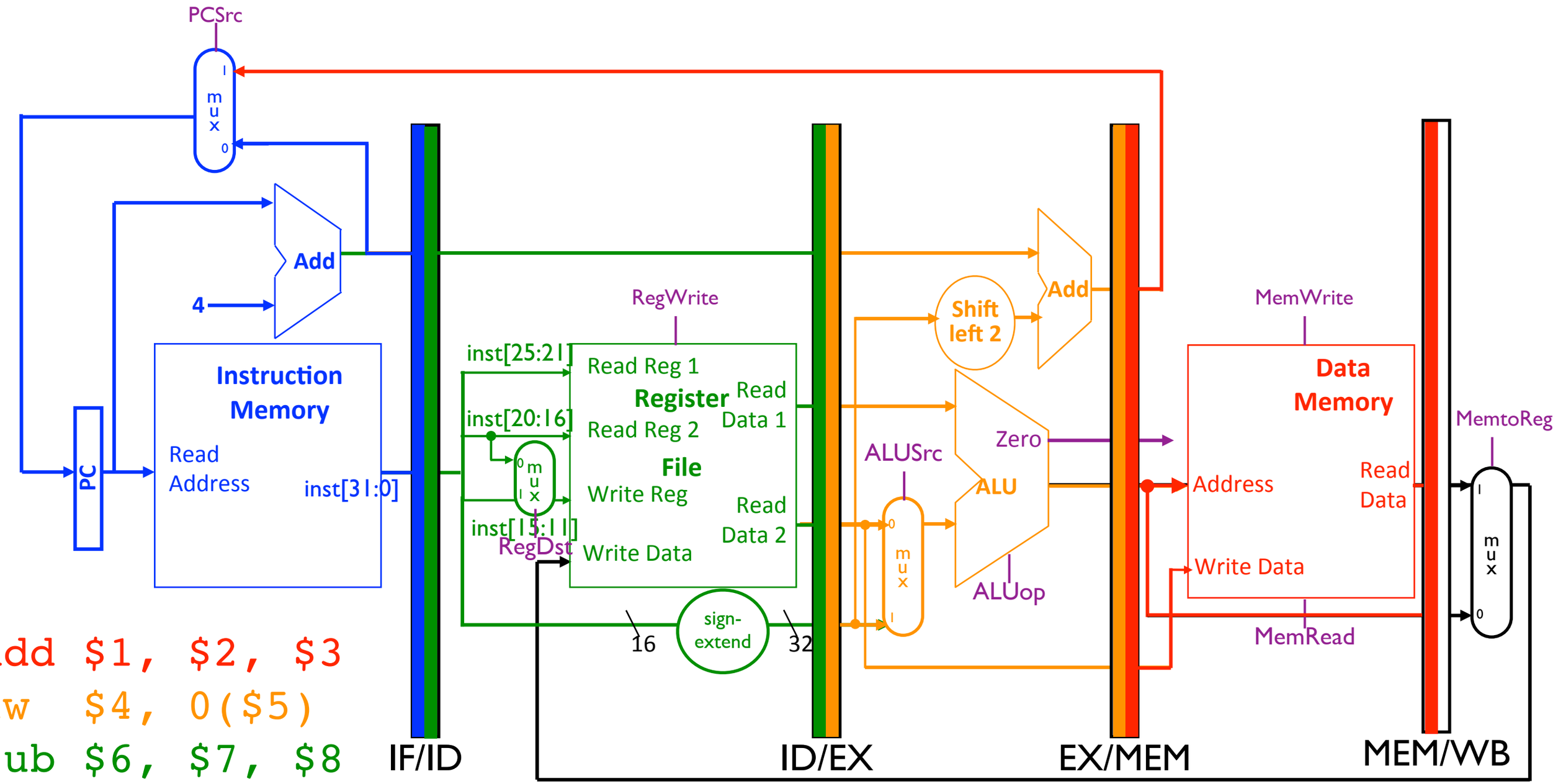

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```

Pipelined datapath



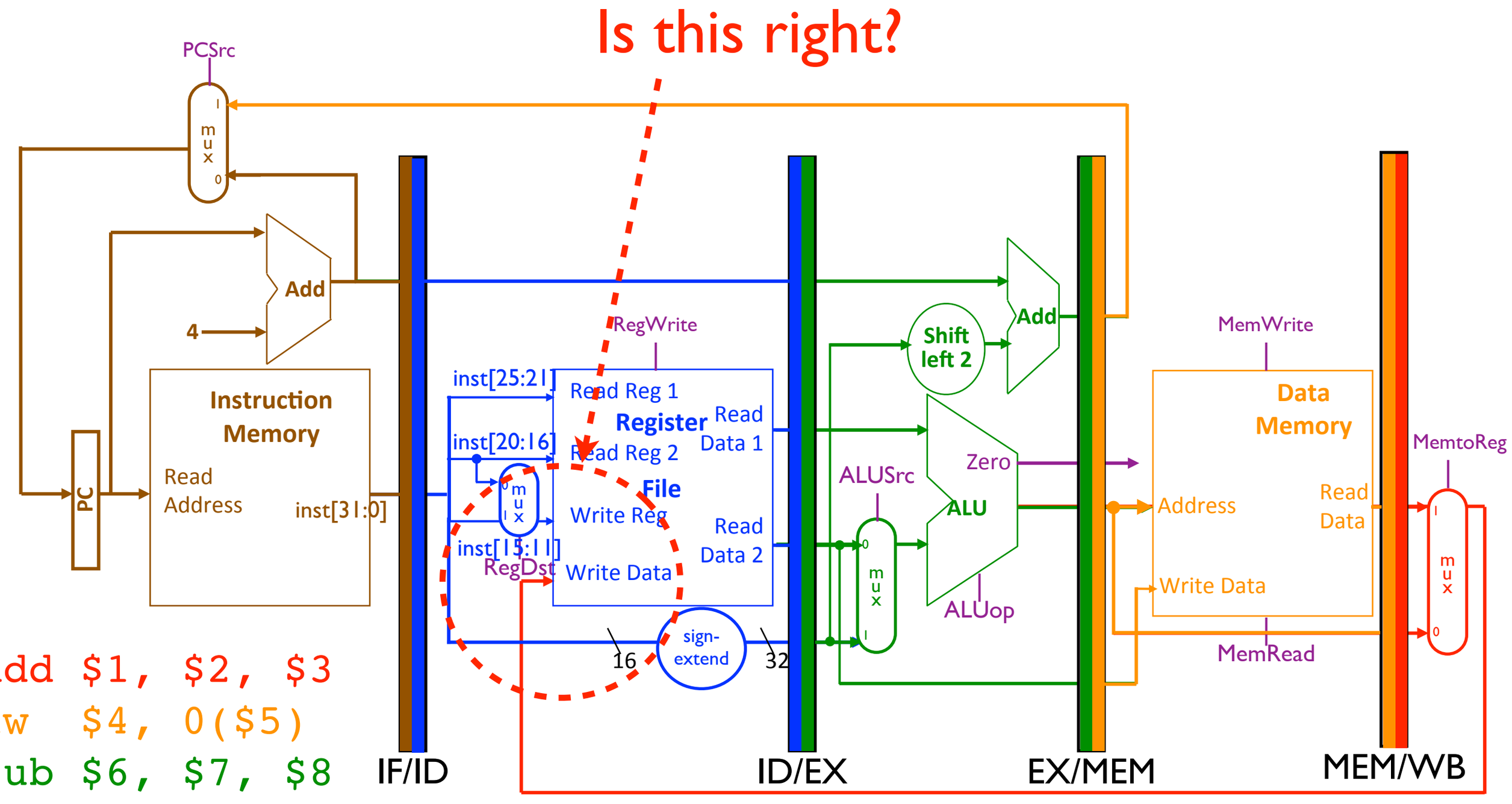

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```

Pipelined datapath



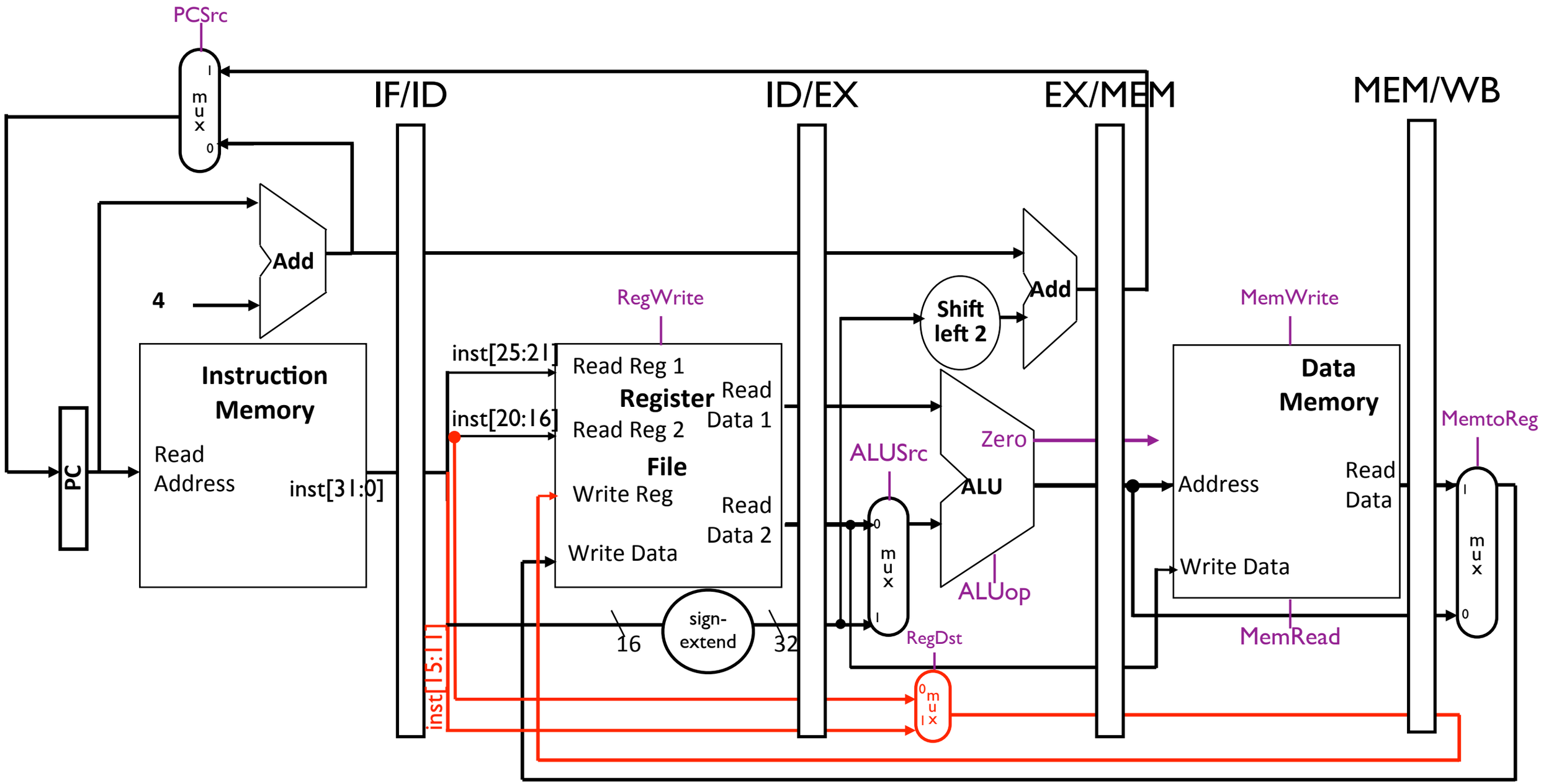

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```

Pipelined datapath

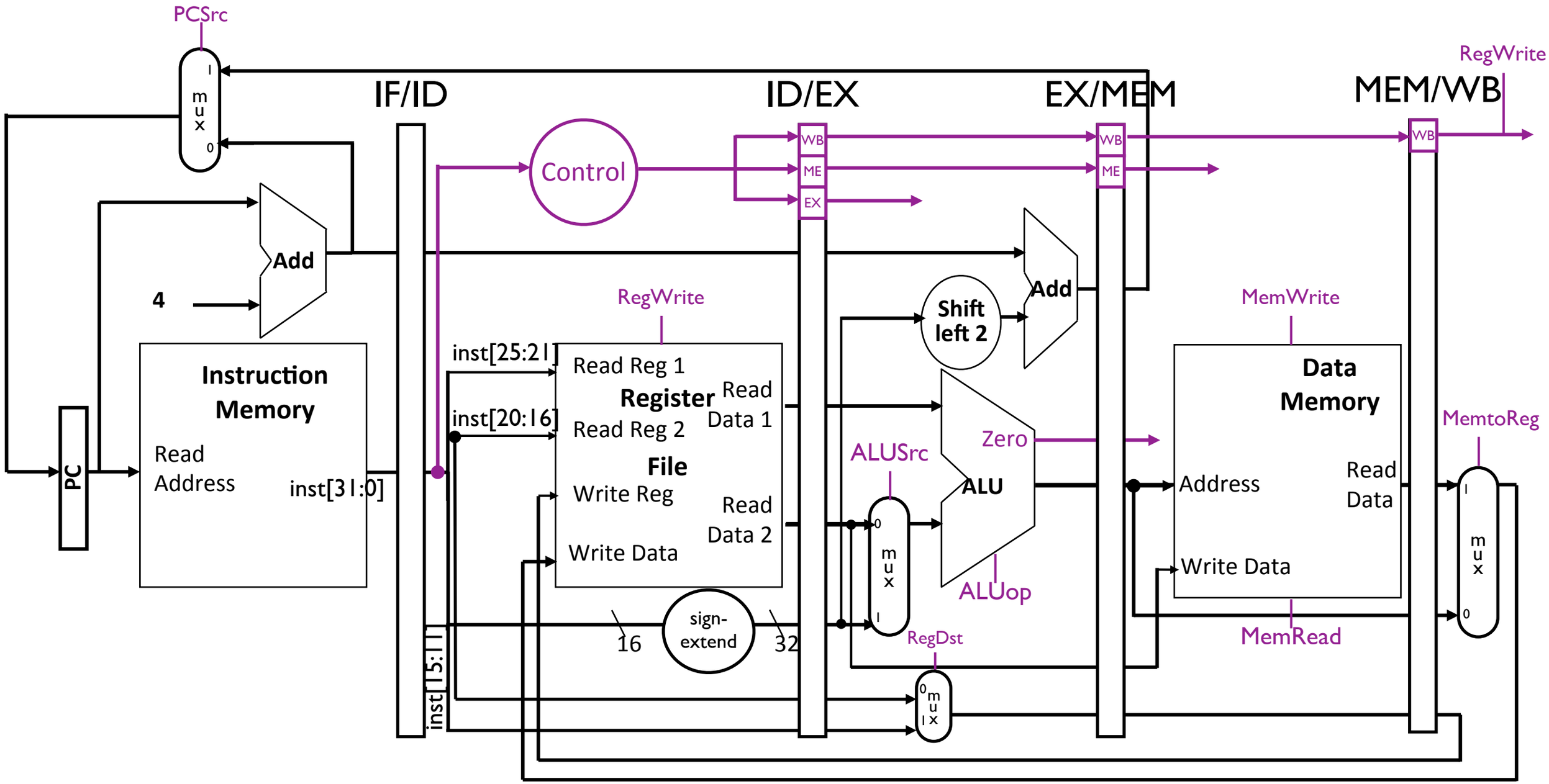



```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```

Pipelined datapath



Pipelined datapath + control

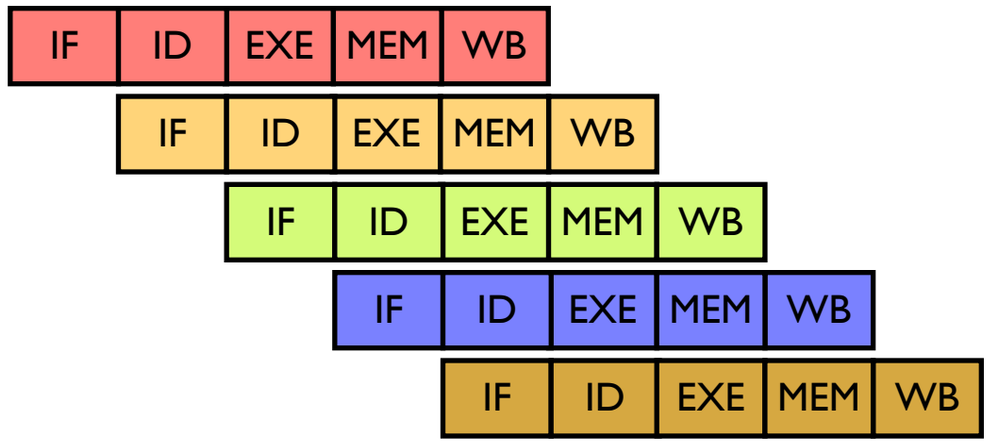


Simplified pipeline diagram

- Use symbols to represent the physical resources with the abbreviations for pipeline stages.
