Pipelining with no hazards

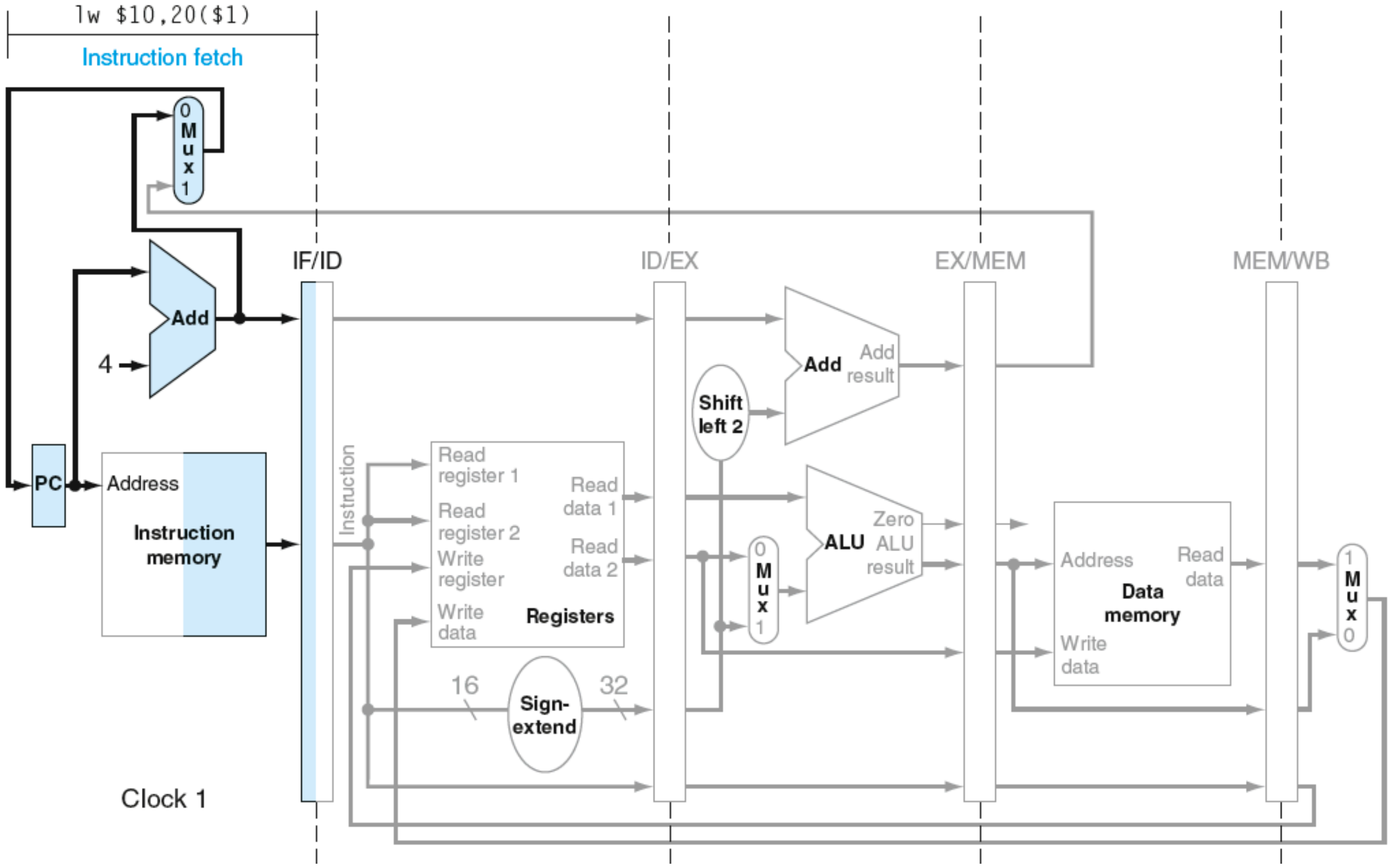
Pipelining the instructions below produces the expected results

