Chapter 6 Solutions: For More Practice

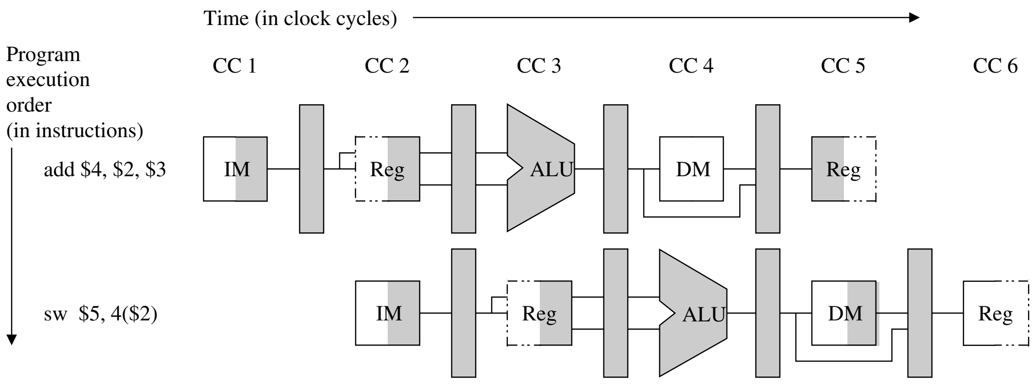
6.5 Obviously either the load or the `addi` must occupy the branch delay slot. We can't just put the `addi` into the slot because the branch instruction needs to compare `$3` with register `$4` and the `addi` instruction changes `$3`. In order to move the load into the branch delay slot, we must realize that `$3` will have changed. If you like, you can think of this as a two-step transformation. First, we rewrite the code as follows:

```
Loop:    addi $3, $3, 4
         lw   $2, 96($3)
         beq  $3, $4, Loop
```

Then we can move the load into the branch delay slot:

```
Loop:    addi $3, $3, 4
         beq  $3, $4, Loop
         lw   $2, 96($3)        # branch delay slot
```

6.7 See following figure.



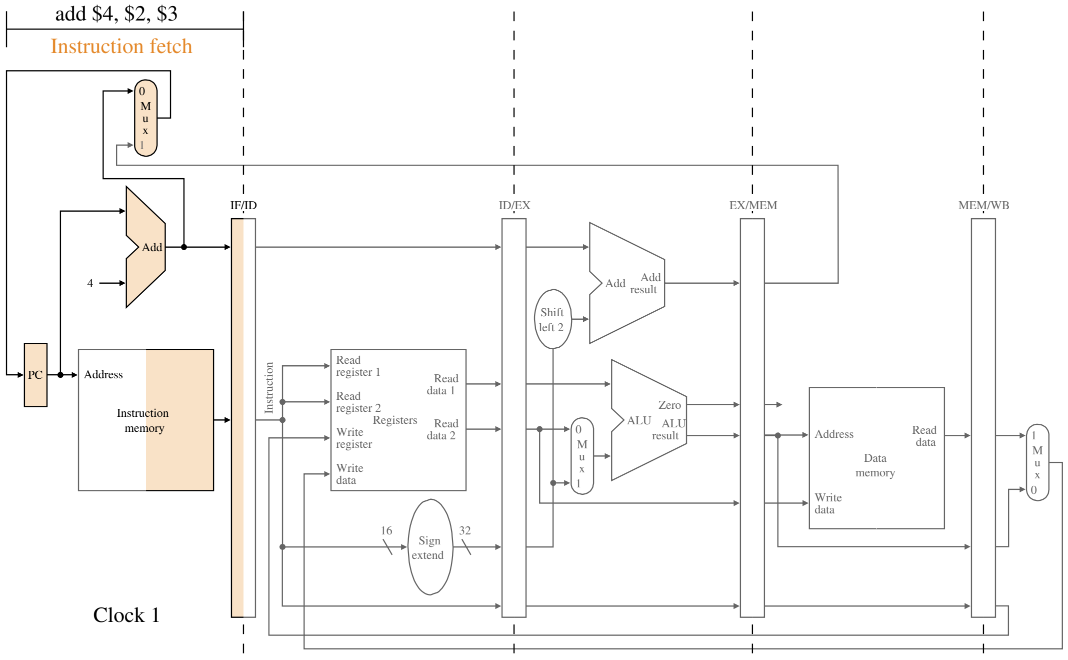
6.8 See following figure.

