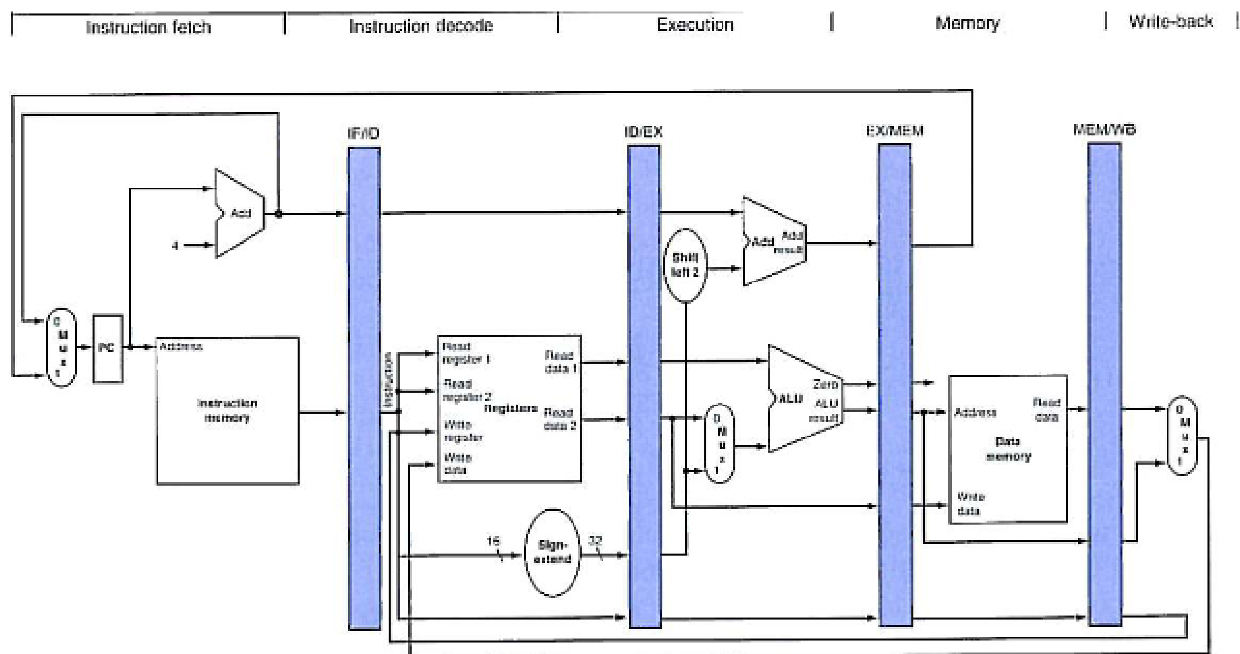
Everett Salley
4/18/2012
EEL4713 Assignment #5

Introduction:

In this lab, we took the single-cycle MIPS processor we had designed in the previous labs and pipelined it. The main reason for pipelining is to improve the overall efficiency of the processor by allowing it to perform different stages of multiple instructions at the same time as opposed to one instruction per clock cycle. By sectioning off the execution of an instruction into multiple registered stages, the amount of time required to execute each stage is smaller than the overall time required to execute an instruction. As a result, one can theoretically increase the overall clock speed of the processor by evenly dividing the delay amongst the different pipeline stages. The tradeoff is that this method introduces numerous hazards and properly dealing with said hazards increases the overall design complexity. Below is a simplified picture showing the 5 pipeline stages in our MIPS processor:



The Instruction Fetch (IF) stage as its name implies fetches the instruction from the instruction memory using the program counter as a pointer to the address. The Instruction Decode (ID) stage decodes the instruction to generate the proper control signals and outputs the requested register data from the register file. The Execute (EX) stage is where the actual arithmetic required by the instruction is performed. The Memory Access (MEM) stage is where data is either written to or read from the data memory. Finally the Write Back (WB) stage is where data is written back to the register file.

As was mentioned before, the introduction of a pipeline also introduces numerous hazards that unless dealt with will break the operation of a program. The problem arises from the fact that there is now a latency from when an instruction is first fetched to when it is fully executed and results are stored. A hazard occurs whenever the program needs data that was a result of a previous operation, but has not yet been stored in either data memory or the register file. Since this data propagates throughout the pipeline registers, the most efficient way of

handling such a hazard is to simply forward it (ie don't wait until it is written to the register file) to the requested location. However, there are some instances where the data is simply not available to be forwarded (fetching data from memory for instance) and the only possible course of action is to "stall" the pipeline until the data is available. Stalling basically consists of accessing the same instruction repeatedly while asserting the control signals to zero for the redundant instructions. The result is that the processor continues executing the instructions before the stall, but performs no operation (nops) until the stall is over. The combination of forwarding and stalling makes the pipelined processor viable. The Hazard Table on the following page shows the various hazards encountered and how to deal with them.

My hazard table differed slightly from the one that was provided to use in class. For instance, my register file is written to on the falling clock edge, which means that the same data can be written to and read from a particular register in the same clock cycle. This completely eliminates the need for the forwarding muxes in the ifid stage. As a result, I only have to check for hazards over 2 stages and my hazard table is significantly reduced in many areas.

	R	AI	L	S	JR	LUI	B	JAL
R	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3)	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4)	1a: idex.rd=ifid.rs(stall) 2a: exmem.rd=ifid.rs(8)		1a: idex.rd=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rd=ifid.rs(8) =ifid.rt(10)	
AI	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3)	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: idex.rt=ifid.rs(stall) 2a: exmem.rt=ifid.rs(8)		1a: idex.rt=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rt=ifid.rs(8) =ifid.rt(10)	
L	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3)	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: ifid.rt=pre.rs(stall) 2a: idex.rt=pre.rs(stall)		1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: idex.rt=pre.rs(stall) =pre.rt(stall)	
S								
JR								
LUI	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14)	1a: idex.rt=ifid.rs(17) 2a: exmem.rt=ifid.rs(18)		1a: idex.rt=ifid.rs(17) =ifid.rt(19) 2a: exmem.rt=ifid.rs(18) =ifid.rt(20)	
B								
JAL	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23)	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24)	1a: idex.31=ifid.rs(27) 2a: exmem.31=ifid.rs(28)		1a: idex.31=ifid.rs(27) =ifid.rt(29) 2a: exmem.31=ifid.rs(28) =ifid.rt(30)	

(1).	forward exmem.aludata to rs of alu
(2).	forward exmem.aludata to rt of alu
(3).	forward WBdata to rs of alu
(4).	forward WBdata to rt of alu
(5).	forward WBdata to rs of idex reg
(6).	forward WBdata to rt of idex reg
(8).	forward exmem.aludata to rs of idex reg
(10).	forward exmem.aludata to rt of idex reg
(11).	forward ui from exmem to rs of alu
(12).	forward ui from exmem to rt of alu
(13).	forward ui from memwb to rs of alu
(14).	forward ui from memwb to rt of alu
(15).	forward ui from memwb to rs of idex reg
(16).	forward ui from memwb to rt of idex reg