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

There are no dependencies, so no stalls are needed

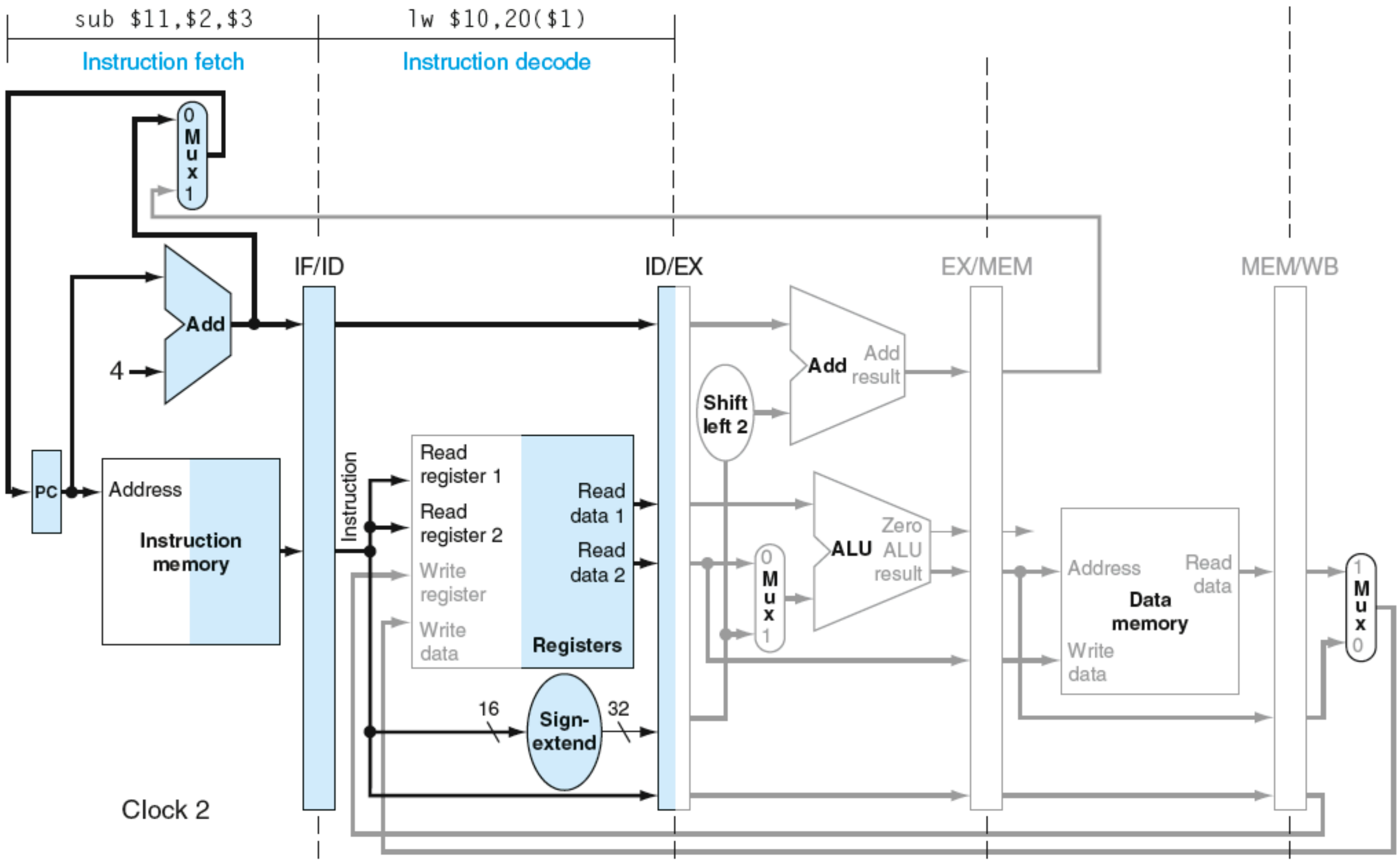
As a simple example, we can trace the first 2 instructions