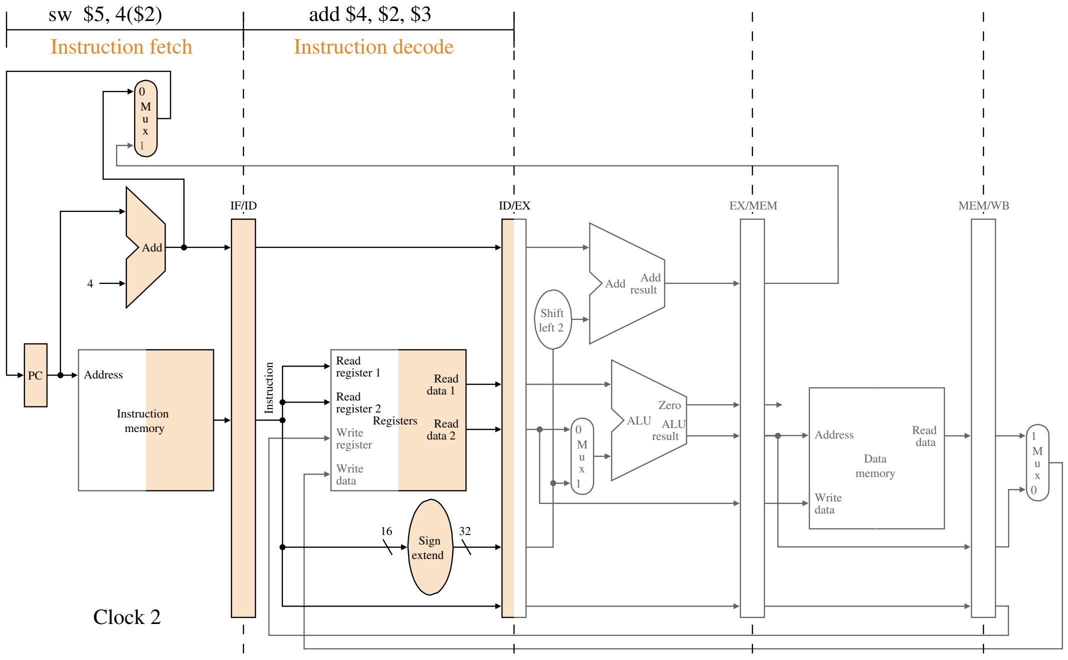
Time (in clock cycles)

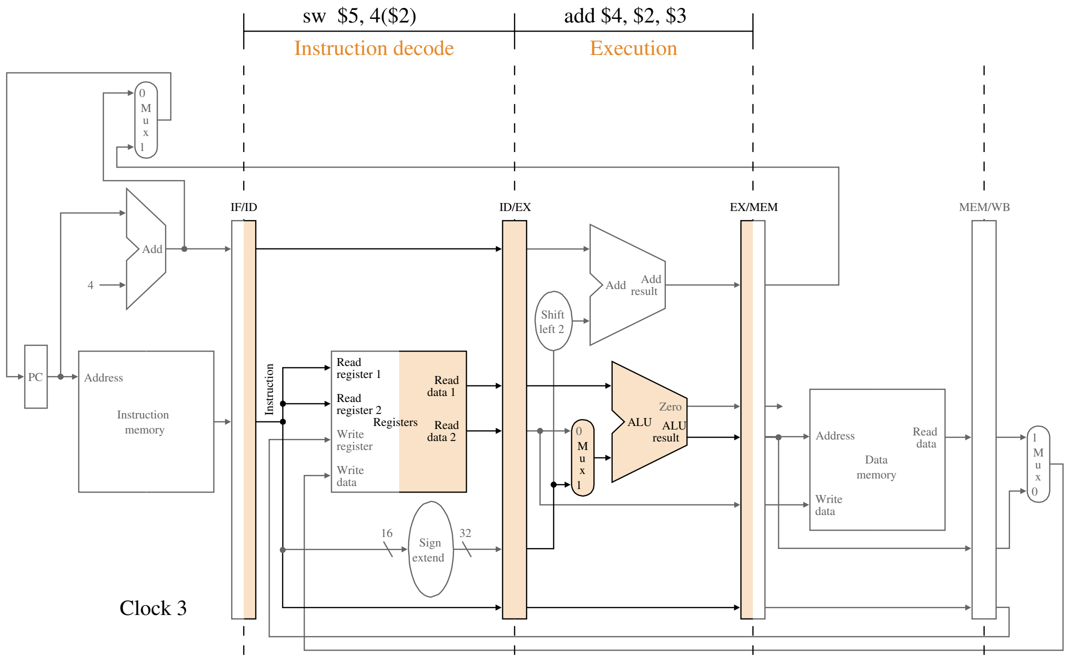
Program execution order (in instructions)

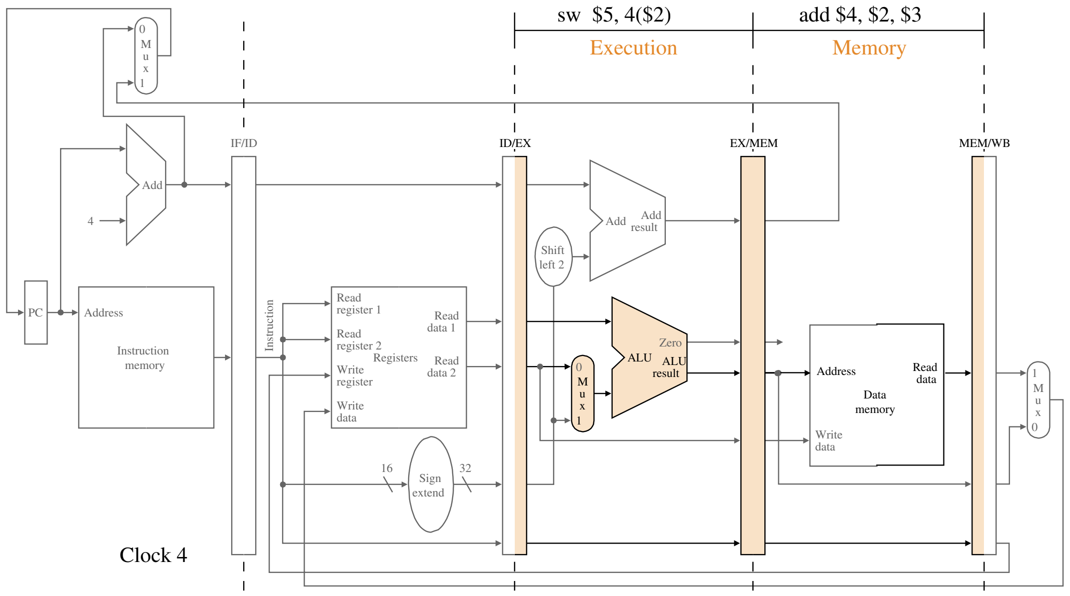
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6
add $4, $2, $3	Instruction fetch	Instruction decode	Execution	Data access	Write back	
sw $5, 4($2)		Instruction fetch	Instruction decode	Execution	Data access	Write back