  - IF, ID, EXE, MEM, WB
- Horizontal axis represent the timeline, vertical axis for the instruction stream

- Example:

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```



Pipeline hazards

Pipeline hazards

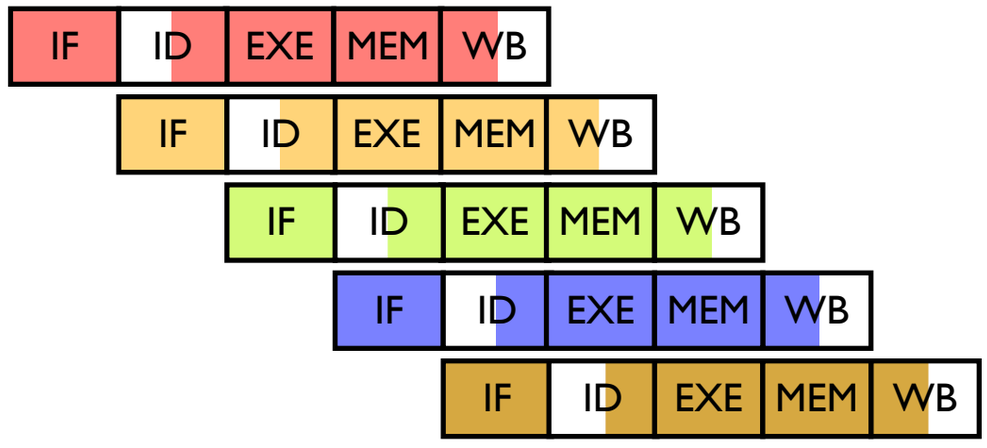
- Even though we perfectly divide pipeline stages, it's still hard to achieve CPI == 1.
- Pipeline hazards:
  - Structural hazard
    - The hardware does not allow two pipeline stages to work concurrently
  - Data hazard
    - A later instruction in a pipeline stage depends on the outcome of an earlier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

Structural hazard

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)
```



