Course Name: Computer Architecture and Assembly Lab

Course Number and Section: 14:332:333:02

Experiment: [Experiment # [6] – CPU Structure, Pipeline Programming and Hazards]

Lab Instructor: Jalal Abdulbaqi

Date Performed: 11/28/2018

Date Submitted: 12/10/2018

Submitted by: [Jahidul Islam - 171001155]

Course Name: Computer Architecture and Assembly Lab

Course Number and Section: 14:332:333:02

! Important: Please include this page in your report if the submission is a paper submission. For electronic submission (email or Sakai) please omit this page.

----------For Lab Instructor Use ONLY----------

GRADE: _____

COMMENTS:

Electrical and Computer Engineering Department
School of Engineering
Rutgers University, Piscataway, NJ 08854

ECE Lab Report Structure

1. Purpose / Introduction / Overview – describe the problem and provide background information
2. Approach / Method – the approach took, how problems were solved
3. Results – present your data and analysis, experimental results, etc.
4. Conclusion / Summary – what was done and how it was done

Assignment 1

addi t0, t0, 10

RegDest	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
0	1	0	1	0	0	0	00

lb t1, 32(t0)

RegDest	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
0	1	0	1	1	0	0	00

ori t2, t0, 4

RegDest	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
0	0	0	1	0	0	0	01

bne t2, t0, exit

RegDest	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
0	0	1	0	0	0	1	01

Assignment 2

1)

CC	1	2	3	4	5	6	7	8	9	10	11	12	13			
lw	IF	ID	EX	MEM	WB											
add		IF	ID	EX	MEM	WB										
sub			IF	ID	EX	MEM	WB									
addi				IF	ID	EX	MEM	WB								
sub					IF	ID	EX	MEM	WB							

2)
Original:
t0 = 2
t1=5
t2=8
t3=2
t4=4
t5=1

Code:
lw t0, 0(t3)
add t0, t0, t1
sub t1, t3, t1
addi t4, t3, 4
sub t5, t5, t4

After Venus:
t5 = -5 or 0xfffffffb
t4 = 6 or 0x00000006
t3 = 2 or 0x00000002
t2 = 8 or 0x00000008

t1 = -3 or 0xfffffffd
t0 = 2192769043 or 0x82b30013

3)

```
lw t0, 0(t3)
                #There is a hazard going from lw to add since add depends on t0 which is loaded
                by lw. Stalls or forwarding must be implemented. This is Data Hazard or Pipeline
                Data Hazard
add t0, t0, t1

sub t1, t3, t1

addi t4, t3, 4
                #Hazard going from addi to sub since sub depends on the t4 found by addi. This
                is a Data Hazard or Pipeline Data Hazard.
sub t5, t5, t4
```

4)

```
lw t0, 0(t3)
stall
stall
add t0, t0, t1
sub t1, t3, t1
addi t4, t3, 4
stall
sub t5, t5, t4
```

CC	1	2	3	4	5	6	7	8	9	10	11	12	13			
lw	IF	ID	EX	MEM	WB											
add		IF	stall	stall	ID	EX	MEM	WB								
sub					IF	ID	EX	MEM	WB							
addi						IF	ID	EX	MEM	WB						
sub							IF	Stall	ID	EX	MEM	WB				

5)
Original:
(a)lw t0, 0(t3)
(b)add t0, t0, t1
(c)sub t1, t3, t1
(d)addi t4, t3, 4
(e)sub t5, t5, t4

With instruction reordering, instructions that don't require the previous instruction can be process in between two that are dependent to act as a kind of a stall between the two dependant instructions.

So we can have:
With Re-ordering:
(a)lw t0, 0(t3)
(d)addi t4, t3, 4
(c)sub t1, t3, t1
(e)sub t5, t5, t4
(b)add t0, t0, t1

This requires no stalls.

6)
Forwarding can also help the code as it would require only one stall between lw and add if forwarding was used. Also the one stall between addi and the final sub would not be need anymore meaning two clock cycles in total would be saved with forwarding alone. Now if we also consider reordering ot be an option we can write the code again with no stalls. The difference would be much less re-ordering would need to be done to eliminate stalls if forwarding was an option that could be utilized.

Assignment 3
Consider the following code:

```
        addi t1, t0, 3
Loop:   addi t2, t1, 1
        sub t0, t0, t1
        bne t0, zero, loop
```

1) Without Hazard Handling;

CC	1	2	3	4	5	6	7	8

addi	IF	ID	EX	MEM	WB			
addi		IF	ID	EX	MEM	WB		
sub			IF	ID	EX	MEM	WB	
bne				IF	ED	EX	MEM	WB

Without Hazard Handling Hardware we can see it would take 8 clock cycles to execute.