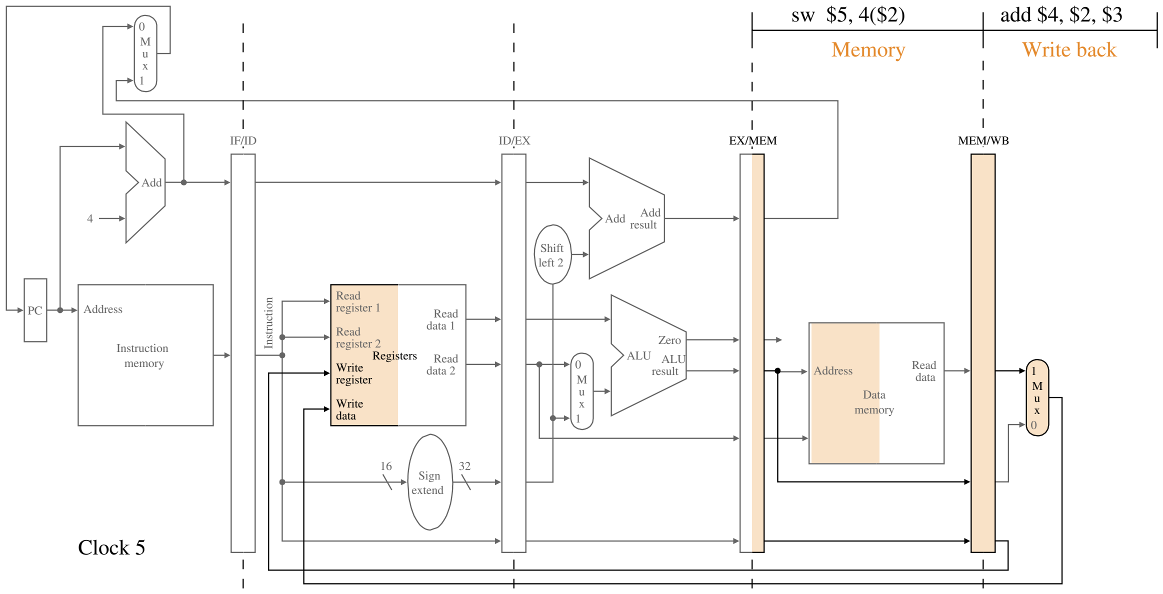
6.9 See following figures.



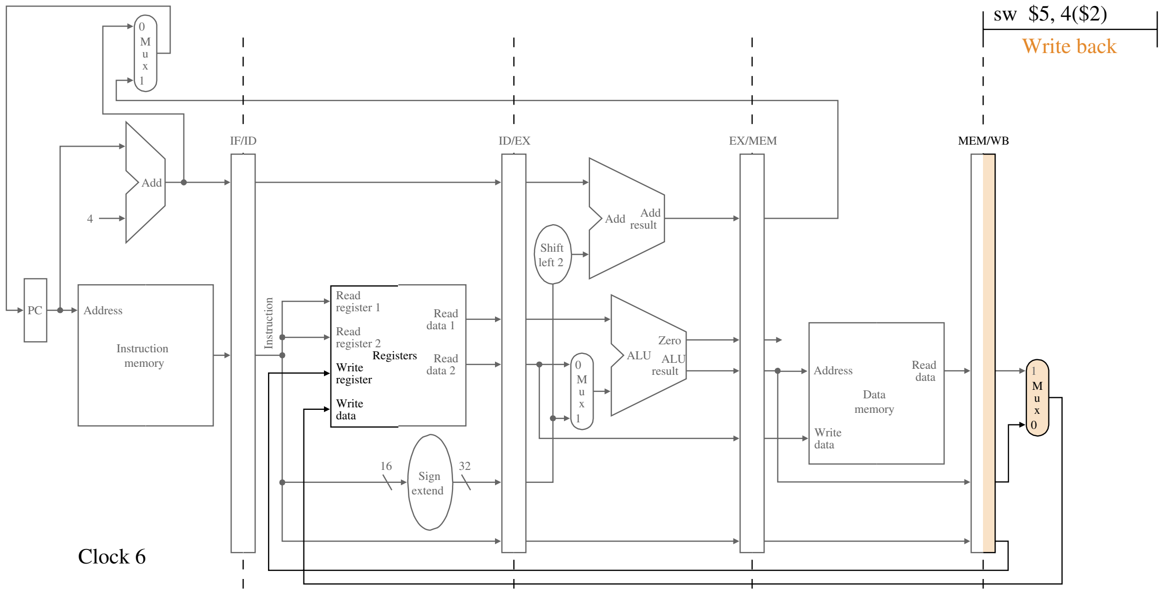



sw $5, 4($2)
add $4, $2, $3
Instruction decode
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
Add
4
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
Clock 3