Data hazard

Data hazard

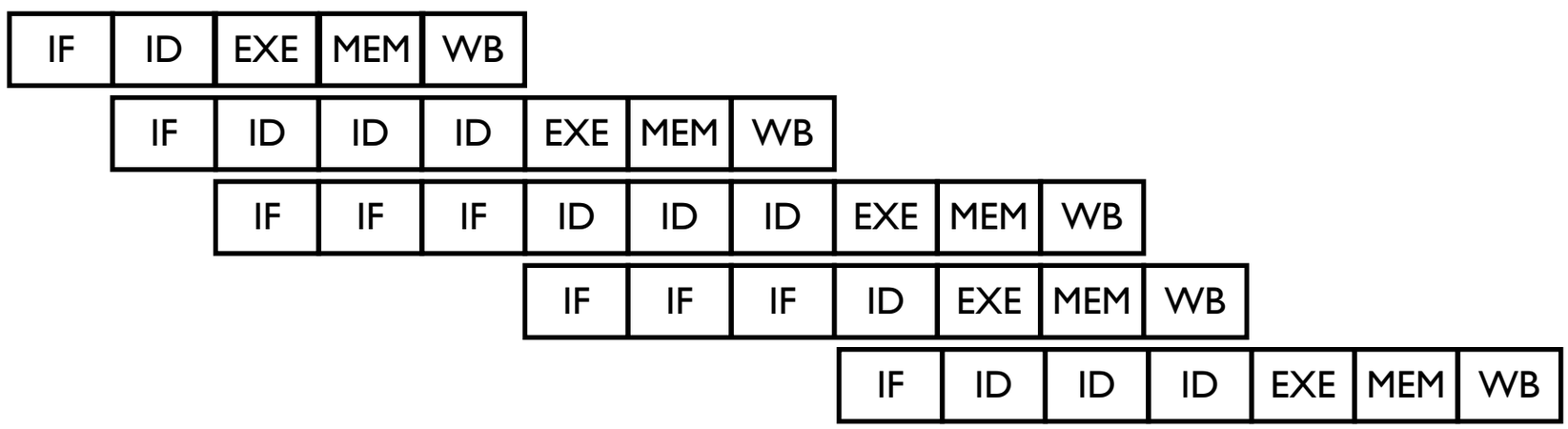
- When an instruction in the pipeline needs a value that is not available
- Data dependences
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline

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)
```



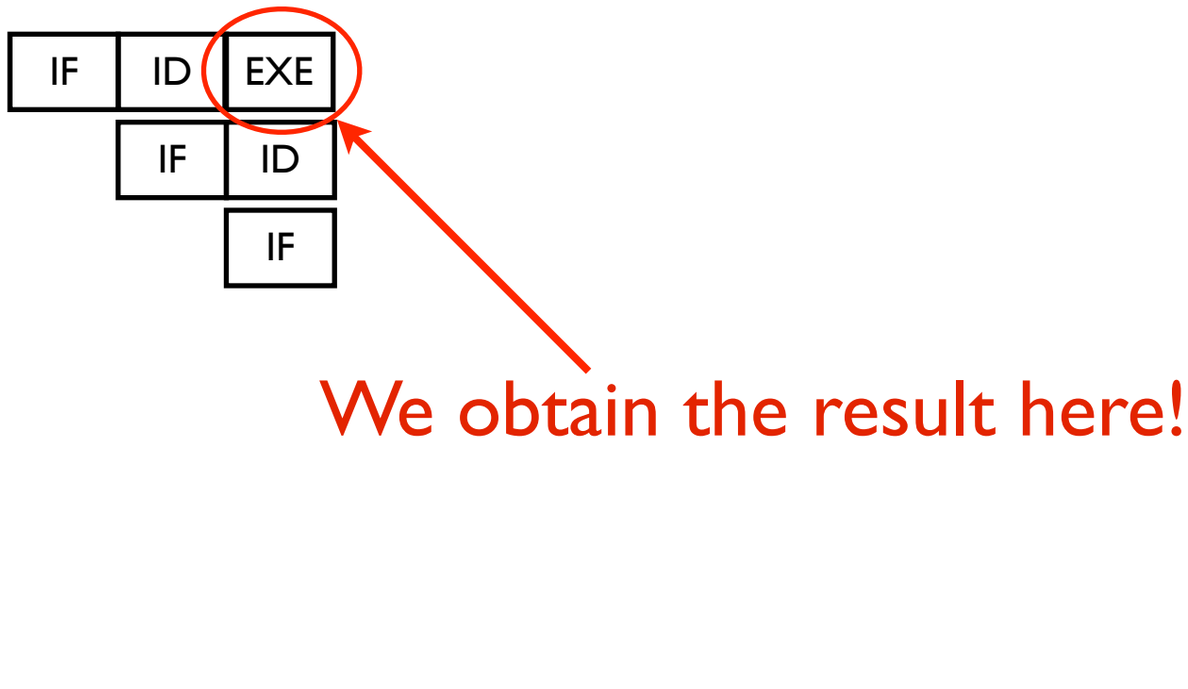
15 cycles! CPI == 3

(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

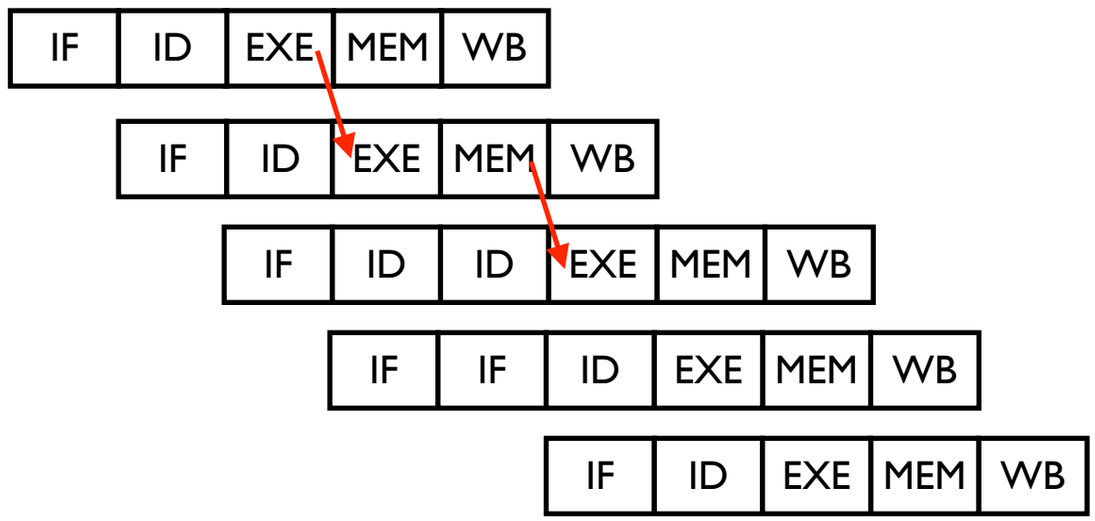
```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