Pipelining with no hazards



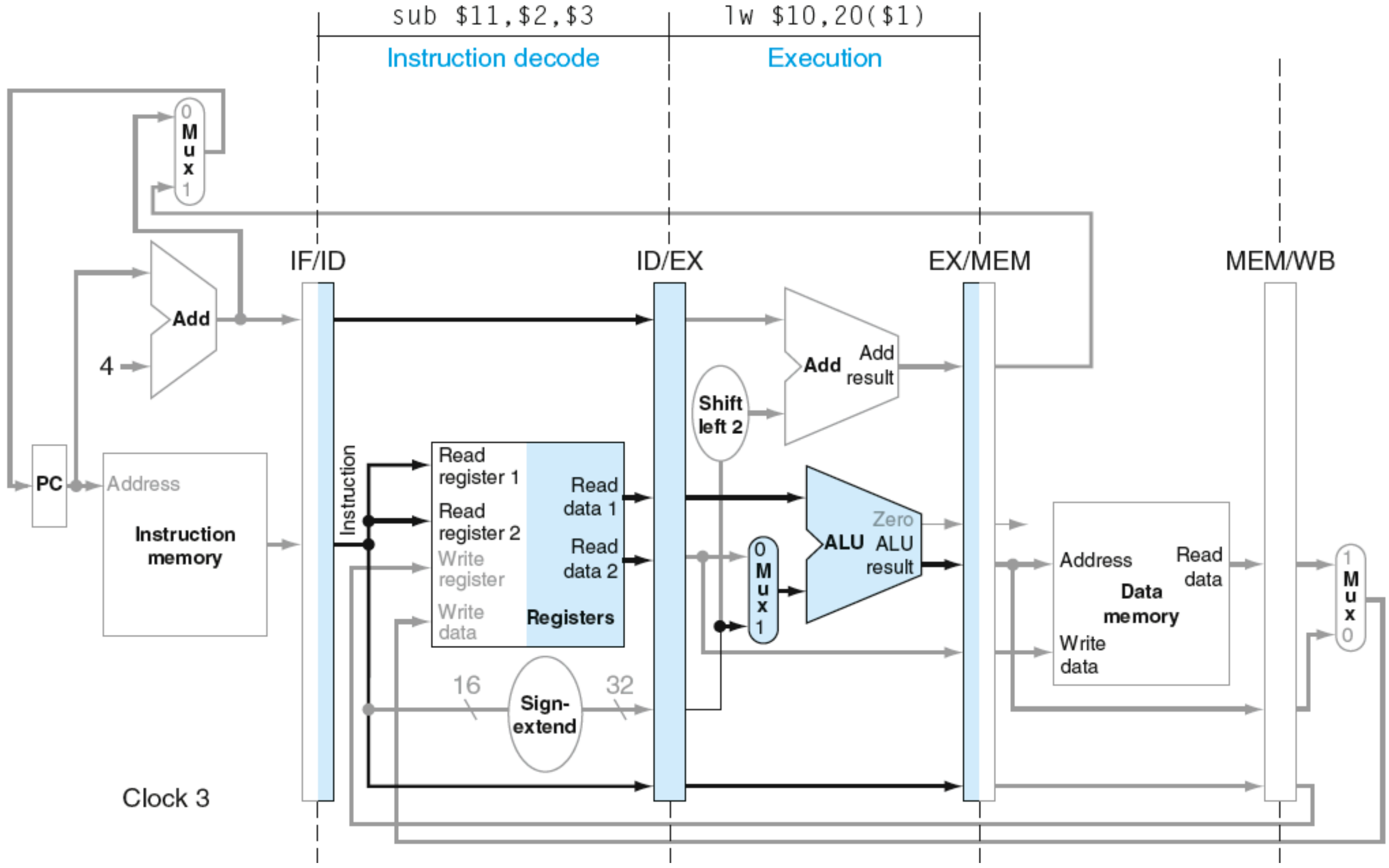
LW is fetched in cycle 1

Pipelining with no hazards