sw $5, 4($2)
add $4, $2, $3
Execution
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
Add
4
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
Clock 4



Clock 5



Clock 6

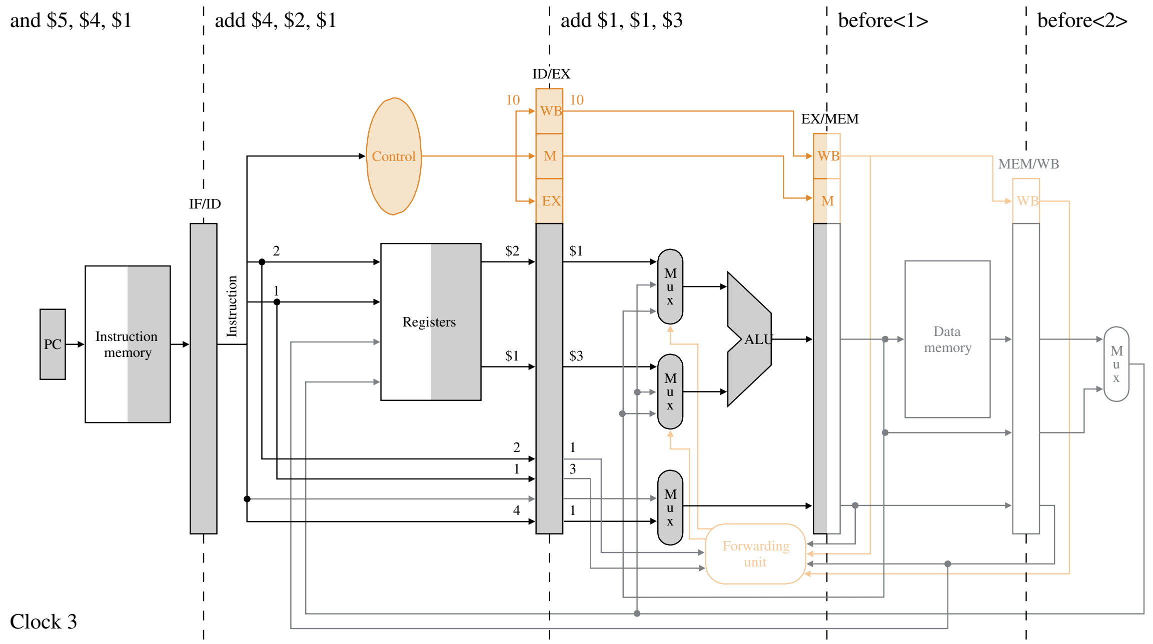
6.10 Consider each register:

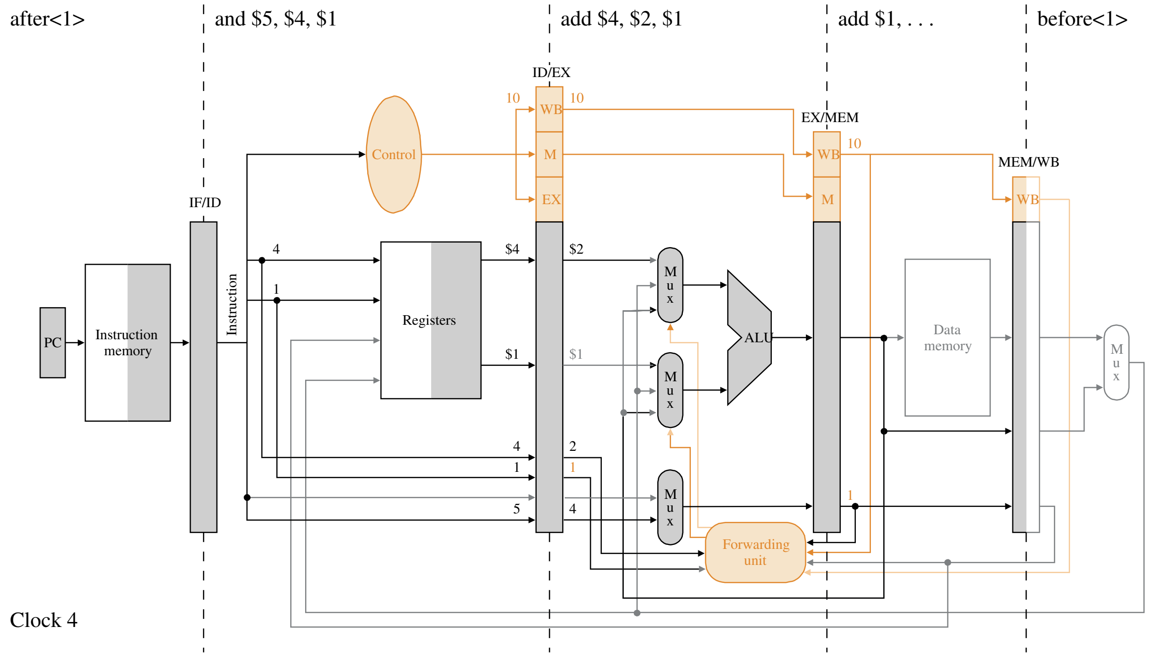
- IF/ID holds PC + 4 (32 bits) and the instruction (32 bits) for a total of 64 bits.
- ID/EX holds PC + 4 (32 bits), Read data 1 and 2 (32 bits each), the sign-extended data (32 bits), and two possible destinations (5 bits each) for a total of 138 bits.
- EX/MEM holds PC target if branch (32 bits), Zero (1 bit), ALU Result (32 bits), Read data 2 (32 bits), and a destination register (5 bits) for a total of 102 bits.
- MEM/WB holds the data coming out of memory (32 bits), ALU Result (32 bits), and the destination register (5 bits) for a total of 69 bits.