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)
```

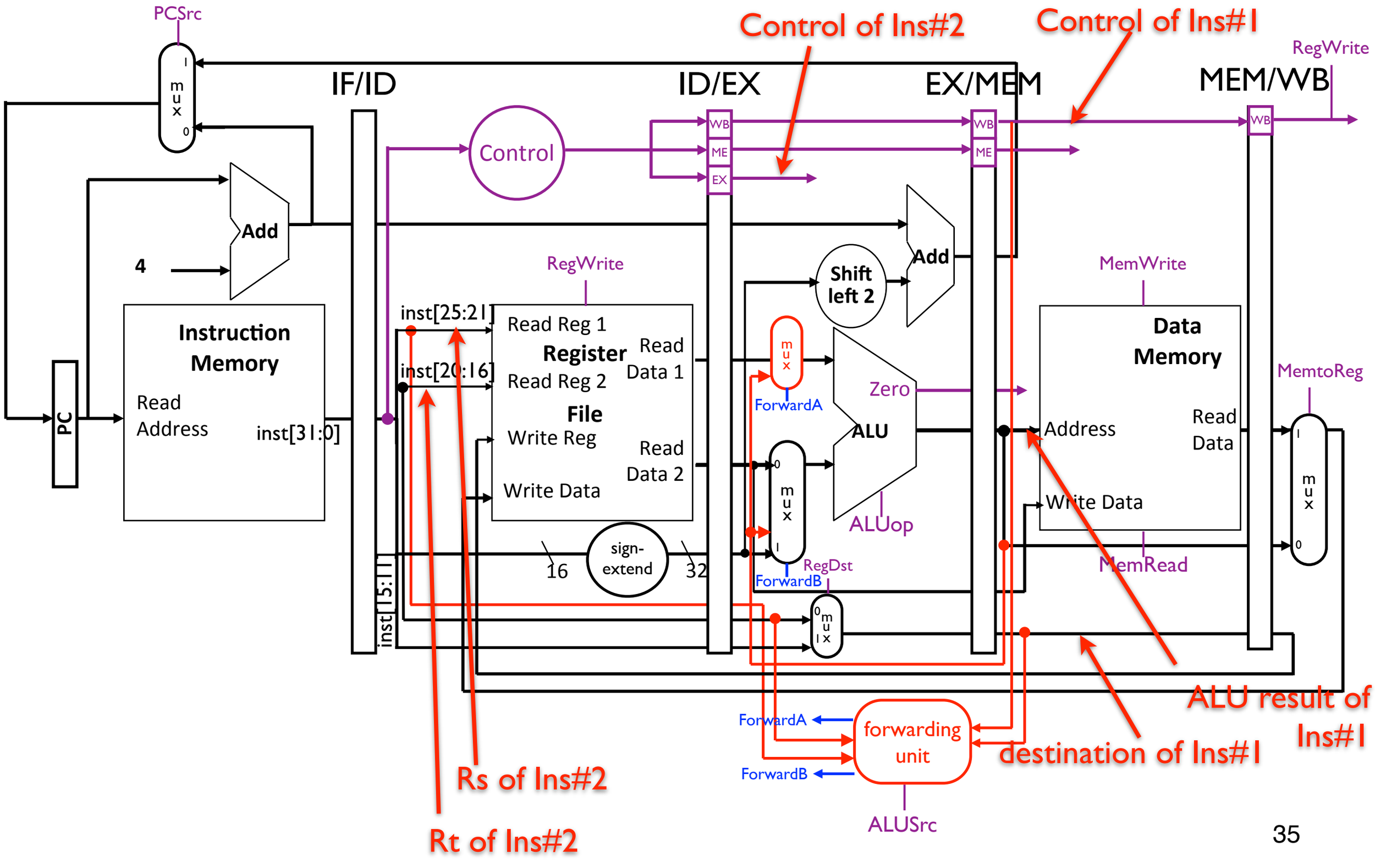


10 cycles! CPI == 2 (Not optimal, but much better!)

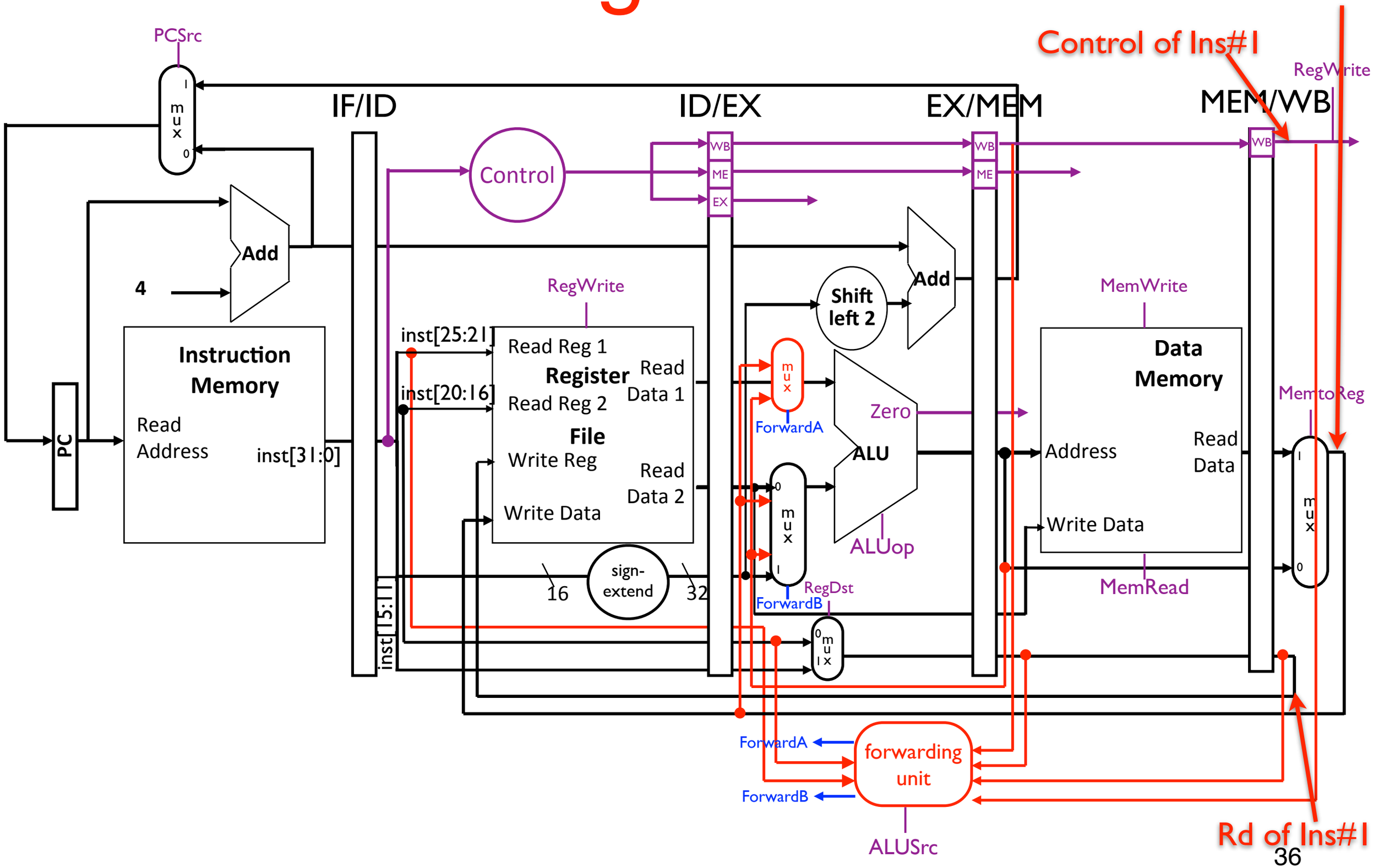
When can/should we forward data?

- 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

Forwarding in hardware



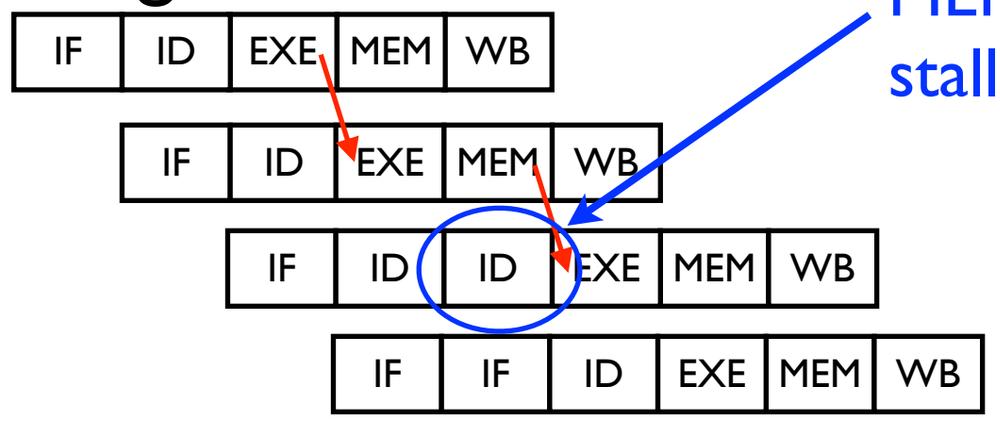
Forwarding in hardware



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)
```

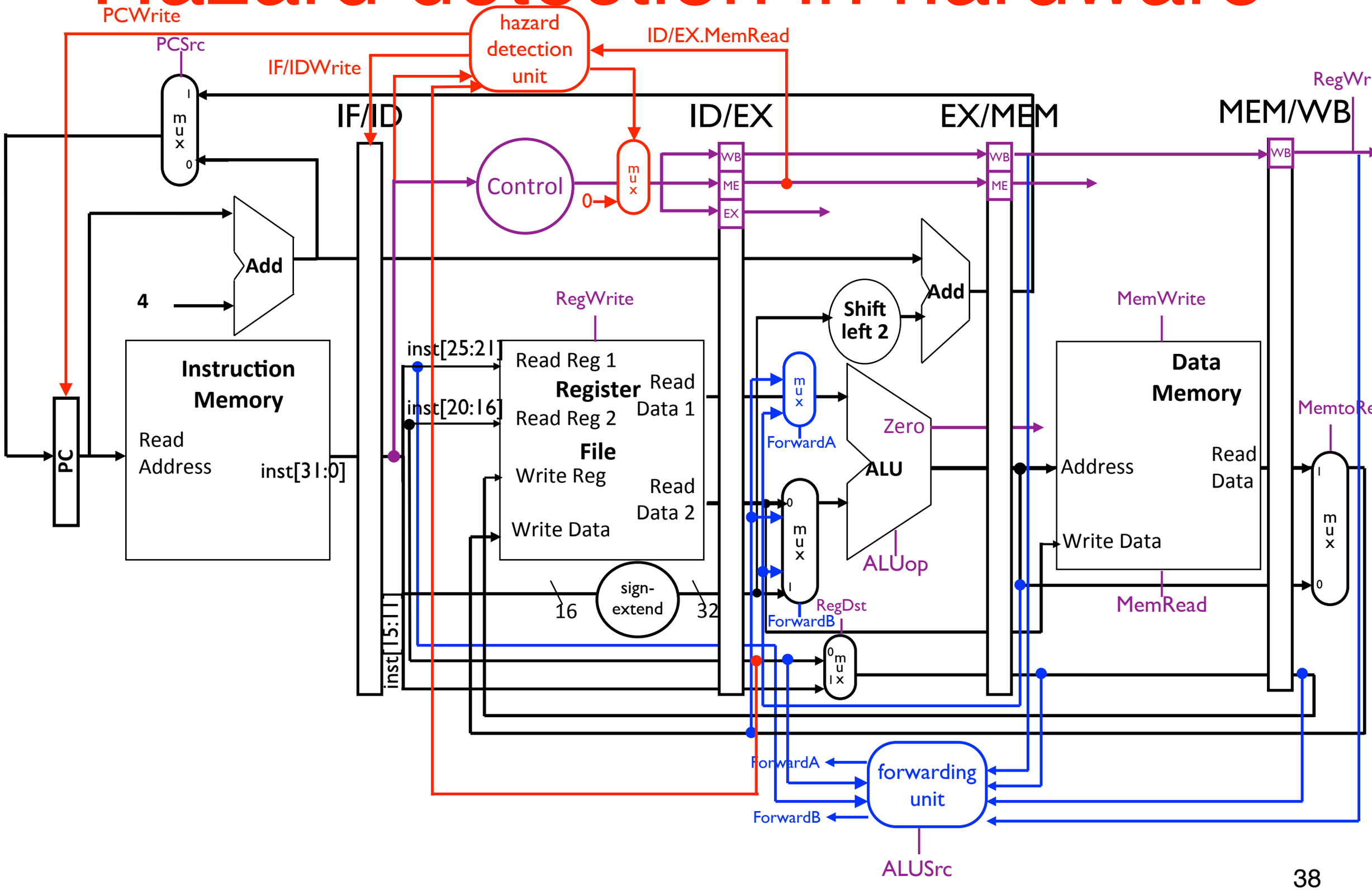


lw generates result at MEM stage, we have to stall

- If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!
  - We call this hazard detection