(17).	forward ui from idex to rs of idex reg
(18).	forward ui from exmem to rs of idex reg
(19).	forward ui from idex to rt of idex reg
(20).	forward ui from exmem to rt of idex reg
(21).	forward PC+4 from exmem to rs of alu
(22).	forward PC+4 from exmem to rt of alu
(23).	forward PC+4 from memwb to rs of alu
(24).	forward PC+4 from memwb to rt of alu
(25).	forward PC+4 from memwb to rs of idex reg
(26).	forward PC+4 from memwb to rt of idex reg
(27).	forward PC+4 from idex to rs of idex reg
(28).	forward PC+4 from exmem to rs of idex reg
(29).	forward PC+4 from idex to rt of idex reg
(30).	forward PC+4 from exmem to rt of idex reg

Book Questions: (USING THE BLUE EDITION)

4.12.1)

-pipeline speed is derived by taking the largest latency found in any of the 5 stages
-non-pipelined speed is equivalent to the sum of all latencies

a) pipelined: 500ps
non-pipelined: 1650ps

b) pipelined: 200ps
non-pipelined: 800ps

4.12.2)

-for a non-pipelined processor, it takes 1 cycle to fully execute an instruction, therefore the instruction latency is simply the same as the clock time
-for a pipelined processor it takes 5 cycles to fully execute any given instruction, therefore the total instruction latency is 5 times the clock cycle time

a) pipelined: 5*500 = 2500ps
non-pipelined: 1650ps

b) pipelined: 5*200 = 1000ps
non-pipelined: 800ps

4.12.3)

-To increase the clock speed one would ideally split up the stage with the highest latency.

a) MEM stage; new clock cycle = 400ps
b) IF stage; new clock cycle = 190ps

4.20.1)

There are three possible data dependencies here
i) (RAW) Instr needs to read data from register file before it is written
ii) (WAW) Instr writes to reg that was previously written to
iii) (WAR) Instr writws to reg that was previously read from

a)
Instr1 & Instr2 (WAR)
Instr3 & Instr4 (WAR)
Instr1 & Instr3 (RAW)
Instr2 & Instr3 (RAW)
Instr2 & Instr4 (RAW)
Instr3 & Instr4 (RAW)
Instr1 & Instr3 (WAW)

b)
Instr1 & Instr2 (WAR)
Instr1 & Instr2 (RAW)
Instr1 & Instr3 (WAR)
Instr1 & Instr3 (WAW)
Instr2 & Instr4 (WAR)
Instr2 & Instr3 (WAR)
Instr3 & Instr4 (WAR)
Instr3 & Instr4 (RAW)

4.20.2)

For the 5 stage pipeline we only have to worry about read after write. Most cases can be avoided by using forwarding but some instances like a load followed by a read are unavoidable hazards. Therefore, part a has no hazards to worry about (assuming the reg file is falling edge triggered. part b has one data hazard between instr 3 and 4.

4.24.1)

-accuracy is determined by the number of correct branches taken divided by the total number of branch decisions

a) always take:75% (3/4)
always nt: 25% (1/4)

b) always take:60% (3/5)
always nt: 40% (2/5)

4.24.2)

-starts off in predict branch not taken state and moves around state machine depending on correctness of decisions. Considering only the first 4 branch decisions...

a) 0% (all predictions are incorrect)
b) 25% (3rd prediction is correct)

4.24.3)

-eventually a steady state is reached in the prediction pattern such that it begins every iteration in the same state

a) eventually it settles in the top left state and the accuracy is 75%
pattern is T,T,NT,T -> correct, correct, incorrect, correct

b) eventually it settles in the bottom right state and the accuracy is 40%
pattern is T,T,T,NT,NT -> incorrect, correct, correct, incorrect, incorrect

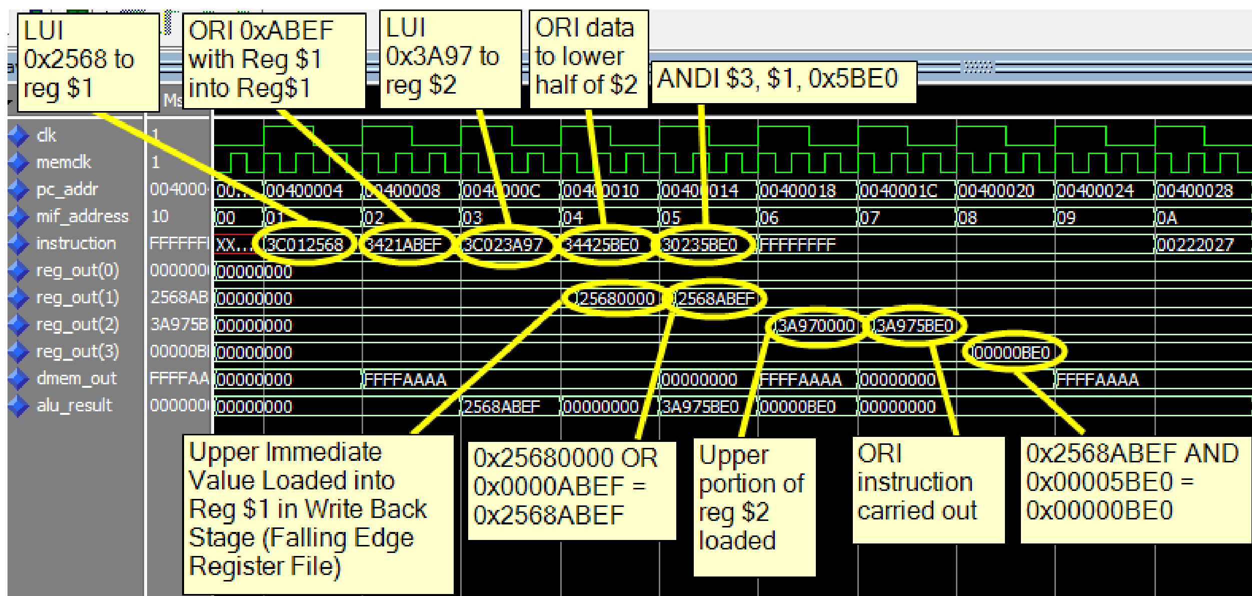
Demonstration of Instructions:

Many changes were made to the original MIPS processor to make it pipelined so it is important to show that it still functions as expected. This section is dedicated to showing the complete functionality of all 29 of the base MIPS instructions by individually testing each one (in some cases more than once) with custom test programs. The first such test program is designed to examine all of the logical instructions in the MIPS instruction set and the code is given below:

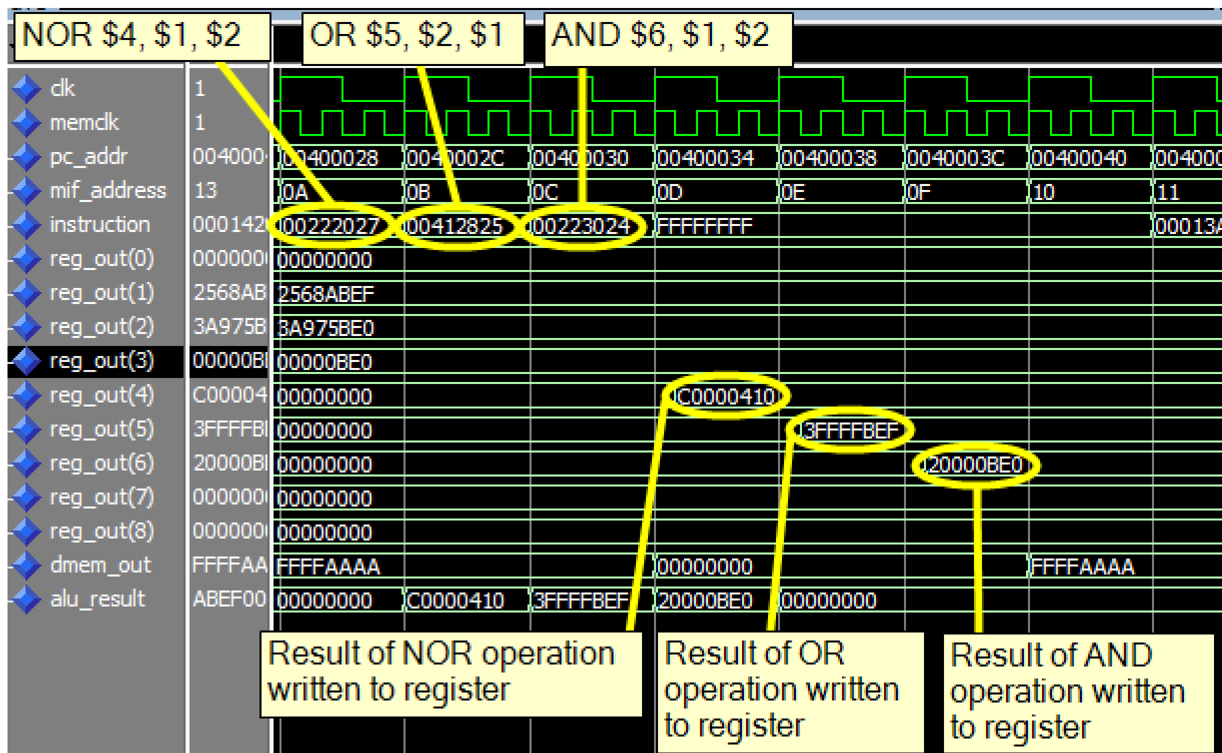
```
lui  $1, $0, 0x2568
ori  $1, $1, 0xABEF
lui  $2, $0, 0x3A97
ori   $2, $2, 0x5BE0
andi $3, $1, 0x5BE0

nor  $4, $1, $2
or    $5, $2, $1
and  $6, $1, $2

sll   $7, $1, 8
sll   $7, $1, 16
srl   $8, $1, 8
srl   $8, $1, 16
```

The first instructions to be examined are the load upper immediate, OR immediate, and the AND immediate instructions because they are so widely used for initializing registers. The annotated waveform provided on the next page shows the execution of the first five instructions in this program which initializes the registers. Now would be a good time to point out the differences that the pipeline introduces to interpreting the waveform results. In the non-pipelined version the results of any instruction was available on the clock cycle immediately following the instruction. Here that is not the case as there is a multicycle delay from when the instruction is fetched to when it is fully executed. We can examine the first instruction in more detail to see how this works.

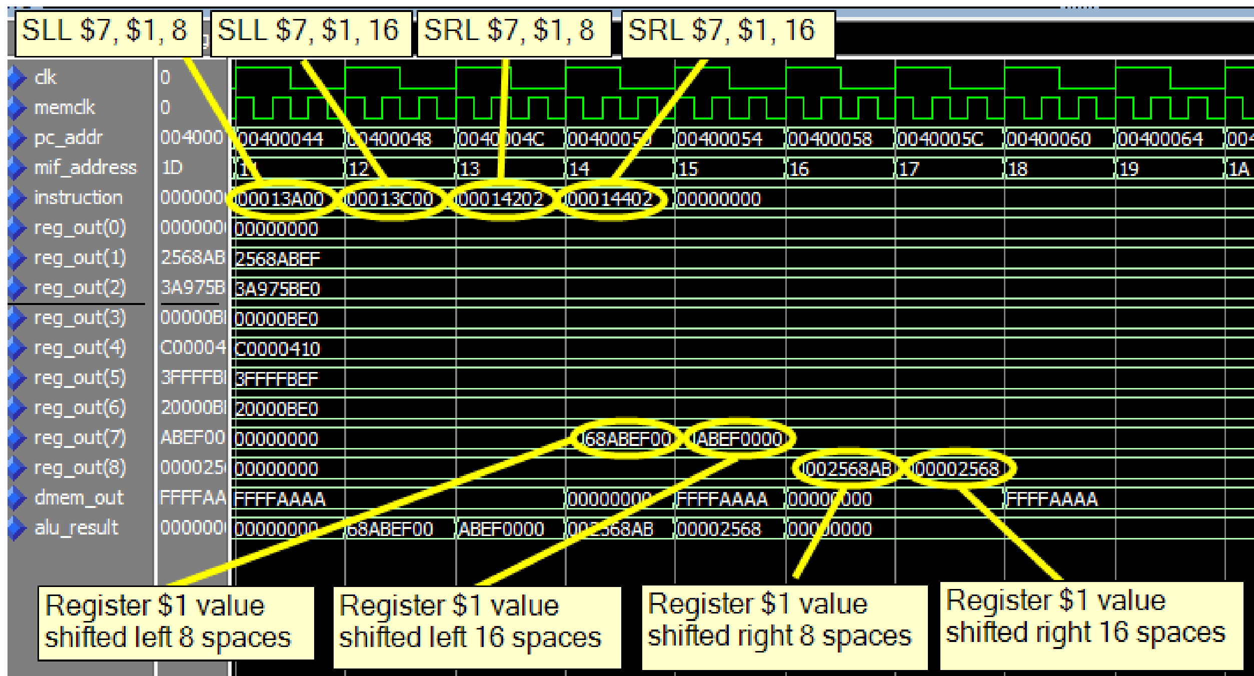
In clock cycle 1, the first instruction is being fetched from the instruction memory (instruction fetch stage). In the next clock cycle, the instruction signal (located in the instruction decode stage) now displays the instruction that was previously fetched which is then decoded and sent to the register file. Clock cycle 3 is the execute stage, where the data fetched from the register file is usually (not in this case) sent through the ALU for some sort of computation. Clock cycle 4 is the memory access stage where data is either written to, or read from data memory. Finally clock cycle 5 is the write back stage where the data is written to the destination register in the register file. Notice that in my design I used a falling edge triggered register file, so the data is stored to the register file, and thus available for reads BEFORE the next rising edge. Thus, whenever an instruction appears in the instruction field in a waveform, its results will be written to the register file after 2 rising clock edges.



The next waveform shows the operation of the basic logic instructions AND, OR, and NOR. All three cases use the input values 0x2568ABEF and 0x3A975BE0.