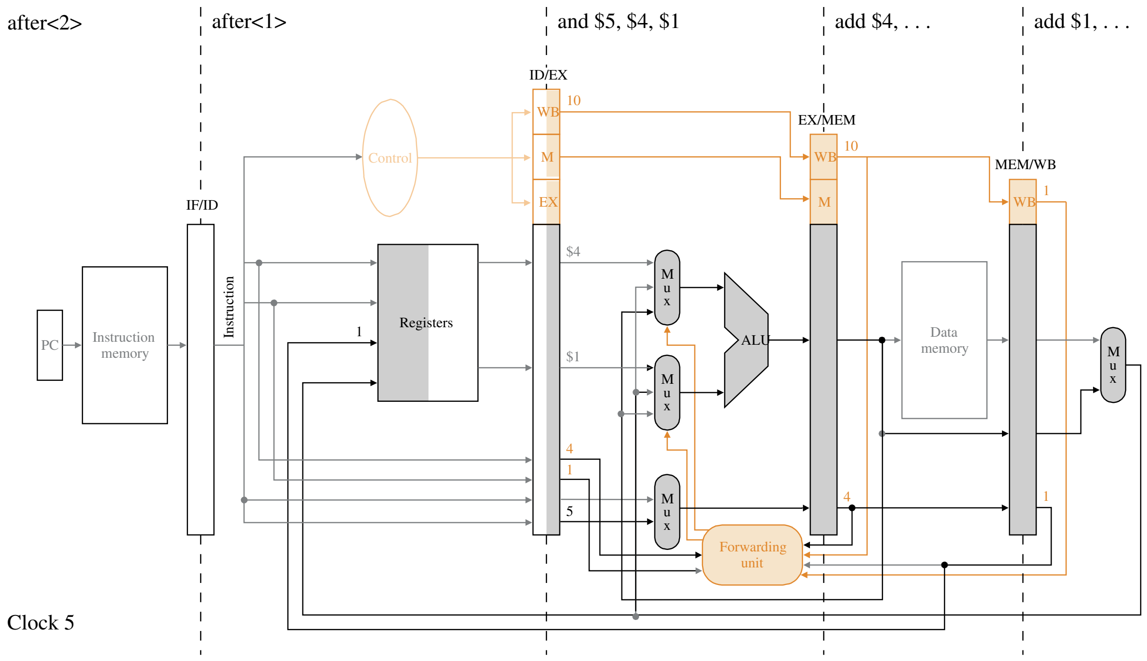
6.11 No solution provided.

6.15 No solution provided.

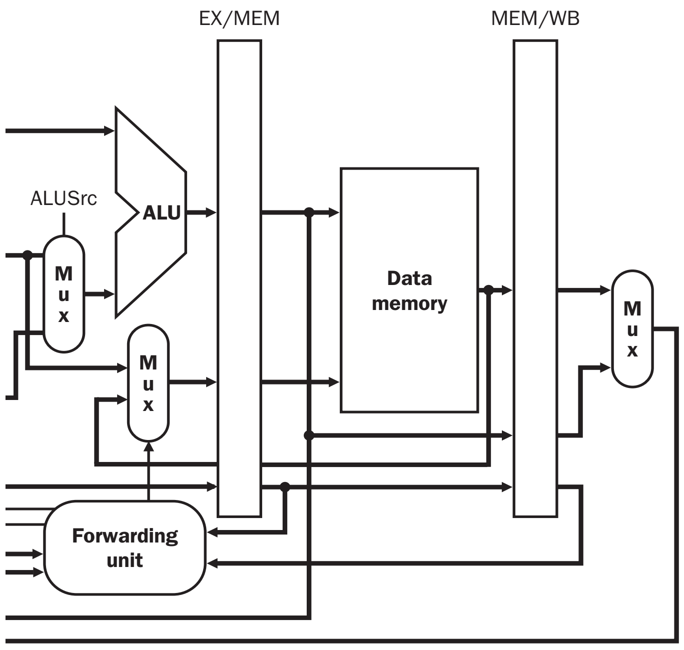
6.16 See following figures.






