**PRE-LAB**

(i) Review the pipelined datapath shown on the last page and briefly explain the role of each pipeline stage.

Fetch- fetches instruction

Decode- decodes instruction and sets up instruction

Execute- executes instruction

Write back- write's results back into register or memory

(ii) The following is a list all of the control signals that you need to derive during the decode stage of the pipeline. Based on your understanding of the pipelined datapath, specify which pipeline stage (i.e., DEC, EXE, or WB) each signal is used in. For example, the control signal used to indicate writing to the register file (*reg_rw*) needs to be used in the WB stage. (See the output ports of id.v)

| Control signals | Pipeline Stage | Control signals | Pipeline Stage |
|---|---|---|---|
| reg_src1 | DEC | mem_wr | WB |
| reg_src2 | DEC | reg_wr | WB |
| reg_dst | DEC | imm_sel | DEC |
| imm | DEC | data_sel | EX |
| add_sub | EX | dp_en | EX |
| mem_rd | EX | halt | WB |



(iii)Specify an opcode that you will use for a NOP instruction.
Imma use 1001 because that's the next

(iv)For each instruction type, the ID block (id.v) should generate proper control signals. Fill out the following table.

| | load | move | sub | add | disp | halt | subi | addi | store | nop |
|---|---|---|---|---|---|---|---|---|---|---|
| add_sub | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| mem_rd | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_wr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| reg_wr | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| imm_sel | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| data_sel | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| dp_en | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| halt | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

(v) Review the program you used for lab 5 and determine if it would present any data hazards. If it does, then re-write your program so that it no longer has any data hazards. You need to include at least one NOP instruction, but feel free to use more if you need to.

Note that, in previous labs, we put the data at address location 48. This was done to separate the data from the instructions of your program, given that they were stored in the same memory. For this lab, since we're actually instantiating a separate memory for data, we could avoid this step and have our data start from data address 0. But to keep things consistent and minimize change from the previous labs, we'll stick with the data starting at address 48.

```
2000 // sub r0, r0, r0
2111 // sub r1, r1, r1
7006 // addi r0, r0, 6
2222 // sub r2, r2, r2
3000 // add r0, r0, r0 (12)
2333 // sub r3, r3, r3
3000 // add r0, r0, r0 (24)
9000 // NOP stall 1
3000 // add r0, r0, r0 (48)
9000 // NOP stall 2
0100 // load r1, r0 (48)
7001 // addi r0, r0, 1
9000 // NOP stall 3
0200 // load r2, r0 (49)
7001 // addi r0, r0, 1
9000 // NOP stall 4
0300 // load r3, r0 (50)
3221 // add val1 and val2
9000 // NOP stall 5
8032 // store r2 at r3 value
0030 // load r3 address value in r0
3323 // add r3 and r2
4000 // display r0 to prove the store works
4030 // display r3 (now the sum of r3,r2,r1)
5000 // HALT!
```