I have attached multiple Verilog files since I built multiple modules and then integrated them together. TQ4_HW6.v is the top level file that combines everything together. All the attached Verilog files are necessary for the functioning of the processor. I also attached the imem.v, dmem.v, imem.mif, and dmem.mif files along with the Quartus project file just to be safe. I built separate modules for the fetch, decode, execute, memory and write back stage. The F/D registers were added in the fetch module and so on until the M/W register was added to the memory module. There is no register in the write back stage. I tried to use my pipelined multiplier built in HW 4 (multDiv.v) and tried to integrate it with stall logic (processMulDiv.v) but it didn't work as expected so I ended up using the lpm_mul and lpm_div megafunctions in the execute stage. The design with these megafunctions was tested with a clock period of 5 ns in functional simulation and it worked; although a timing simulation would give a better idea of how slow the clock needs to be. The ldStall.v contain the stall logic for a load instruction which is not followed by a save instruction, and bypassCtrl.v contains all the bypass logic for avoiding data hazards. Stalling was also done for branch and jump instructions. The bypass, load stall, and the 5 stage modules were integrated together in pipeProcessor.v file which is then called in the top level design entity file.

A load instruction which is not followed by a store instruction leads to a 3 cycle stall in my processor. Branch instructions also lead to 3 cycle stalls while jal and jr instructions lead to 2 cycle stalls. There is no stall for the simple jump instruction since it is detected in the fetch stage. Each stage (apart from write back stage which doesn't have any memory elements) takes 1 cycle. The imem module and the dmem module are given the inverted clock as input to ensure that the fetch and memory stage only take 1 cycle.

Static branch prediction is used by predicting that branches are not taken, and if they are taken then the instructions are flushed by clearing the appropriate register similar to what is suggested in Figure 1 (from lecture slides). Similar flushing is done for jal and jr instructions.

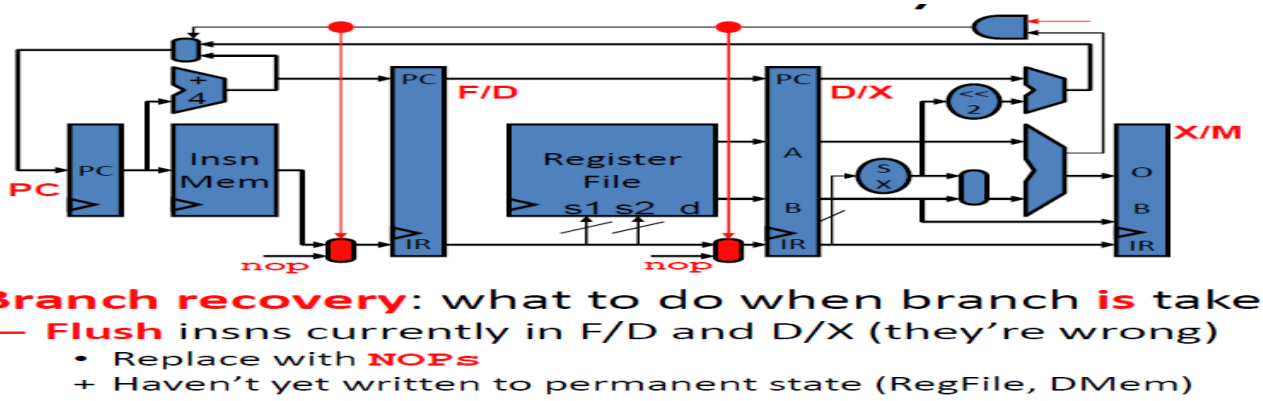

Figure 1: Flushing when branch taken

Stalling for the load instruction which is not followed by a store is done similar to the way shown in Figure 2. The stall logic equation in the figure is slightly wrong because the parentheses are mismatched due to a typo. The equation was corrected and extra logic was added for detecting load at memory stage and not store at the execute stage.

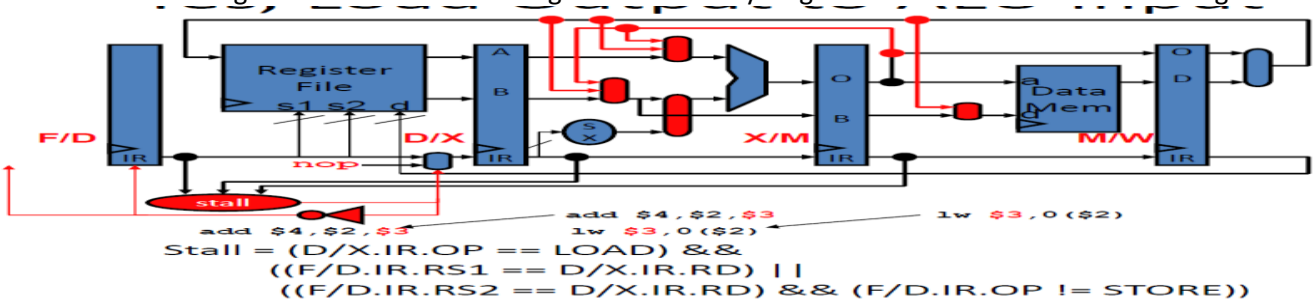

Figure 2: Load stall logic

Bypassing was done similar to the way shown in Figure 3 and Figure 4. Some extra logic was added such as bypassing from write back stage to decode stage when appropriate.