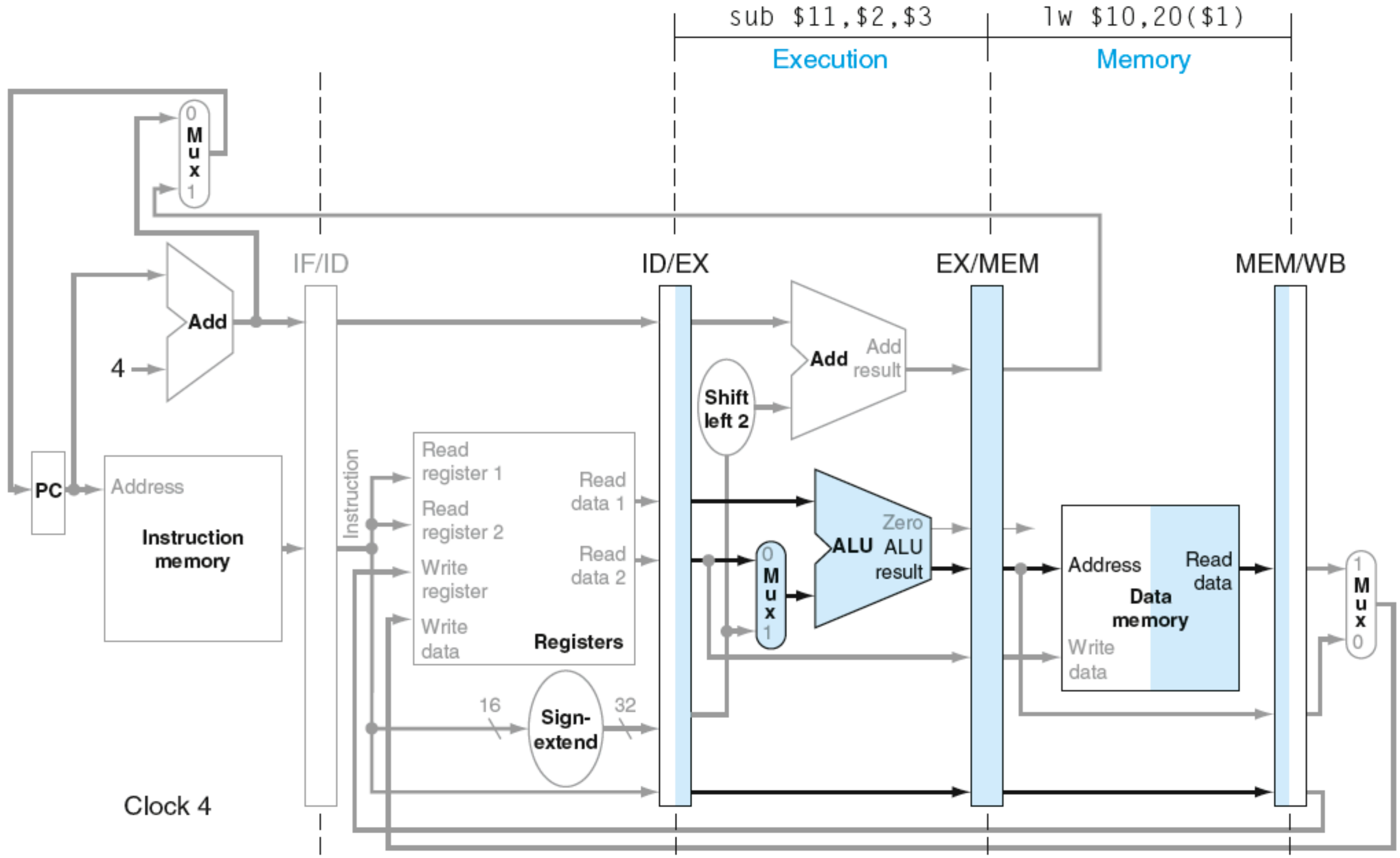
In cycle 2, LW advances to the decode stage and SUB is fetched