    We need to know the following:
    1. If an instruction in EX/MEM updates a register (RegWrite)
    2. If an instruction in EX/MEM reads memory (MemRead)
    3. If the destination register of EX/MEM is a source of ID/EX (rs, rt of ID/EX == rt of EX/MEM #1)

Hazard detection in hardware

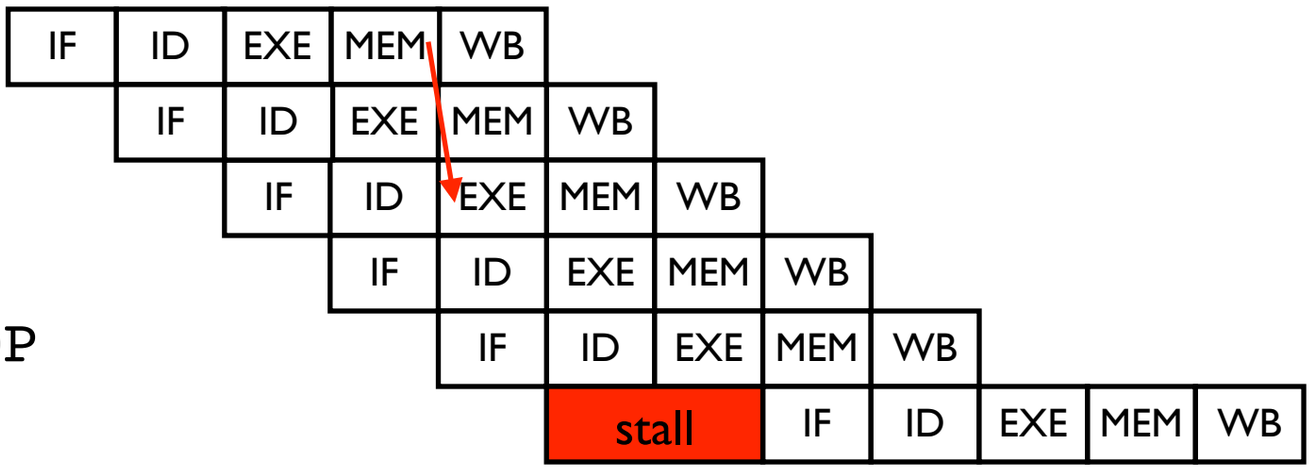


Control hazard

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
      lw   $t3, 0($s0)
```



7 cycles per loop

Solution I: Delayed branches

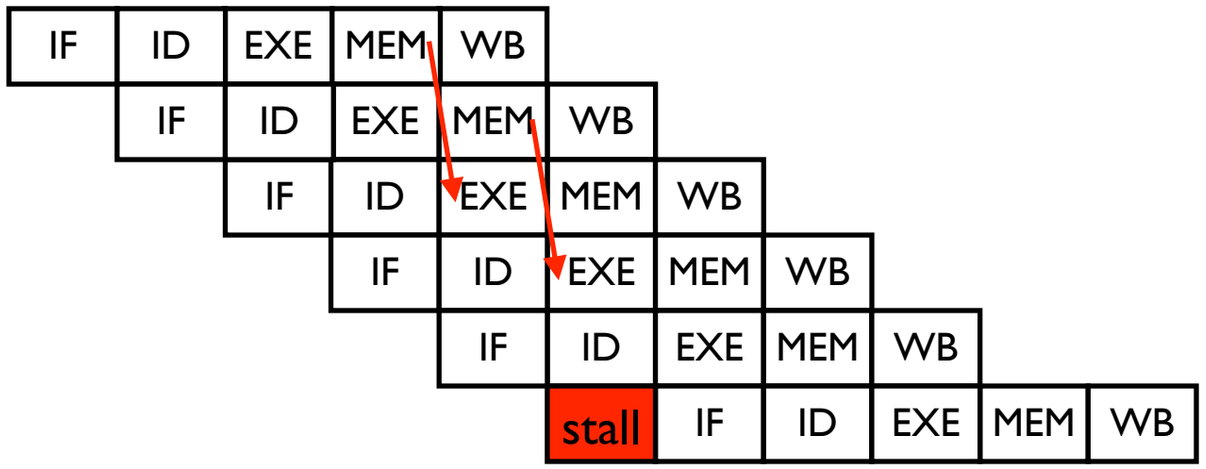
- An agreement between ISA and hardware
  - "Branch delay" slots: the next N instructions after a branch are *always* executed
  - Compiler decides the instructions in branch delay slots
    - Reordering the instruction cannot affect the correctness of the program
    - MIPS has one branch delay slot
- Good
  - Simple hardware
- Bad
  - N cannot change
  - Sometimes cannot find good candidates for the slot

Solution I: Delayed branches

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

branch delay slot

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

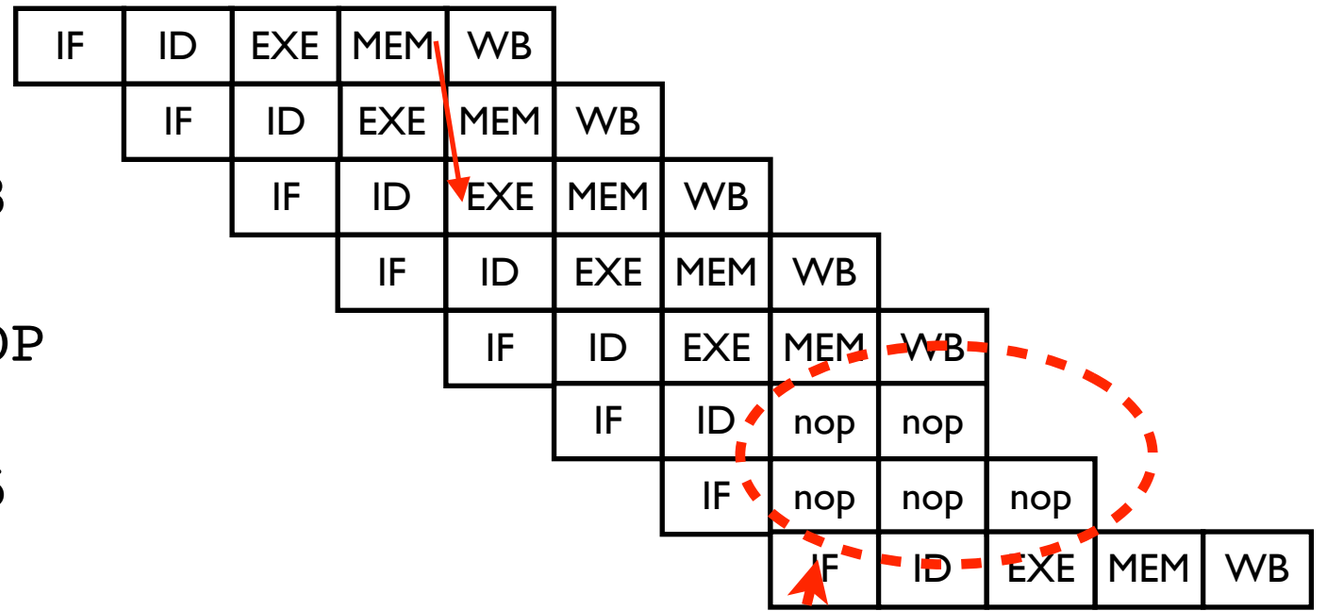


6 cycles per loop

Solution II: always predict not-taken

- Always predict the next PC is PC+4

