Lab 4

Introduction

This lab serves to introduce us to pipelined processors and how they work. Further more, we learn how to deal with hazards within assembly programs (rather how to prevent them), as well as how to handle exceptions and interrupts.

Assignment 1

Description: *Place each instruction in approppriate control signal based on functionality. Validate each control signal using binary digits 0 and 1.*

Instruction	RegDst	ALUSrc	Mem2Reg	RegWrite	MemRead	MemWrite	Branch	ALUOp
addi $t0, $t0, 10	0	1	0	1	0	0	0	00
sw $t0, 32($s0)	x	1	x	0	0	1	0	00
bne $t2, $t0, QUIT	x	0	x	0	0	0	1	01
xor $s0, $t1, $t2	1	0	x	1	0	0	0	10
j Print	x	x	x	0	0	0	0	00

Assignment 2

Description: *Each instruction goes through a series of operations prior to execution. If that particular instruction was dependent upon some instruction before it, then a delay was caused. The delay will then be minimized.*

This assignment refers to the following instructions:

```
A: lw $t0,0($t3)
B: add $t1, $t0, $t2
C: sub $t3, $t3, $t1
D: addi $t4, $t4, 4
E: add $t5, $t5, $t4
```

1. Values of the registers after running the instructions hazard-processor-handling.

Register	Value Before	Value After
$t0	2	2
$t1	5	10
$t2	8	8
$t3	2	-8
$t4	4	8
$t5	1	9

1. NOP Instructions necessary.

Instruction/Cycle	1	2	3	4	5	6	7	8	9	10	11	12
lw $t0, 0($t3)	IF	ID	EX	MEM	WB							
add $t1, $t0, $t2		IF	**STALL**	**STALL**	ID	EX	MEM	WB				
sub $t3, $t3, $t1			IF	**STALL**	**STALL**	**STALL**	**STALL**	ID	EX	MEM	WB	
addi $t4, $t4, 4				IF	**STALL**	ID	EX	MEM	WB			
add $t5, $t5, $t4					IF	**STALL**	**STALL**	**STALL**	ID	EX	MEM	WB

1. Instructions can be reordered to reduce cycles. Also, it will reduce the need to depend on registers as well as the need for instruction delay. For

example, instruction D can be moved to the beginning of the code as none of the prior instructions depend on it.

2. Overall, the amount of cycles would be lessened as a result of instruction forwarding. since the next instruction does not have to wait until after the previous instruction is finished with the write-back to get the appropriate value. Instead, it can just access the result from wherever the execution point of the result.

3. For example, if the instruction is add or sub, then the next instruction's ID can start in the same clock cycle as that instruction's EX. If the instruction is load, then the next instruction's ID can start in the same clock cycle as that instruction's MEM because it needs to access the memory for load.

Assignment 3

Description: *To find and minimize hazards using NOP instructions.*

This assigment is based off of the following code:

```
addiu $t0, 3
loop: addiu $t1, 1
sub $t0, $t0, $t1
bne $t0, $zero, loop
```

1. It would take **14 cycles** for the code to run without handling hazards.

2. I was not able to finish this.

3. Program rewritten using NOP instructions:

```
addiu $t0, 3
#NOP Instruction
loop: addiu $t1, 1
#NOP Instruction
```

```
#NOP Instruction
sub $t0, $t0, $t1
#NOP Instruction
#NOP Instruction
bne $t0, $zero, loop
```

4. The use of instruction forwarding would mean that fewer amount of instructions would be needed to get data from the previous steps. More properly phrased, *the number of required cycles would be reduced as a result of this*.

Assignment 4

Description: *The effect of reordering code on program execution.*

1. Please refer to *quad_sol_reordered.asm* for this assignment.

   First block:

```
li        $t0, 2       # Load constant number to integer register
mul        $t4,$t2,$t2  # t4 = t2*t2, where t2 holds b
mul        $t5,$t1,$t3     # t5 = t1*t3, where t1 holds a and t3 holds c
mul        $t5,$t5,4    # Multiply value of s0 with 4, creating 4*a*c
sub        $t6,$t4,$t5  # Calculate D = b^2-4*a*c
tlt        $t6,$0         # If D is less than 0 issue an exception
```

   First block [REORDERED]:

```
#li        $t0, 2       # Load constant number to integer register
mul        $t5,$t1,$t3     # t5 = t1*t3, where t1 holds a and t3 holds c [REORDERED]
mul        $t4,$t2,$t2  # t4 = t2*t2, where t2 holds b
#mul        $t5,$t1,$t3     # t5 = t1*t3, where t1 holds a and t3 holds c
```

```
mul        $t5,$t5,4    # Multiply value of s0 with 4, creating 4*a*c
sub        $t6,$t4,$t5  # Calculate D = b^2-4*a*c
tlt        $t6,$0         # If D is less than 0 issue an exception
```

The reordered first block would take 12 cycles < 14 cycles = original first block.

Second block:

```
neg        $s2,$t2         # Calculate -b and save it to s2
add        $s3,$s2,$s0  # Calculate -b+sqrt(D) and save it to s3
sub        $s4,$s2,$s0  # Calculate -b-sqrt(D) and save it to s4
mul        $s5,$t1,$t0  # Calculate 2*a and save it to s5
div        $s6,$s3,$s5  # Calculate first integer solution
div        $s7,$s4,$s5  # Calculate second integer solution
```

Second block [REORDERED]:

```
neg        $s2,$t2         # Calculate -b and save it to s2
li        $t0, 2       # Load constant number to integer register [REORDERED]
add        $s3,$s2,$s0  # Calculate -b+sqrt(D) and save it to s3
mul        $s5,$t1,$t0  # Calculate 2*a and save it to s5 [REORDERED]
sub        $s4,$s2,$s0  # Calculate -b-sqrt(D) and save it to s4
#mul        $s5,$t1,$t0  # Calculate 2*a and save it to s5
div        $s6,$s3,$s5  # Calculate first integer solution
div        $s7,$s4,$s5  # Calculate second integer solution
```

The reordered second block would take 10 cycles < 11 cycles = original second block.

The conclusion can be made drawn that 3 clock cycles were saved overall, and the reordering of the code did not affect the output at all.

2. If instruction-forwarding was suported, then the program execution would become much faster due to the fact that the amount of stalling in the code would be minimzed which would therefore reduce the amount of clock cycles required. Consider, for example, the first 5 lines of the First Block [REORDERED]. One can observe that both $t4 and $t5 would be ready by the time its execution phase is completed. Hence, the third instruction can use $t5 a lot sooner as opposed to stalling back until the write-back phase. Such is the case for other areas within the code.

Conclusion

This lab taught us about pipelining processors as well as how to "remove" the hazards that exist within them. Also, we learned about exceptions and interrupts. To conclude, as a whole, this lab would be considered a success.