after<2>
after<1>
and $5, $4, $1
add $4, . . .
add $1, . . .
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
10
1
PC
Instruction memory
Instruction
Registers
$4
$1
4
1
5
M u x
ALU
Data memory
Forwarding unit
Clock 5

6.20 This solution checks for the `lw-sw` combination when the `lw` is in the MEM stage and the `sw` is in the EX stage.



```
if      (ID/EX.MemWrite and// sw in EX stage?
        EX/MEM.MemRead and// lw in MEM stage?
        (ID/EX.RegisterRt = EX/MEM.RegisterRd) and
        // same register?
        (EX/MEM.RegisterRd ≠ 0))// but not r0?

then
        Mux = 1    // forward lw value

else
        Mux = 0    // do not forward
```

The following solution checks for the `lw-sw` combination when the `lw` is in the WB stage and the `sw` is in the MEM stage.

```
if      (EX/MEM.MemWrite and      // sw in MEM stage?
        (MEM/WB.MemToReg = 1) and MEM/WB.RegWrite and
                                  // lw in WB stage?
        (EX/MEM.RegisterRd = MEM/WB.RegisterRd) and
                                  // same register?
        (MEM/WB.RegisterRd ≠ 0))  // but not r0?
```

```
    then
            Mux = 1                     // forward lw value

    else
            Mux = 0                     // do not forward
```

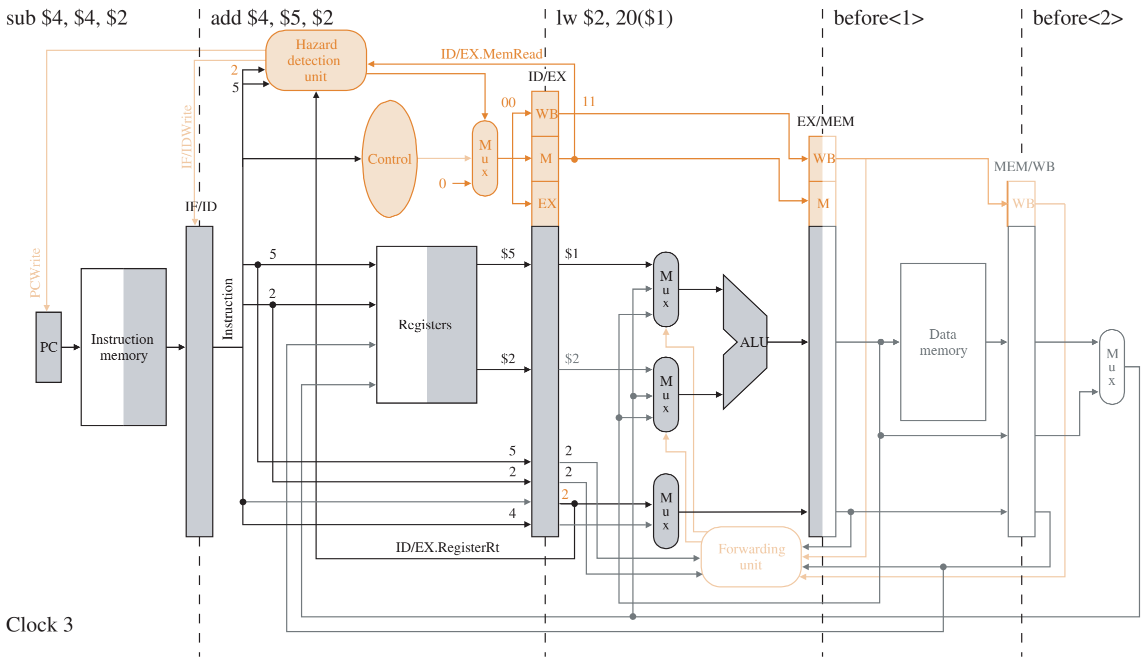
For this solution to work, we have to make a slight hardware modification: We must be able to check whether or not the `sw` source register (`rt`) is the same as the `lw` destination register (as in the previous solution). However, the `sw` source register is not necessarily available in the MEM stage. This is easily remedied: As it is now, the `RegDst` setting for `sw` is `X`, or "don't care" (refer to Figure 6.25 on page 401). Remember that `RegDst` chooses whether `rt` or `rd` is the destination register of an instruction. Since this value is never used by a `sw`, we can do whatever we like with it—so let's always choose `rt`**.** This guarantees that the source register of a `sw` is available for the above equation in the MEM stage (`rt` will be in `EX/MEM.WriteRegister`).