2)

CC	1	2	3	4	5	6	7	8
addi	IF	ID	EX	MEM	WB			
addi		IF	ID	EX	MEM	WB		
sub			IF	ID	EX	MEM	WB	
bne				IF	ED	EX	MEM	WB

Hazards:

- Between the first two addi's there is a data hazard because the second addi depends on the t1 resulting from the first addi
- The sub also depends on t1 from the first addi and since it has not yet been written back that also has a data hazard
- t0 is dependant on the sub before bne can use it meaning there is a data hazard between bne and sub.
- Since it is a loop the t0 from the bne checks to make sure t0 is not 0 and then proceeds to loop or the next instruction which can cause a control hazard. So we will use two afters before proceeding back to beginning of loop for assurance.

3)

Loop:

```
addi t1, t0, 3
stall
addi t2, t1, 1
sub t0, t0, t1
stall
bne t0, zero, loop
<after 1>
<after 2>
```

CC	1	2	3	4	5	6	7	8		
addi	IF	ID	EX	MEM	WB					
addi		IF	stall	ID	EX	MEM	WB			
sub				IF	ID	EX	MEM	WB		
bne					IF	stall	ID	EX	MEM	WB
After 1						IF	ID	bubble	bubble	bubble
After 2							IF	bubble	bubble	bubble
addi								IF	ID	EX

The afters are used to separate loops and allow bne to make decision.

4) With Forwarding:

RED -> RED

BLUE -> BLUE

CC	1	2	3	4	5	6	7	8
addi	IF	ID	EX	MEM	WB			
addi		IF	ID	EX	MEM	WB		
sub			IF	ID	EX	MEM	WB	
bne				IF	ED	EX	MEM	WB

With forwarding no stalls would need to be used in this code.

Assignment 4

1)

This will display the changes made and a separate text file is included as well.

THIS RED SECTION WE WILL CALL SECTION 1

```
#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
addi a3,x0,4           #added
mul t5,t5,a3           # Multiply value of s0 with 4, creating 4*a*c
sub t6,t4,t5           # Calculate D = b^2-4*a*c
```

```
# calculating the integer square root by the equation x*x = D
li a0, 1               # Square Root Partial Result, sqrt(D).
mv s1,t6               # Move value in register t6 to register s1 for safety purposes.
```

THIS BLUE SECTION WE WILL CALL SECTION 2

```
sqrtloop:              # calculating the integer square root of D

        mul s2,s0,s0
        bge s2, s1, endsqrt
        addi s0,x0, 1
```

THIS GREEN SECTION WE WILL CALL SECTION 3

```
endsqrt:
        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
        li t0, 2  # Load constant number to integer register
        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
```

```
SECTION 1 RE-ORDERED
#li t0, 2
addi a3,x0,4          #re-ordered addi to above because it does not depend on anything but
                      having it below results in stalls because mul t5,t5,a3 depends on the addi
                      so having it here means no stall needed.[1 CC saved]

mul t5,t1,t3          #switched mul t4,t2,t2 with mul t5,t1,t3 to account for mul t5,t5,a3
                      requiring the t5 from mul t5,t1,t3.[1 CC saved]

mul t4,t2,t2

mul t5,t5,a3

sub t6,t4,t5          #Although this sub needs a stall as it depends on t5 from the previous we
                      do not have another instruction to put in between so a stall will be kept

        From section 1 we saved two clock cycles total thus far.

SECTION 2 RE-ORDERED
sqrtloop:             # calculating the integer square root of D

        mul s2,s0,s0

        addi s0,x0, 1     #We switched the addi and bge because bge requires s2 from mul
                          and thus needed a stall so having addi which is not dependant on
                          anything else in between saves us from having to use a stall.
                          [1 CC Saved]

        bge s2, s1, endsqrt

        From section 2 we saved one clock cycle.

SECTION 3 RE-ORDERED
endsqrt:
        neg s2,t2

        li t0, 2          # We moved li up here to act as the stall between the neg and add since
```

add depends on the s2 from the neg. Also having li up here means that the t0 that mul depends on for it does not require mul to have stalls before it since it is done at an earlier clock cycle. [3 CC Saved]

```
add s3,s2,s0

sub s4,s2,s0

mul s5,t1,t0    # Between this mul and the next div we need to stall. We could put the sub in between them to lower the clock cycle but that would mean adding a stall to mul because after li two clock cycle stalls are needed before the t0 can be used properly. The current position of the sub acts in place of the second stall that is needed so moving it would not save any clock cycles.

div s6,s3,s5

div s7,s4,s5
```

From section 3 we saved three clock cycles.

Total Clock Cycles Saved:
Section 1 + Section 2 + Section 3 = 2 + 1 + 3 = **6 Clock Cycles Saved**

2)
If we used forwarding, clock cycles could be reduced further as stalls would not be as prevalent since between add and sub for example we would not need a stall. Which means if we look at a particular section of code:

```
mul s5,t1,t0    # Between this mul and the next div we need to stall. We could put the sub in between them to lower the clock cycle but that would mean adding a stall to mul because after li two clock cycle stalls are needed before the t0 can be used properly. The current position of the sub acts in place of the second stall that is needed so moving it would not save any clock cycles.

div s6,s3,s5
```

Here the stall that is needed between the mul and div would no longer be needed if forwarding is used saving a clock cycle already. Also with forwarding, much less re-ordering would need to be done to save the same clock cycles.