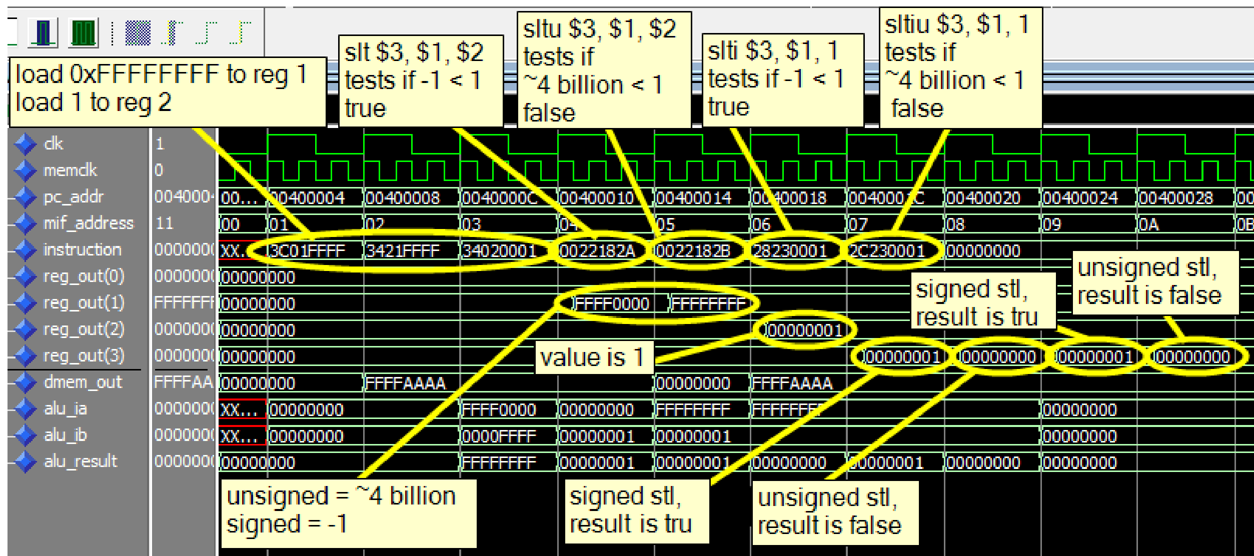
The next waveform shows the operation of the shift left logical left and right instructions. In every case it operates on the value 0x2568ABEF. Shifts of 8 and 16 are performed in both directions to be thorough.



The next program investigates every iteration of the set less than instruction. It basically works by testing the unsigned and signed version of the set less than instruction with the input -1 and 1. In the case of the signed operation, the -1 is interpreted as being less than 1 which is true and thus results in the destination register getting set to 1. In the case of the unsigned operation, the -1 is interpreted as approximately 4 billion which of course is not less than 1. Thus for the unsigned operation, the result is false and the destination register is set to 0.

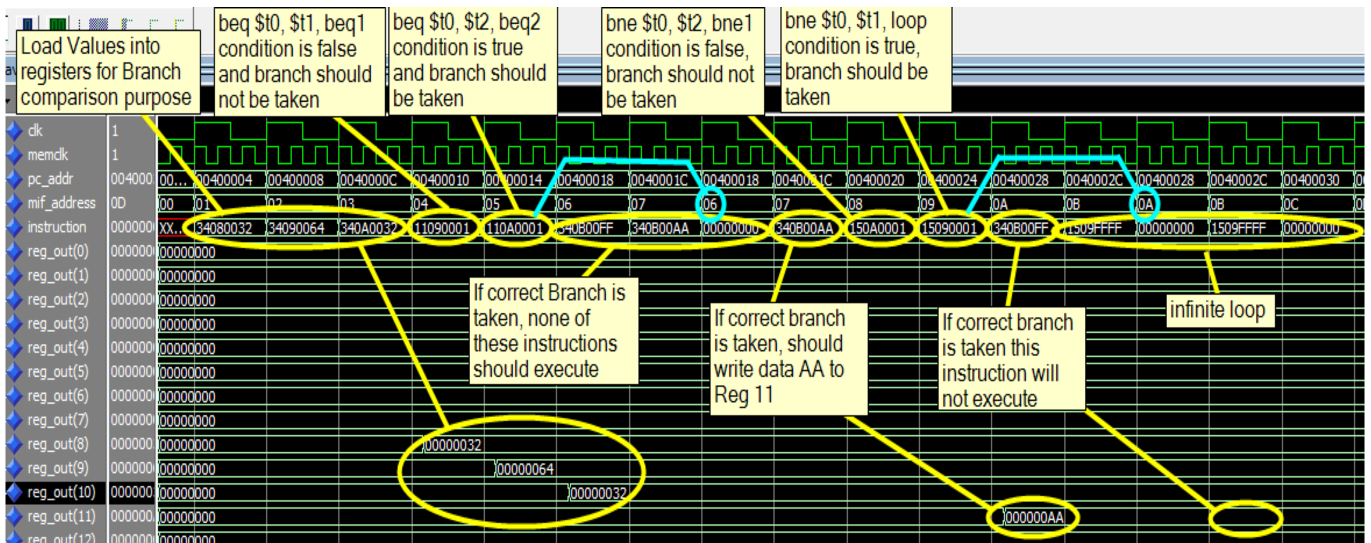
The code for this test and waveform is provided below:

```
lui     $1, $0, 0xFFFF
ori     $1, $1, 0xFFFF
ori     $2, $0, 1
slt     $3, $1, $2
sltu    $3, $1, $2
slti    $3, $1, 1
sltiu   $3, $1, 1
```



The next test will demonstrate all of the branch instructions functioning correctly. For the sake of thoroughness, it shows both when a branch is taken and when it is not. It basically works by first loading values to be compared into registers $t0, $t1 and $t2. It then proceeds by showing a case of BEQ not branching, followed by a case where it does branch. The same is done for the BNE instruction. It finally ends by looping infinitely. The code and waveform are provided below:

```
main:   ori     $t0, $0, 50
        ori     $t1, $0, 100
        ori     $t2, $0, 50
        beq     $t0, $t1, beq1
        beq     $t0, $t2, beq2
beq1:   ori     $t3, $0, 0x00FF
beq2:   ori     $t3, $0, 0x00AA
        bne     $t0, $t2, bne1
        bne     $t0, $t1, loop
bne1:   ori     $t3, $0, 0x00FF
loop:   bne     $t0, $t1, loop
```



The next test examines all three of the jump instructions. I utilize ori instructions as a way of keeping track of where the program is. Correctly functioning code should write a sequence of A, B, and C to register 1 with a value of FFFF indicating an error occurred. It starts by calling the jump instruction which skips to a JAL. The JAL correctly writes the return address to $ra but instead of using $ra, I use my own return address in register $2 equal to the "loop" label. The instruction correctly executes and the program proceeds to loop infinitely. The code and waveform are provided below:

```
main:
        ori $1, $0, 0xAAAA
        j       target
        ori $1, $0, 0xFFFF
        ori $1, $0, 0xFFFF
        ori $1, $0, 0xFFFF
        ori $1, $0, 0xFFFF
target: ori $1, $0, 0xBBBB
        jal dest
        ori $1, $0, 0xDDDD
        ori $1, $0, 0xFFFF
        ori $1, $0, 0xFFFF
loop:   beq $0, $0, loop
dest:   lui $2, $0, 0x1000
        ori $2, $2, 0x002C
        ori $1, $0, 0xCCCC
        jr $2
```