Pipelining with no hazards



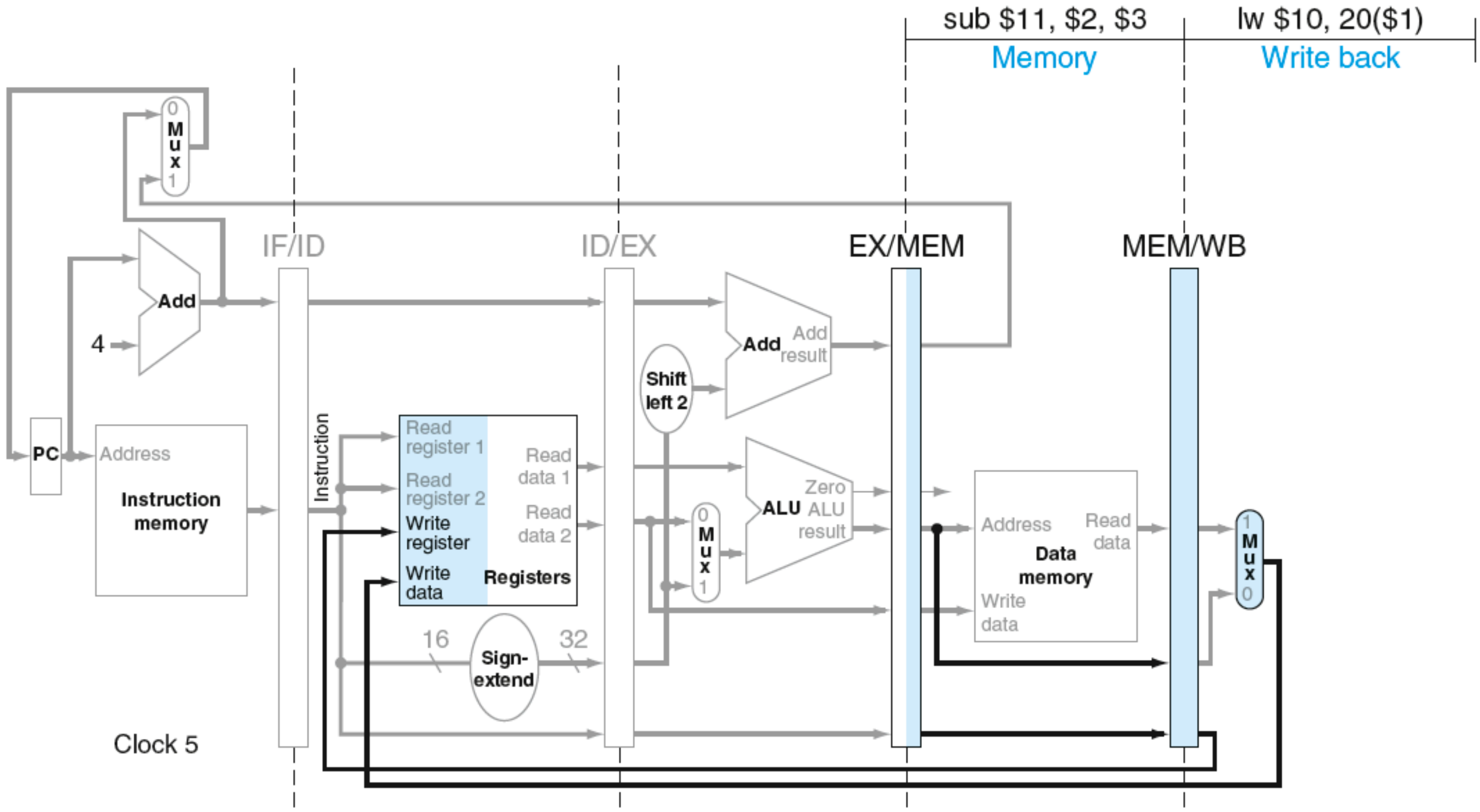
By cycle 3, LW is in the execute stage and SUB is in the decode stage