```
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)
      add  $t4, $t3, $t5
      lw   $t3, 0($s0)
```



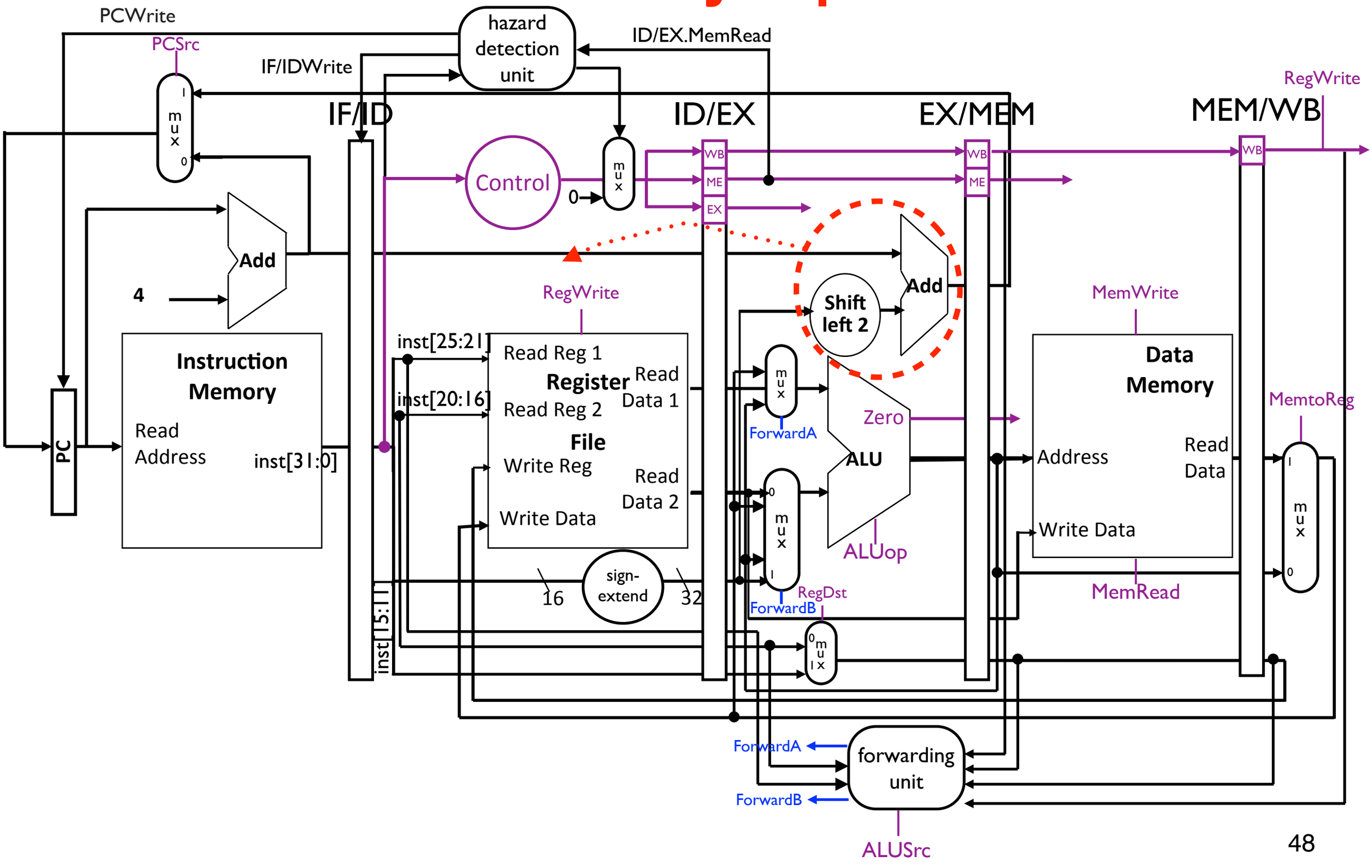
If branch is not taken: no stalls!

If branch is taken: no hurt!

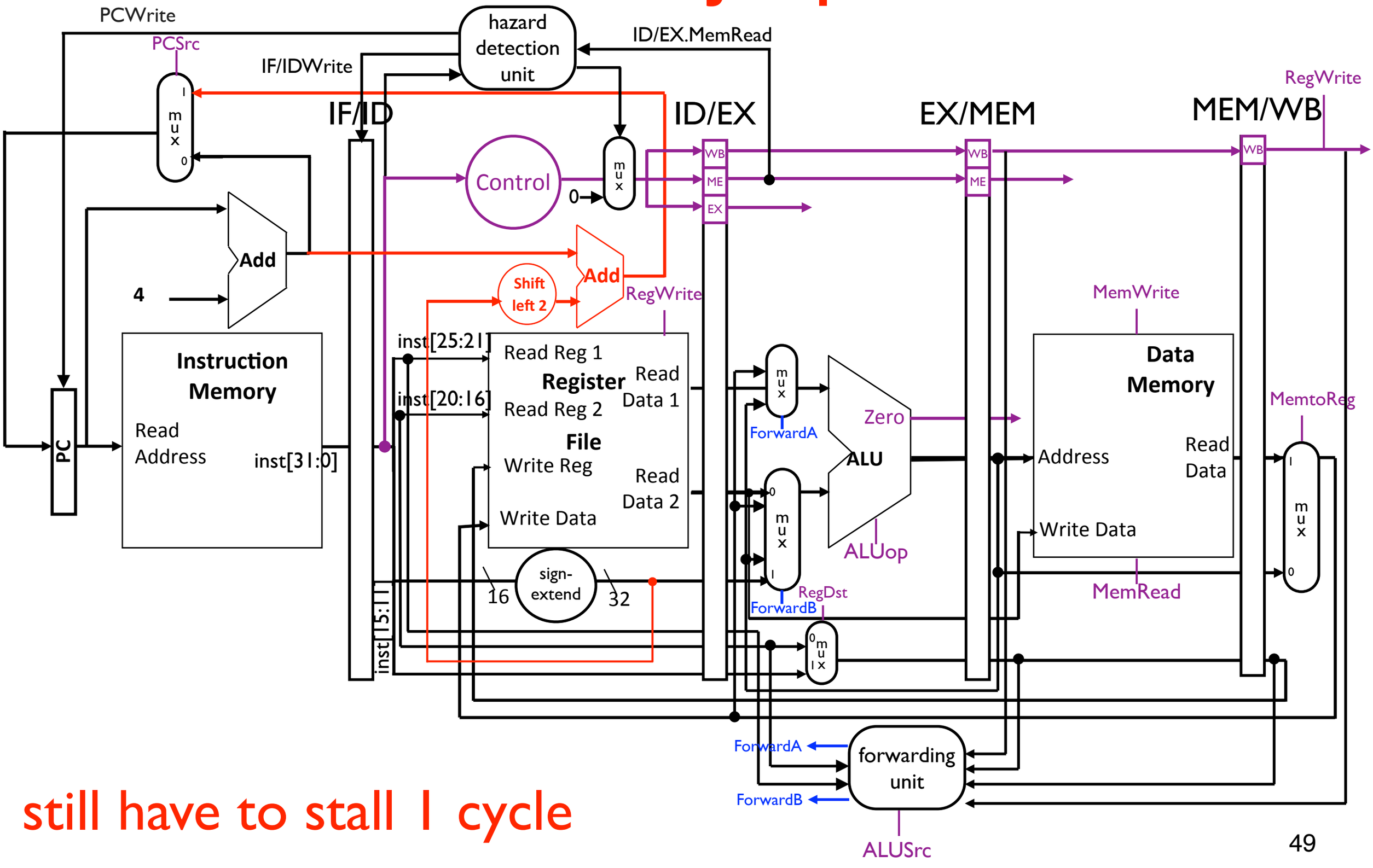
7 cycles per loop

flush the instructions fetched incorrectly

Solution III: always predict taken

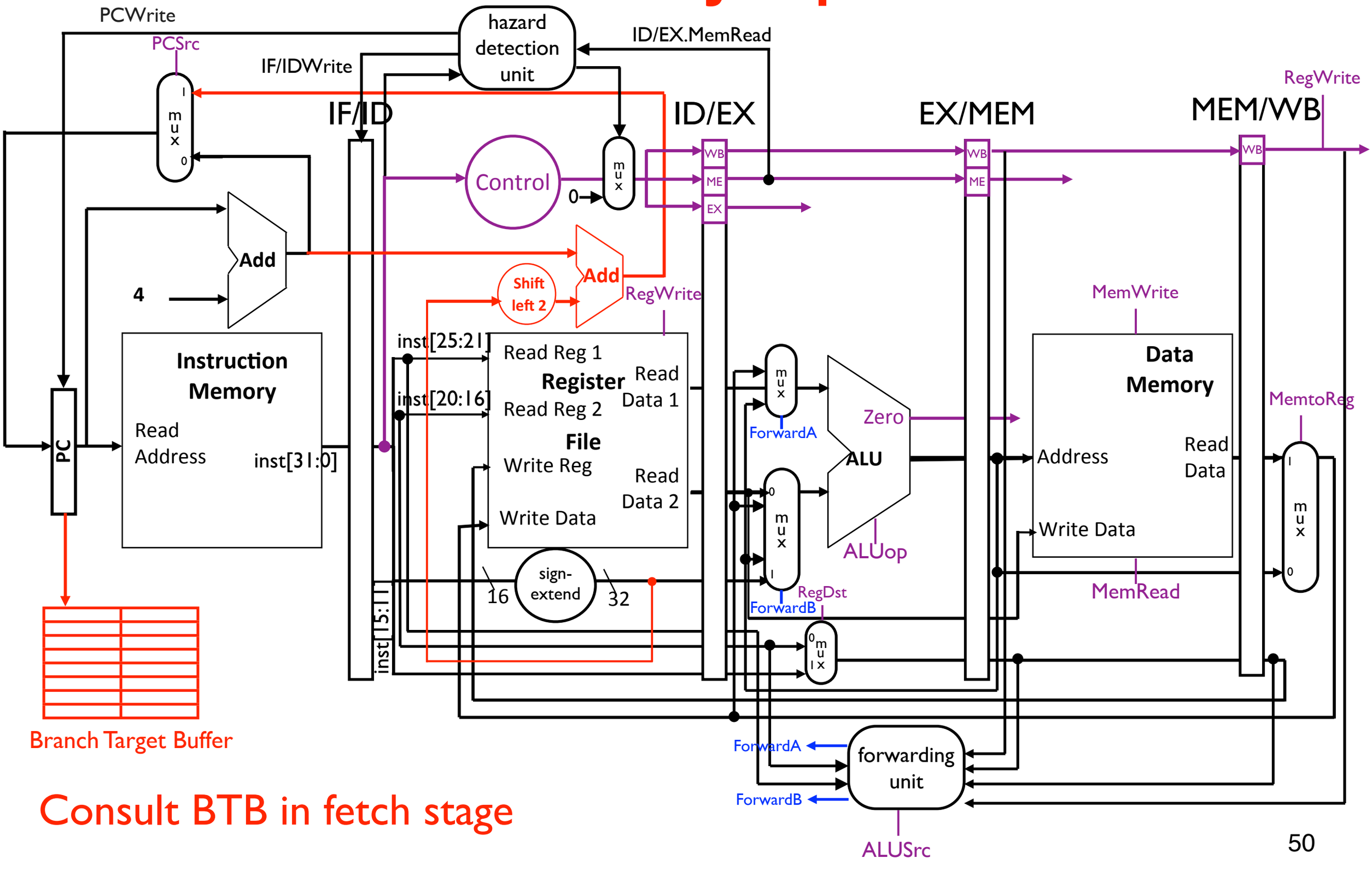


Solution III: always predict taken



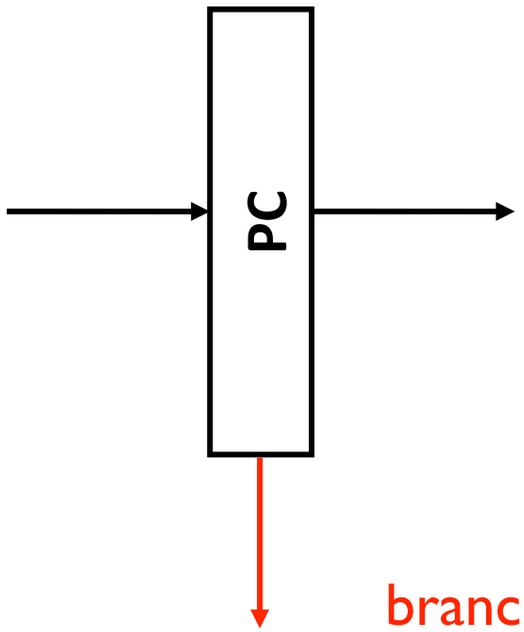
still have to stall 1 cycle

Solution III: always predict taken



Consult BTB in fetch stage

Branch Target Buffer

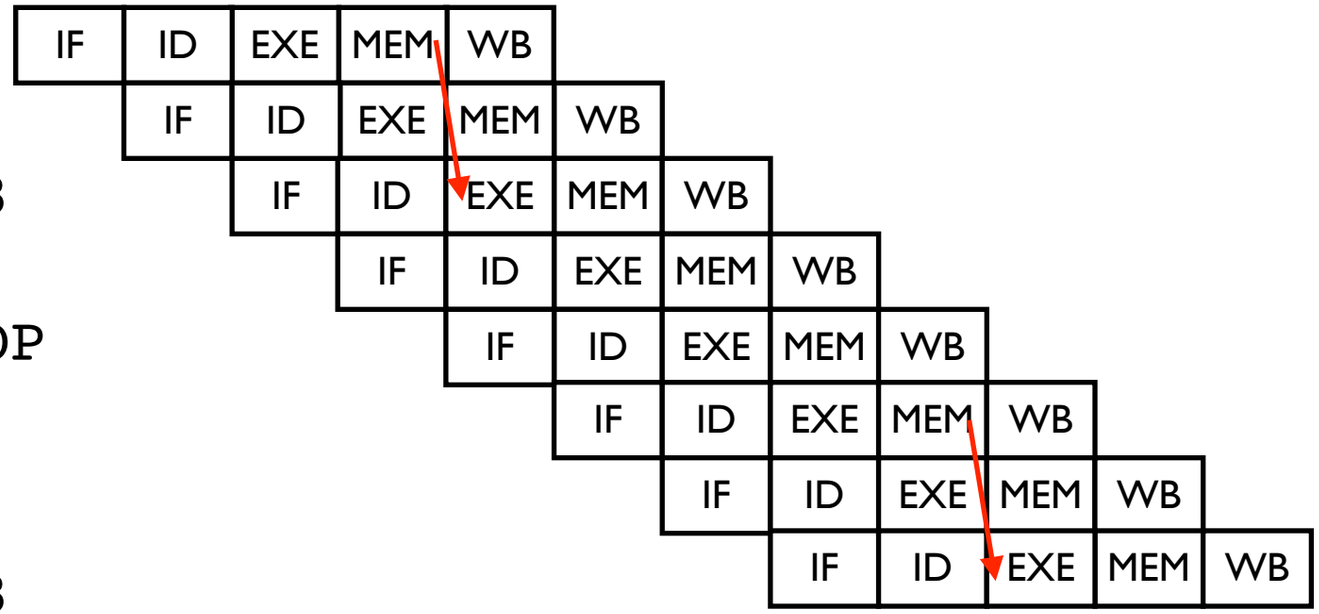


Branch Target Buffer

Solution III: always predict taken

- Always predict taken with the help of BTB