Load value AAAA into reg 1 as a marker

Jump to addr 6. skips over a bunch of instructions that would have written FFFF to reg1

Load value BBBB into reg 1 as a marker

Load value 0x1000002C into reg $2

Load value CCCC into reg 1 as marker

JR using reg $2 as address

JAL to addr 0C. Stores next addr value (addr 9 = 0x00400020) in $r

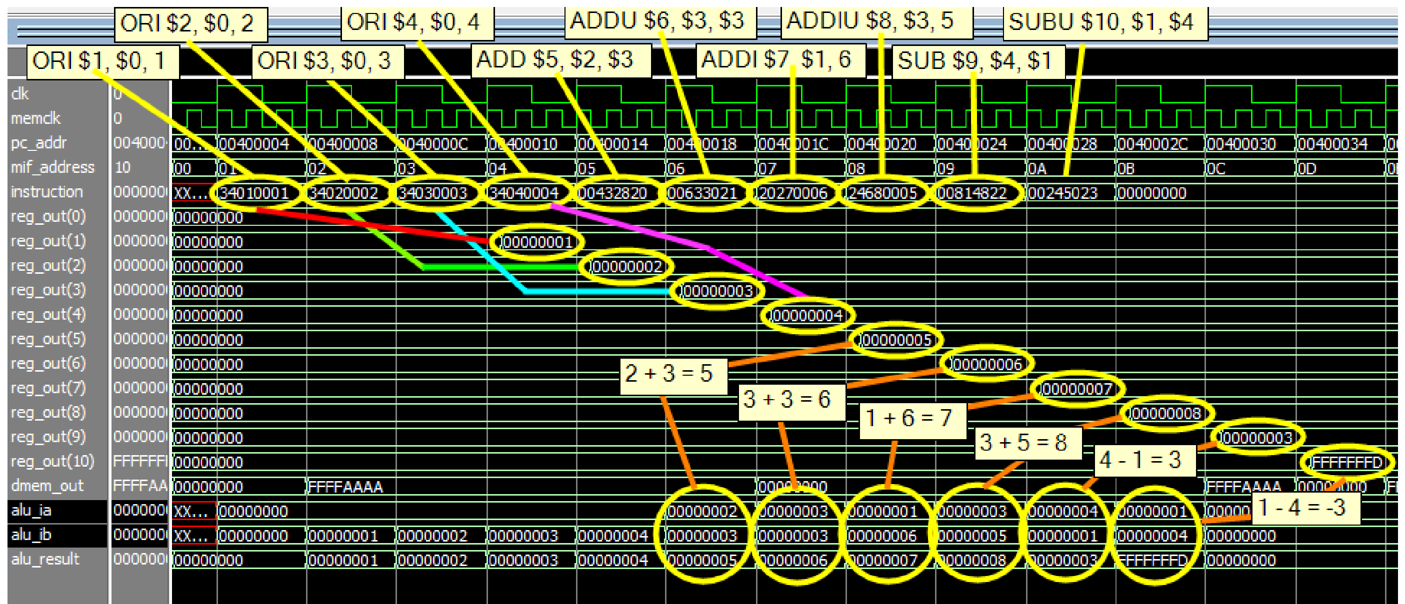
0x1000002C when shifted appropriately yields addr of 0B

Program Loops forever and reg$1 never had FFFF written to it

The next test demonstrates all of the arithmetic operations. It basically loads registers 1-4 with values and uses them throughout the rest of the code to perform operations. It should be noted that functionally there is no difference between ADD and ADDU, or SUB and SUBU. The only difference is that the signed versions can set the overflow flag, while the unsigned versions cannot. We do not deal with the overflow flag in this class however, so these differences are of no consequence to us. The code and waveform are given below:

```
ori     $1, $0, 1
ori     $2, $0, 2
ori     $3, $0, 3
ori     $4, $0, 4
add     $5, $2, $3
addu    $6, $3, $3
addi    $7, $1, 6
addiu   $8, $3, 5
sub     $9, $4, $1
subu    $10, $1, $4
```