Pipelining with no hazards



LW reads from the data memory in cycle 4, while SUB executes in stage 3

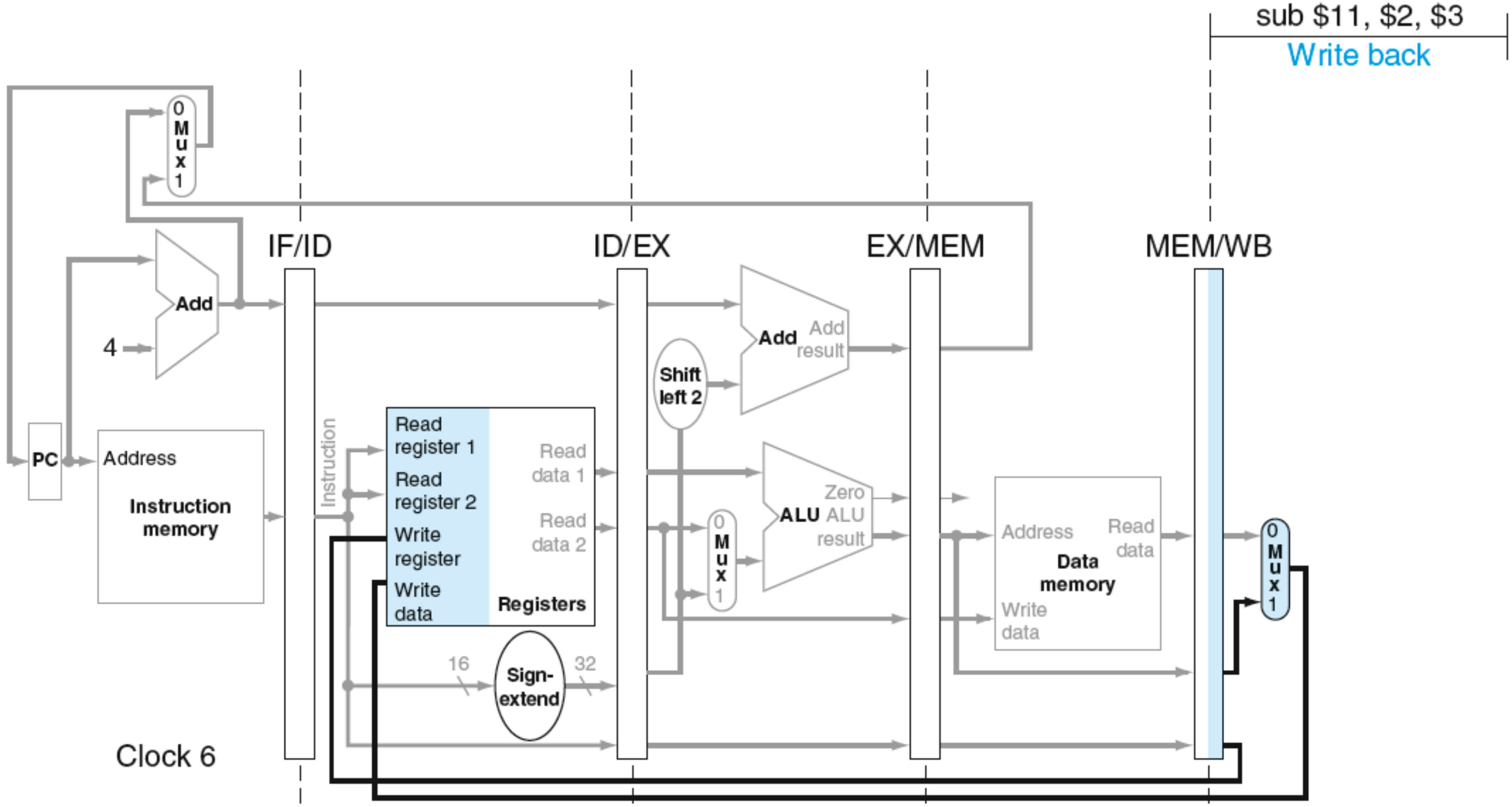
Pipelining with no hazards



LW completes in cycle 5 by writing its result into $10

SUB simply idles in stage 4 since it does not access memory

Pipelining with no hazards



SUB completes in cycle 6, writing its result into $11

The two instructions require a total of 6 cycles to complete

Pipelining with no hazards

- Pipelining provides a higher throughput
  - More instructions complete per unit time
  - Each instruction takes 5 cycles
  - But instructions are overlapped

- On the non-pipelined multi-cycle system
  - lw takes 5 cycles
  - sub takes 4 cycles
  - Total for this example = 9 cycles instead of 6
  - Each instruction completes before the next starts