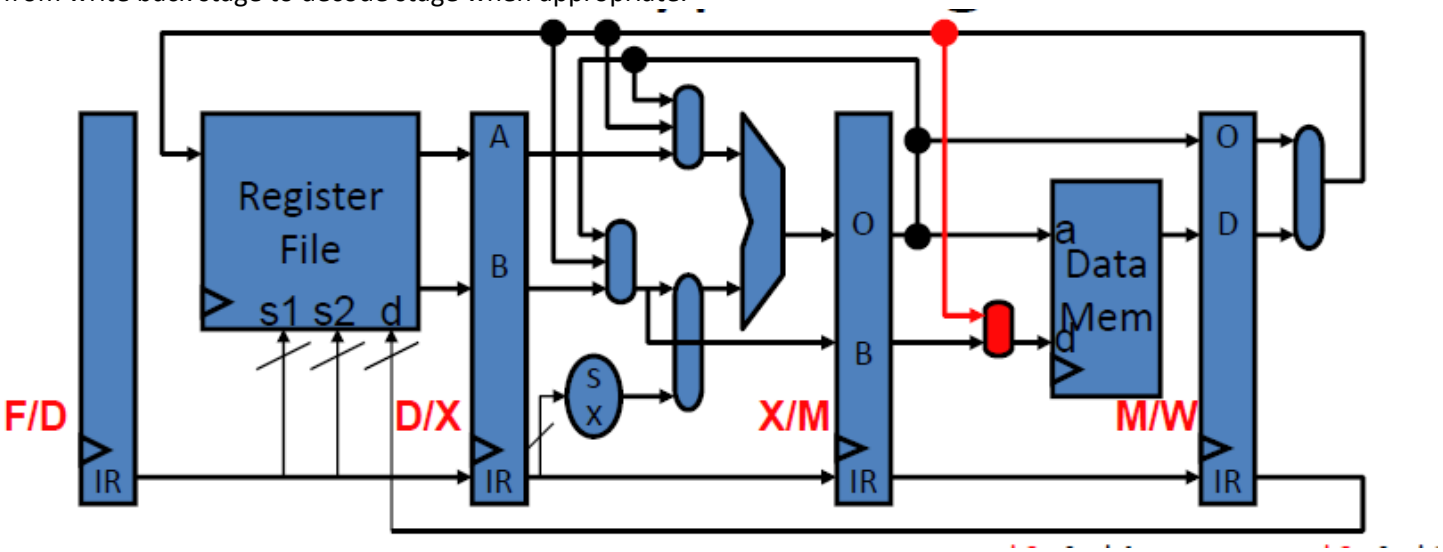

Figure 3: Bypass mechanism

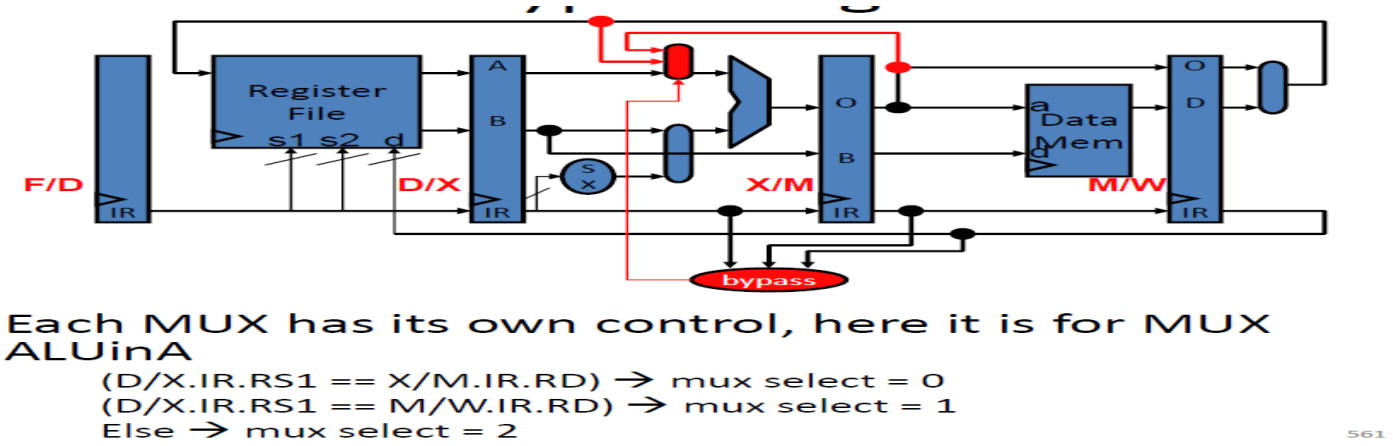

Figure 4: Bypassing mechanism

The following sample assembly code was used for testing and the processor worked for it.

Sample Assembly code used for testing:

```
.text
main:

addi $r1, $r0, 1
blt $r1, $r2, 21
addi $r2, $r1, 2
sub $r3, $r2, $r1
and $r3, $r2, $r3
or $r4, $r1, $r2
sll $r5, $r4, 1
sra $r6, $r5, 1
add $r7, $r6, $r5
lw $r8, 0($r7)
sw $r8, 0($r1)
 lw $r9, 0($r1)
```

```
addi $r10, $r9, 0
addi $r11, $r9, 2
lw $r12, 0($r1)
add $r13, $r12, $r11
add $r13, $r12, $r11
add $r13, $r12, $r11
add $r13, $r12, $r11
lw $r14, 0($r0)
 jr $r14
jal 2
addi $r27, $r0, 0
jal 25
addi $r1, $r5, 0
addi $r27, $r31, 0
mul $r20, $r7, $r6
div $r21, $r7, $r6
addi $r22, $r21, 5
.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```