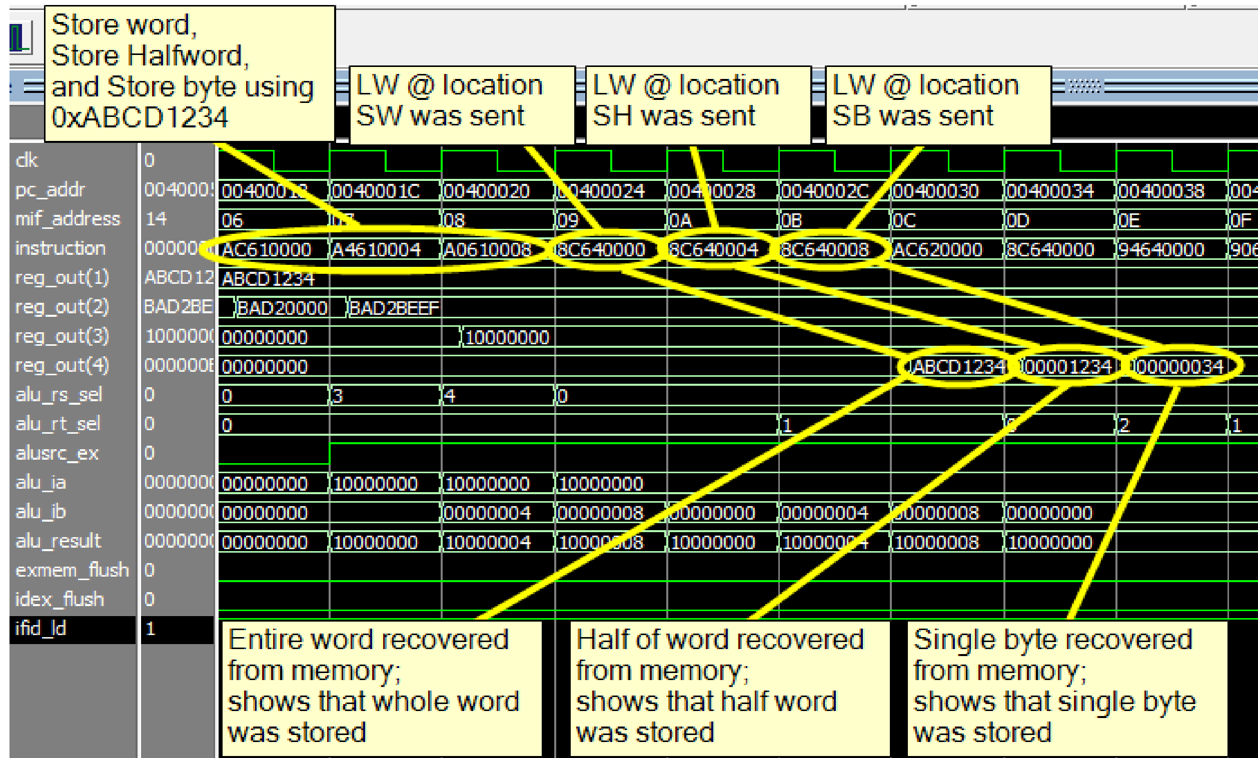
Arithmetic test



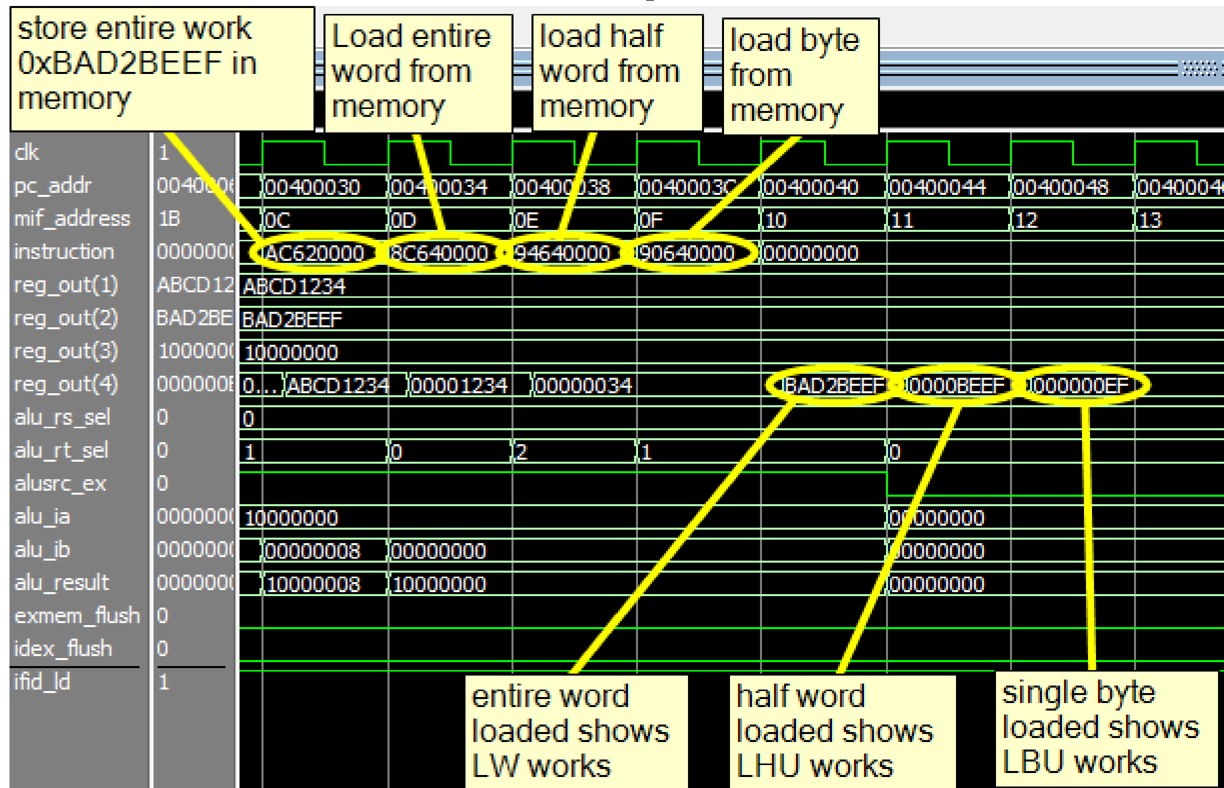
The final test examines the load and store operations. Stores are tested by using store word, store half word, and store byte to store the quantity 0xABCD1234 to data memory. They are then recovered using the load word instruction. Loads are tested by first storing the quantity 0xBAD2BEEF to memory. The value is then recovered using the load word, load half word, and load byte instructions. The code is provided below:

```
lui     $1, $0, 0xABCD
ori     $1, $1, 0x1234
lui     $2, $0, 0xBAD2
ori     $2, $2, 0xBEEF
lui     $3, $0, 0x1000
sw      $1, 0($3)
sh      $1, 4($3)
sb      $1, 8($3)
lw      $4, 0($3)
lw      $4, 4($3)
lw      $4, 8($3)
sw      $2, 0($3)
lw      $4, 0($3)
lhu     $4, 0($3)
lbu     $4, 0($3)
```

The waveform below shows the test for the store operations:



The waveform below shows the test for the load operations:



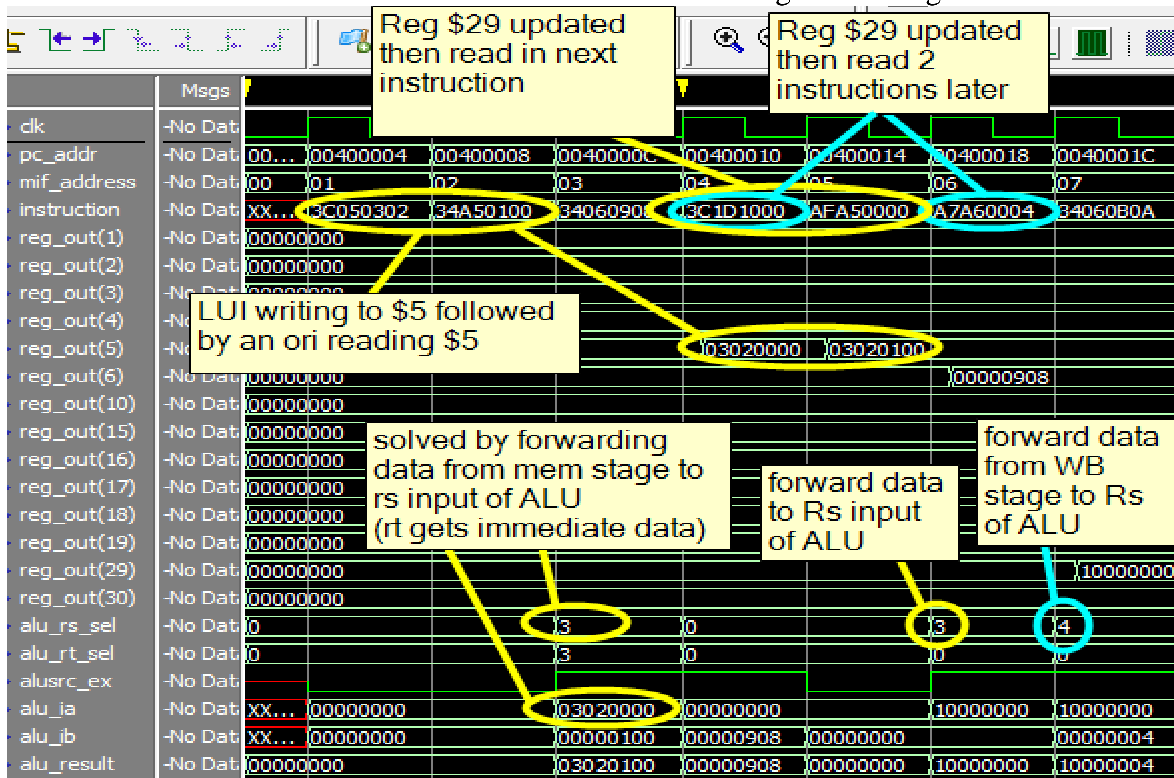
Hazard Detection and Forwarding:

As was discussed previously, without forwarding and stalling the pipelined processor would not function as expected. Therefore it is very important that the processor hold up to rigorous testing of its hazard detection and forwarding capabilities. With this in mind a test program was provided which examines many of the instances in which hazards occur. The decoded assembly is provided below:

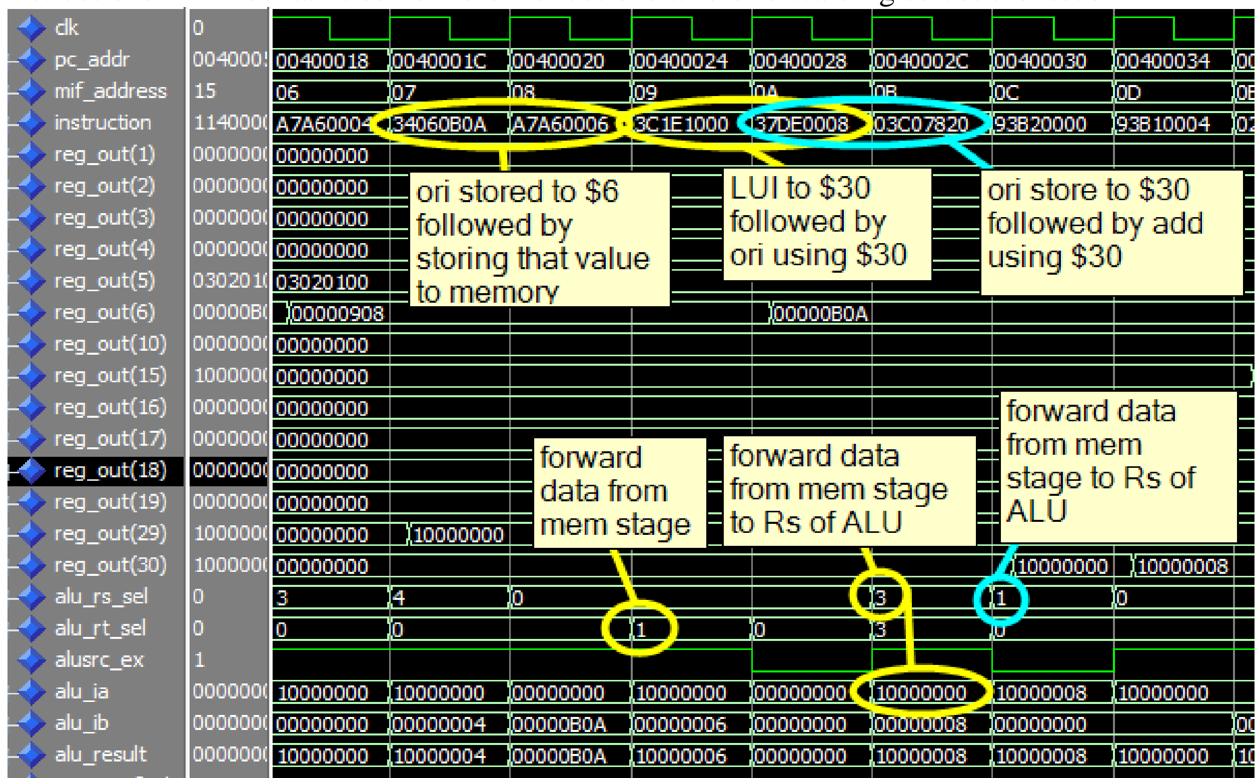
```
1                       lui $5, 0x0302
2                       ori $5, $5, 0x0100
3                       ori $6, $0, 2312
4                       lui $29, 0x1000
5                       sw $5, 0($29)
6                       sh $6, 4($29)
7                       ori $6, $0, 2826
8                       sh $6, 6($29)
9                       lui $30, 4096
10                      ori $30, $30, 8
11                      add $15, $30, $0
12      notmain:        lbu $18, 0($29)
13                      lbu $17, 4($29)
14                      add $16, $17, $18
15                      slti $10, $16, 11
16                      beq $10, $0, case2
17                      add $17, $16, $18
18                      j case3
19      case2:          sub $17, $16, $18
20      case3:          slti $10, $17, 11
21                      beq $10, $0, case4
22                      add $18, $17, $16
23                      j end
24      case4:          sub $18, $16, $17
25      end:            sb $18, 0($fp)
26                      addi $30, $30, 1
27                      addi $29, $29, 1
28                      bne $29, $15, notmain
29                      ori $19, $0, 8
30                      subu $29, $29, $19
31                      lw $1, 0($29)
32                      lw $2, 4($29)
33                      lw $3, 8($29)
34                      lw $4, 12($29)
35      here:           j here
```

The rest of the report will be annotated waveform simulations of this program which point out all of the major data hazards and how they are handled.

Instructions 1 - 6 contains a few hazards that are solved using forwarding.



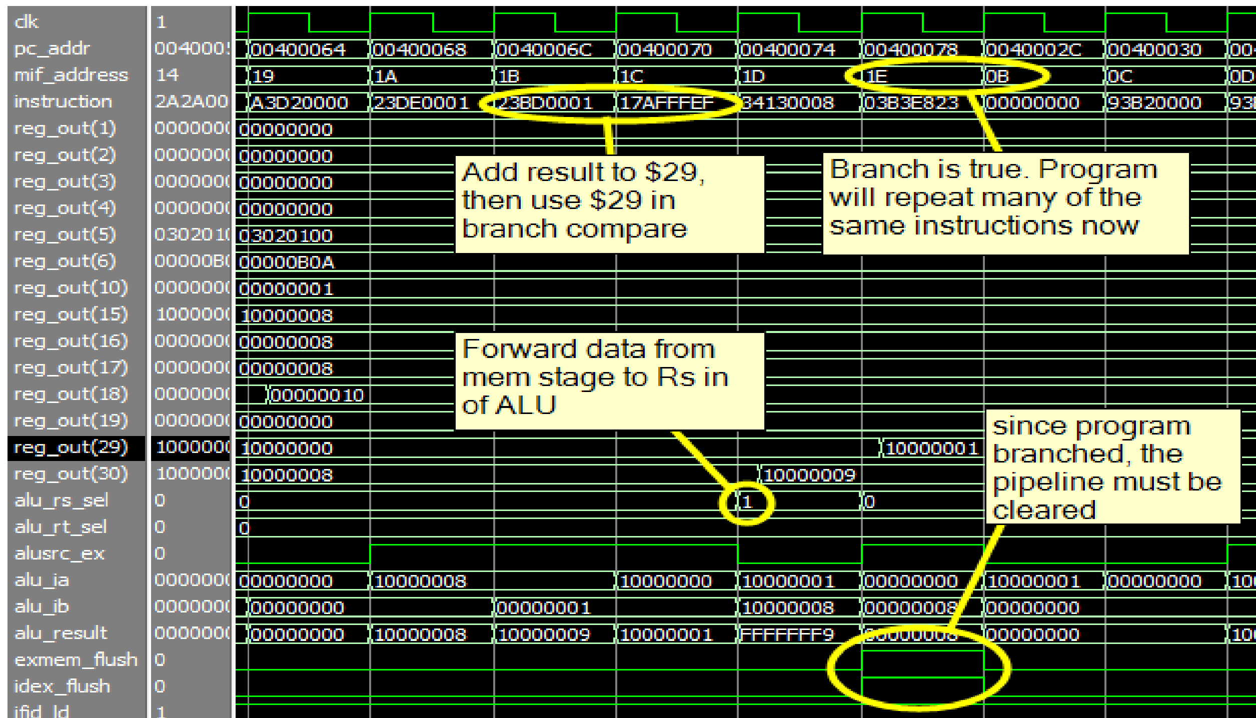
Instructions 7 - 11 contains some more instructions where forwarding solves the issue



Instructions 13 - 16 contains the first hazard that requires a stall to fix. Note that some forwarding is ignored due to the fact that the stall flushes the control lines.