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



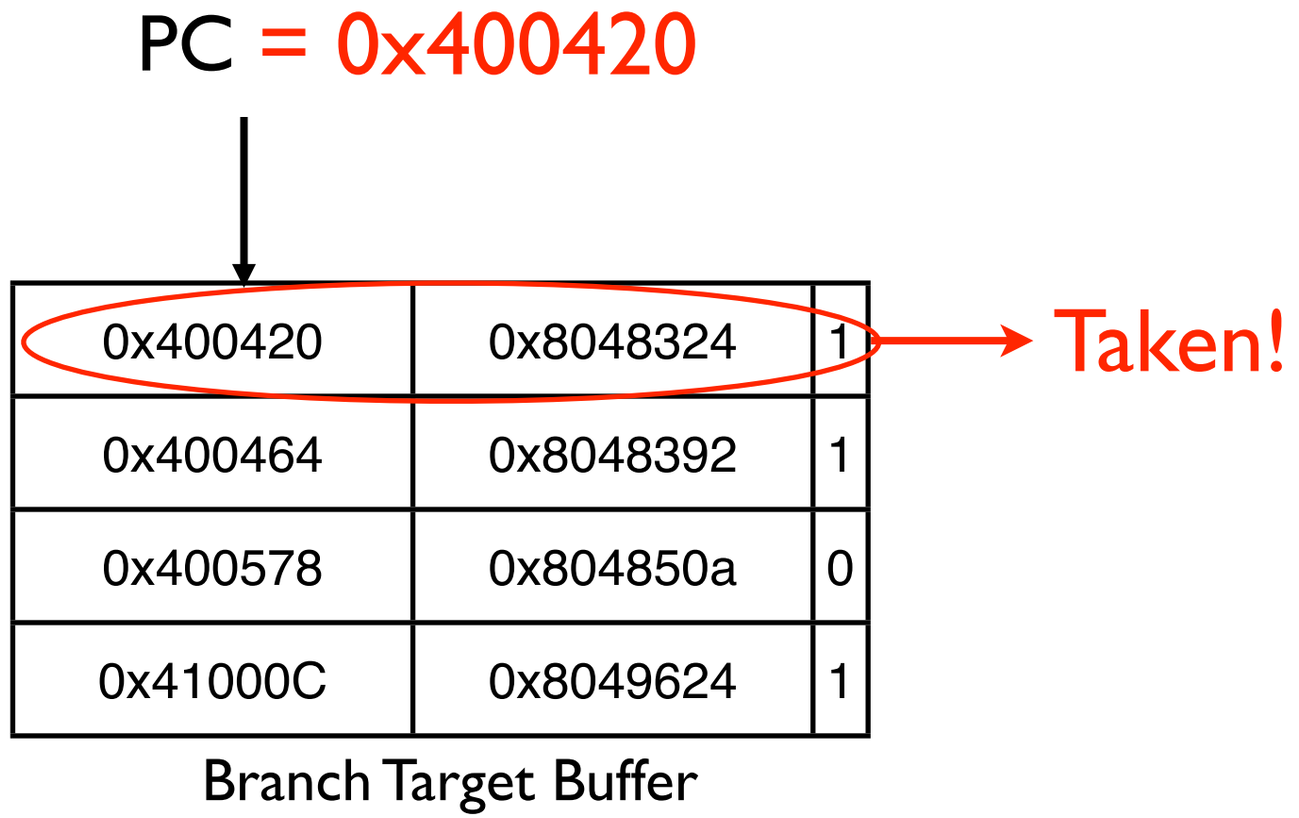
5 cycles per loop
(CPI == 1 !!!)

But what if the branch is not always taken?

Dynamic branch prediction

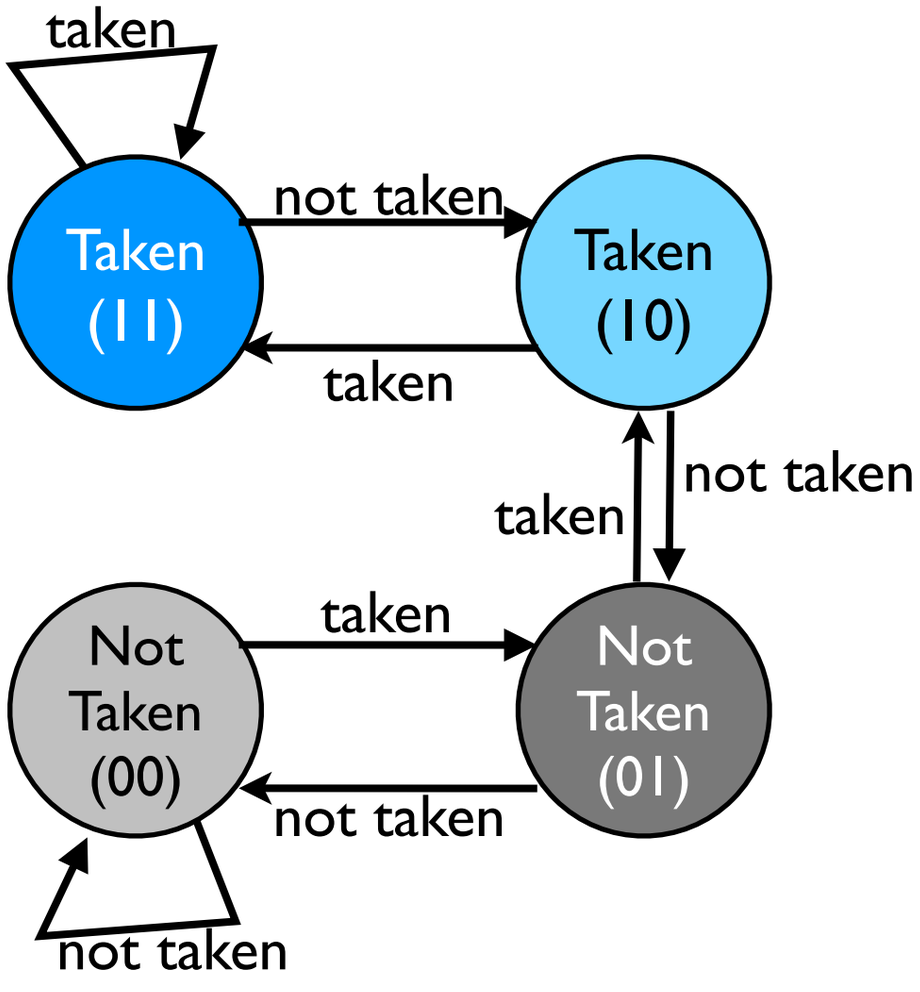
1-bit counter

- Predict this branch will go the same way as the result of the last time this branch executed
  - 1 for taken, 0 for not takens



2-bit counter

- A 2-bit counter for each branch
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4



PC = 0x400420

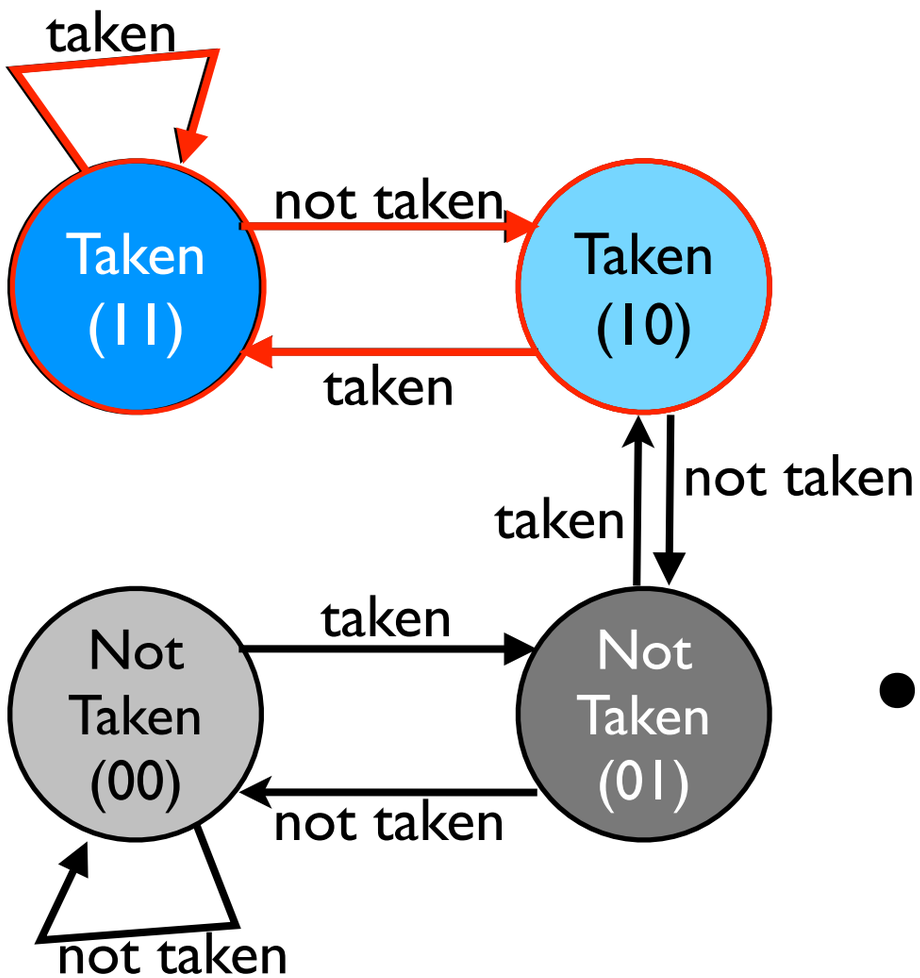
0x400420	0x8048324	11
0x400464	0x8048392	10
0x400578	0x804850a	00
0x41000C	0x8049624	01

Taken!

Branch Target Buffer

Performance of 2-bit counter

- 2-bit state machine for each branch




