Homework #3 Solutions (abridged)

1)

	RegDst=0	ALUSrc=0	MemtoReg=0	Zero=0
Loads	Y	N	N	Y
Stores	Y	N	Y	Y
R-type	N	Y	Y	Y
Branch	Y	Y	Y	N

2)

	RegDst=1	ALUSrc=1	MemtoReg=1	Zero=1
Loads	N	Y	Y	Y
Stores	Y	Y	Y	Y
R-type	Y	N	N	Y
Branch	Y	N	Y	N

3)

The gcc data from Chapter 4 has the following instruction mix:

Instruction Class	**Frequency**
Loads	23%
Stores	13%
R-type	43%
Jump/branch	21%

Here are the CPIs:

Instruction Class	**$CPI_{500\ MHz}$**	**$CPI_{750\ MHz}$**
Loads	5	6
Stores	4	5
R-type	4	4
Jump/branch	3	3

$CPI_{500\ MHz} = (0.23 \times 5) + (0.13 \times 4) + (0.43 \times 4) + (0.21 \times 3) = 4.02$

$CPI_{750\ MHz} = (0.23 \times 6) + (0.13 \times 5) + (0.43 \times 4) + (0.21 \times 3) = 4.38$

The 750 MHz is faster by: (750 / 4.38) / (500 / 4.02) = 1.3767

4)

The second instruction is dependent upon the first ($2).

The third instruction is dependent upon the first (`$2`).
The fourth instruction is dependent upon the first (`$2`) and second (`$4`).

All of these dependencies will be resolved via forwarding.

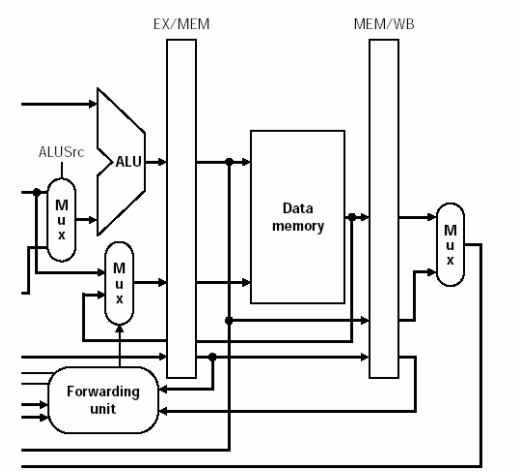
5)

The forwarding unit is seeing if it needs to forward. It is looking at the instructions in the fourth and fifth stages and checking to see whether they intend to write to the register file and whether the register written is being used as an ALU input. Thus, it is comparing:
8 = 4? 8 = 1? 9 = 4? 9 = 1?

6)

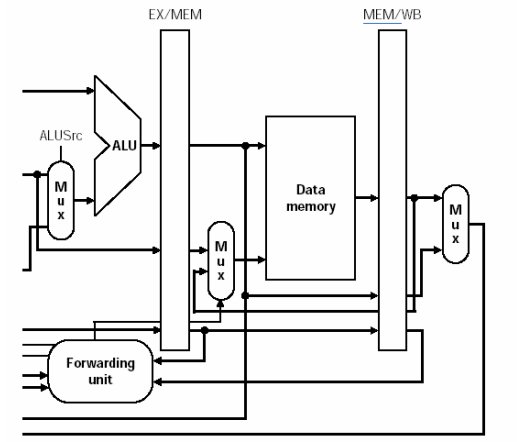
There are multiple solutions for problem number 6. Here are the most common two solutions.

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 6.28 on page 469). 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`). (See Figure 6.30 on page 470.)

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 490. 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.