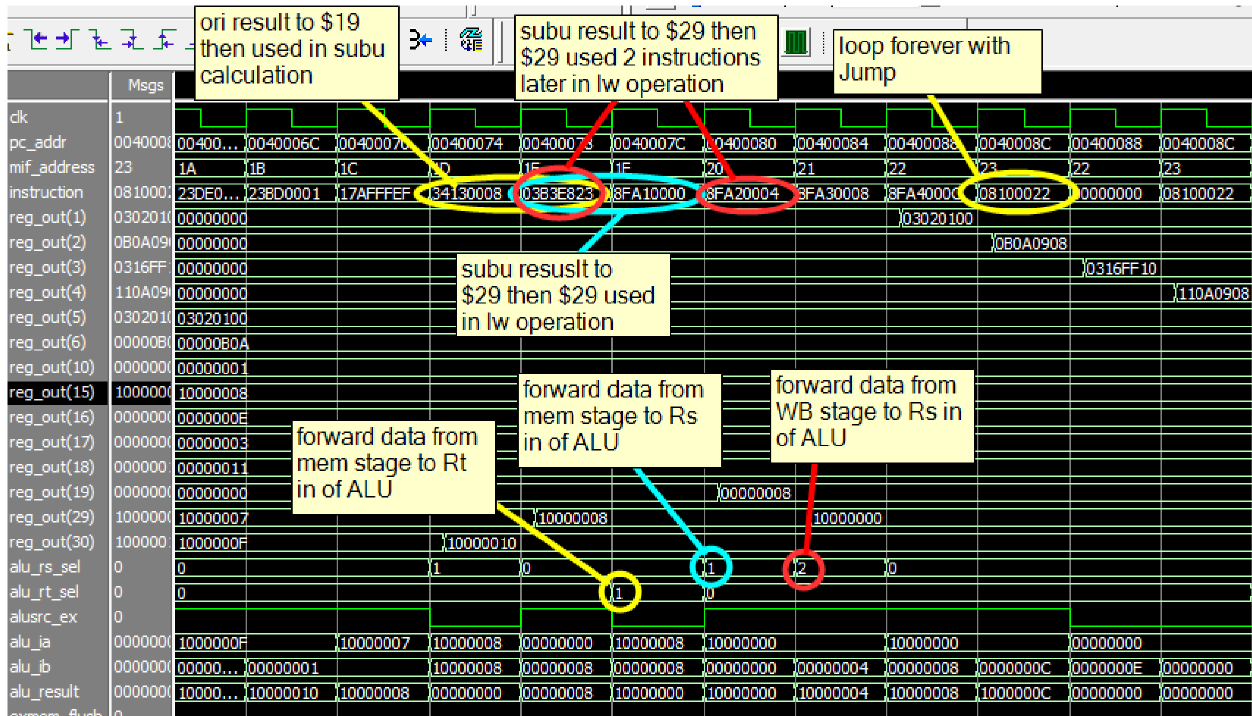
Instruction 20 & 21 contains another hazard solved by forwarding.

Instruction 27 & 28 contains another instance of forwarding, but also shows off a branch for the first time. My design branches in the mem stage which means that some instructions accumulate in the pipeline that should not execute when a branch occurs. To solve this I simply let the instructions proceed normally unless a branch occurs, in which case I flush all control lines for the three instructions that follow the branch. In effect this turns those instructions into nops.

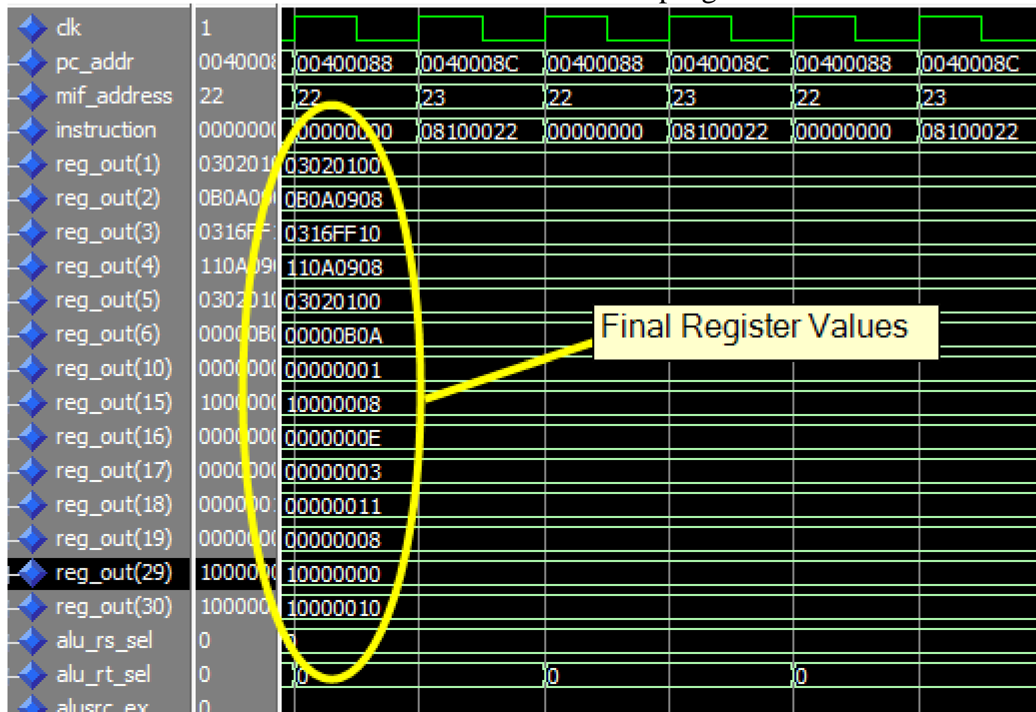


From here the program begins to loop many times wherein it simply repeats section of code that I have already covered. Therefore, I will move on to the end of the program where more hazards await.

Instructions 29 - 35 contain the last hazards within the program. After forwarding takes care of the issues the program proceeds to loop forever.



The waveform below shows the final values of the program.



These final values are in agreement with the values obtained using MIPSim which leads me to conclude that my design passed the hazard stress test.