```
for(i = 0; i < 10; i++)
{
   sum += a[i];
}
```

90% prediction rate!

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
  - 1+20%*(1-90%)*2 = 1.04

Make the prediction better

- Consider the following code:

```
i = 0;
do {
    if( i % 3 != 0)  // Branch Y,
taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

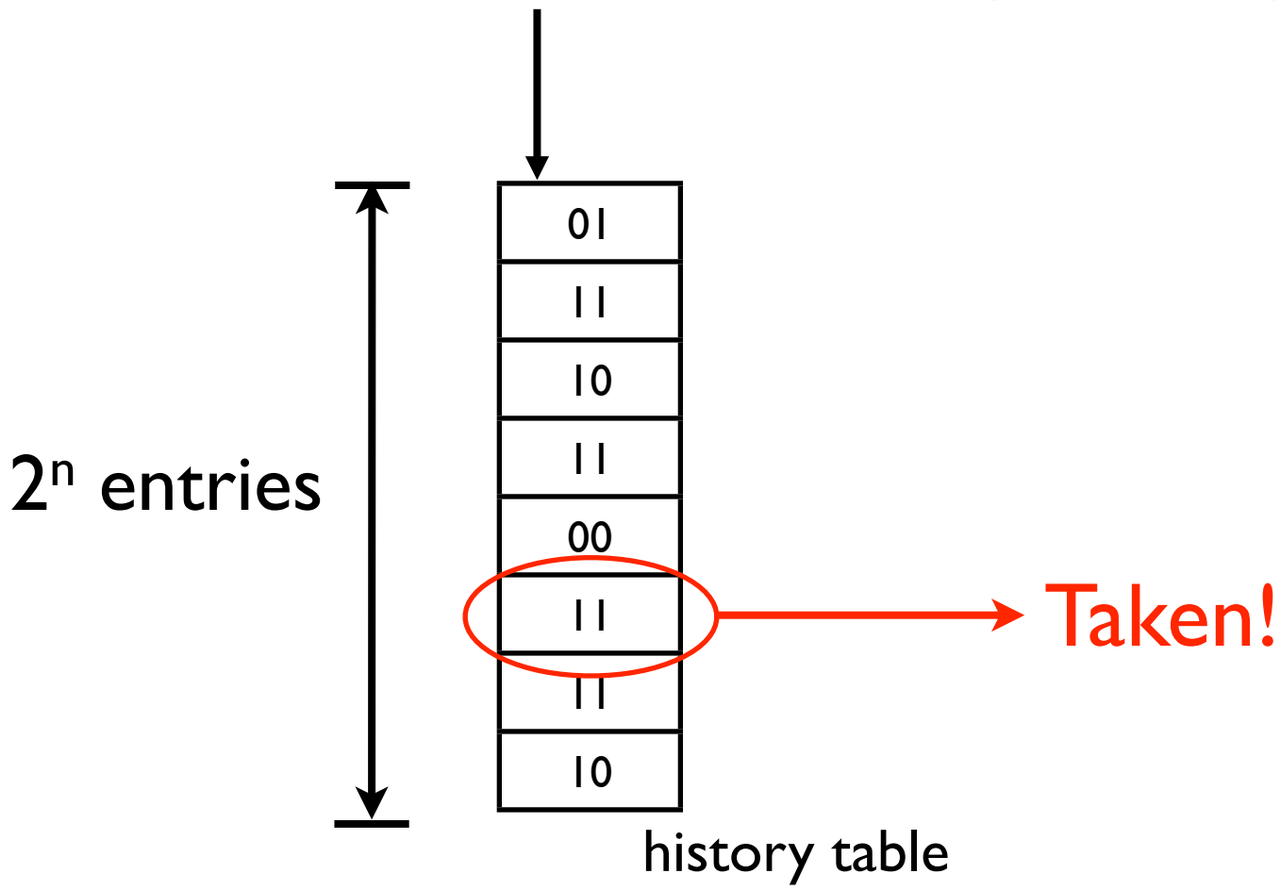
Can we capture the pattern?

i	branch	result
0	Y	T
0	X	T
1	Y	NT
1	X	T
2	Y	NT
2	X	T
3	Y	T
3	X	T
4	Y	NT
4	X	T
5	Y	NT
5	X	T
6	Y	T
6	X	T
7	Y	NT

Predict using history

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Each entry in the history table has its own counter.

n-bit GHR = 101 (T, NT, T)



Performance of global history predictor

- Consider the following code:

```
i = 0;
do {
     if( i % 3 != 0)   // Branch Y,
taken if i % 3 == 0
         a[i] *= 2;
     a[i] += i;
// Branch Y
} while ( ++i < 100) // Branch X
```

Assume that we start with a 4-bit GHR= 0, all counters are 10.

i	?	GHR	BHT	prediction	actual	New BHT
0	Y	0000	10	T	T	11
0	X	0001	10	T	T	11
1	Y	0011	10	T	NT	01
1	X	0110	10	T	T	11
2	Y	1101	10	T	NT	01
2	X	1010	10	T	T	11
3	Y	0101	10	T	T	11
3	X	1011	10	T	T	11
4	Y	0111	10	T	NT	01
4	X	1110	10	T	T	11
5	Y	1101	01	NT	NT	00
5	X	1010	11	T	T	11
6	Y	0101	11	T	T	11
6	X	1011	11	T	T	11
7	Y	0111	01	NT	NT	00
7	X	1110	11	T	T	11
8	Y	1101	00	NT	NT	00
8	X	1010	11	T	T	11
9	Y	0101	11	T	T	11
9	X	1011	11	T	T	11
10	Y	0111	00	NT	NT	00

Nearly perfect after this