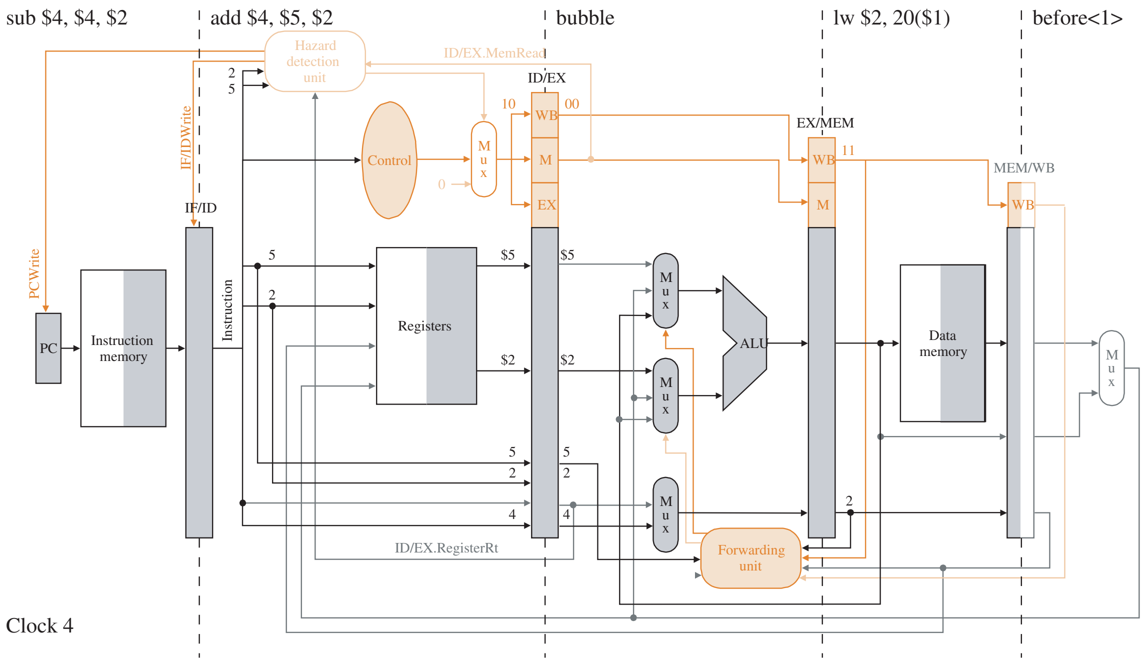
A `lw`–`sw` stall can be avoided if the `sw` offset register (`rs`) is not the `lw` destination register or if the `lw` destination register is `r0`.

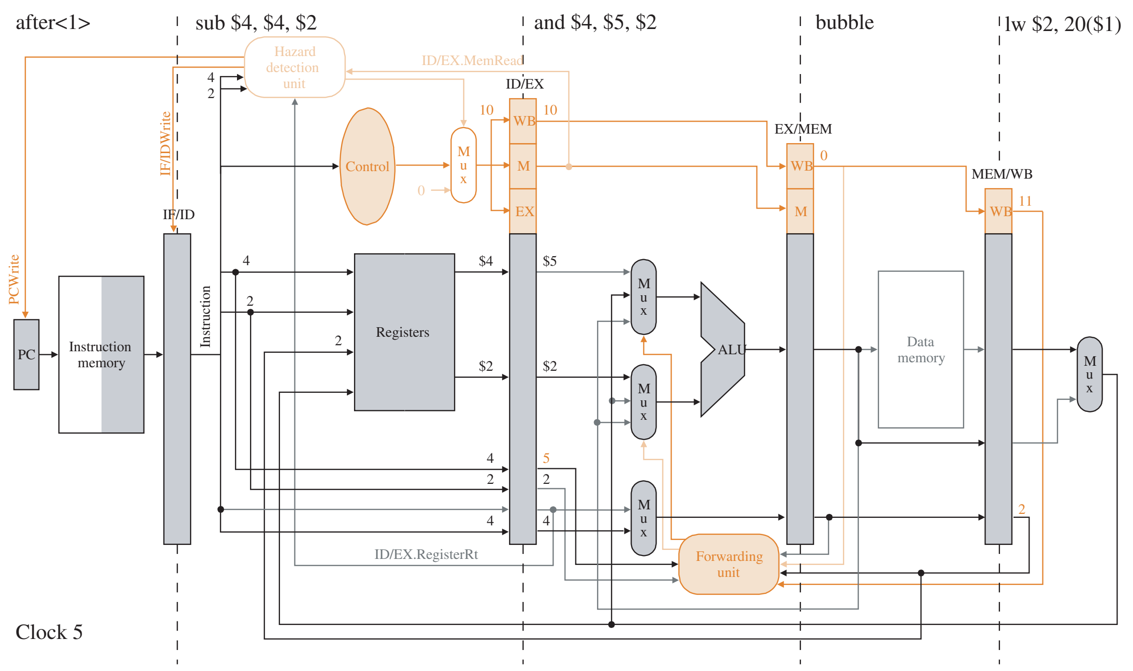
```
    if    ID/EX.MemRead and           // lw in EX stage?
          ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
                                      // same register?
          (ID/EX.RegisterRt = IF/ID.RegisterRt)) and
                                      // but not...
          not (IF/ID.MemWrite and     // sw in ID stage?
          (ID/EX.RegisterRt = IF/ID.RegisterRs)) and
                                      // same register?
          (ID/EX.RegisterRt ≠ 0)      // register r0?

    then
          Stall the pipeline
```

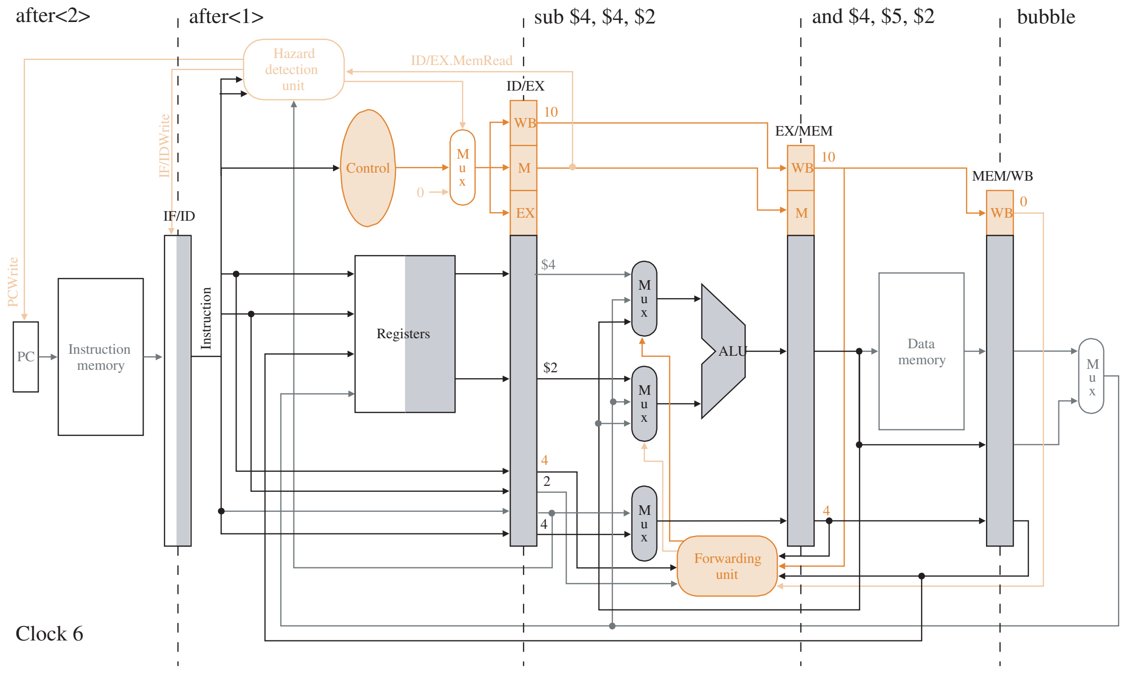
Note that `IF/ID.MemWrite` is a new signal signifying a store instruction. This must be decoded from the opcode. Checking that the `lw` destination is not `r0` is not done in the stall formula on page 413. That is fine. The compiler can be designed to never emit code to load register `r0`, or an unnecessary stall can be accepted, or the check may be added, as is done here.

6.24 See following figures.






after<1>
sub $4, $4, $2
and $4, $5, $2
bubble
lw $2, 20($1)
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/ID
IF/IDWrite
PCWrite
Control
PC
Instruction memory
Instruction
Registers
Data memory
ALU
Forwarding unit
ID/EX.RegisterRt
WB
M
EX
Clock 5


after<2>
after<1>
sub $4, $4, $2
and $4, $5, $2
bubble
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/ID
IF/IDWrite
PCWrite
Control
PC
Instruction memory
Instruction
Registers
Data memory
ALU
Forwarding unit
WB
M
EX
Clock 6

6.25 No solution provided.

6.26 The memory address can now come straight from Read data 1 rather than from the ALU output; this means that the memory access can be done in the EX stage, since it doesn't need to wait for the ALU. Thus, the MEM stage can be eliminated completely. (The branch decision can be taken care of in another stage.) The instruction count will go up because some explicit addition instructions will be needed to replace the additions that used to be implicit in the loads and stores with nonzero displacements. The CPI will go down because stalls will no longer be necessary when a load is followed immediately by an instruction that uses the loaded value. Forwarding will now resolve this hazard just like for ALU results used in the next instruction.

6.27 An `addm` instruction needs six steps:

1. Fetch instruction.
2. Decode instruction and read registers.
3. Calculate memory address of memory operand.
4. Read memory.
5. Perform the addition.
6. Write the register.

This means we will need an extra pipeline stage (and all other instructions will go through an additional stage). This can increase the *latency* of a single instruction, but not the *CPU throughput* because instructions will still be coming out of the pipeline at a rate of 1 per cycle (if no stalls). In fact, a longer pipeline will have a greater potential for hazards and longer penalties for those hazards, which will decrease the *throughput*. Moreover, the additional hardware for new stages, hazard detection, and forwarding logic will increase the cost.

6.28 No solution provided.

6.31

```
slt  $1,  $8, $9
movz $10, $8, $1
movn $10, $9, $1
```

6.32 The reason why sequences of this sort can be higher performance than versions that use conditional branching, even if the instruction count isn't reduced, is that there is no control hazard introduced, unlike with branching. Branching generally introduces stalls, which waste cycles, because the normal sequential flow of execution is disrupted. The new instructions, in contrast, are part of the normal sequential execution stream. The only hazards they introduce are normal data hazards, as with any ALU instruction, and those can be completely conquered using forwarding.