ECE 441
Advanced Digital Design and FPGAs

Lab 4
Daniel Sciortino
Kevin Johnson

April 4th, 2016

Honor Pledge: I pledge to support the honor system of Old Dominion University. I will refrain from any form of academic dishonesty or deception, such as cheating or plagiarism. I am aware that as a member of the academic community, it is my responsibility to turn in all suspected violators of the honor system. I will report to Honor Council hearings if I am summoned.

Introduction

Design Work

When multiplication and division instructions were added to the MIPS model, the execution stage behaves different than other R-type instructions. Since the ALU returns a 32bit result, a special 64bit register was added to the ALU. The special register can only change when the Fcode corresponds to either multiplication or division. This register is read by the ALU during the execution stage when the Fcode corresponds to MFHI or MFLO which causes the appropriate 32bits appears on the ALU result. A slight modification was also made to the next stage. The ALU result was not written to a register if the Fcode was related to multiplication or division. The instructions used to perform the weighted average were designed for a set of 5 numbers.

First, the addresses were calculated and stored in $t0 through $t9. Second, the data was loaded from the RAM. Third, $t0 was calculated through iterative multiplication followed by addition. The formula for the numerator is shown below.

$$\$t0 = \sum_{i=0}^{4} A[256 + i] * B[512 + i]$$

The denominator was calculated by accumulating all of the numbers corresponding to the weights in the register $t1. The formula for denominator is shown below.

$$\$t1 = \sum_{i=0}^{4} B[512 + i]$$

The division of $t0 and $t1 produced the final result of the weighted average. The result is then stored in memory location 0xFF. The final calculations are shown in the formulas below.

$$\$t0 = \$t0/\$t1$$

RAM[255] = $t0

The abstract mathematical formulas discussed above were broken down into assembly which describes the operations in a similar form. Once the addresses are calculated, all of the data is loaded from memory into the registers. First, multiplication of the weights is calculated. Second, the newly calculated numbers are summed together through successive addition. Third, the weighted values are summed together. Fourth, the two summations are divided by each other and the result is transferred to memory.

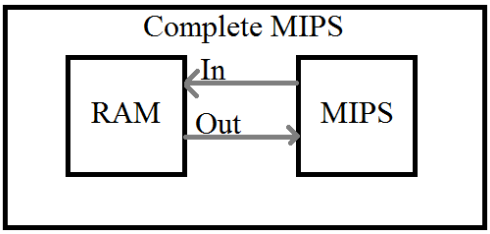
```
addi $t0, $t0, 256
addi $t1, $t1, 257
addi $t2, $t2, 258
addi $t3, $t3, 259
addi $t4, $t4, 260
addi $t5, $t5, 512
addi $t6, $t6, 513
addi $t7, $t7, 514
addi $t8, $t8, 515
addi $t9, $t9, 516
  lw $t0, 0($t0)
  lw $t1, 0($t1)
  lw $t2, 0($t2)
  lw $t3, 0($t3)
  lw $t4, 0($t4)
  lw $t5, 0($t5)
  lw $t6, 0($t6)
  lw $t7, 0($t7)
  lw $t8, 0($t8)
  lw $t9, 0($t9)
 mult $t0, $t5
   mflo $t0
 mult $t1, $t6
   mflo $t1
 mult $t2, $t7
   mflo $t2
 mult $t3, $t8
   mflo $t3
 mult $t4, $t9
   mflo $t4
 add $t0, $t0, $t1
 add $t0, $t0, $t2
 add $t0, $t0, $t3
 add $t0, $t0, $t4
 add $t5, $t5, $t6
 add $t5, $t5, $t7
 add $t5, $t5, $t8
 add $t5, $t5, $t9
   div $t0, $t5
   mflo $t0
 andi $t9, $t9, 0
addi $t9, $t9, 255
  sw $t0, 0($t9)
```

The Assembly Instructions

Operation	Dec Input 1	Dec Input 2	Dec Output	Hex Output
mul	2	17	34	22
mul	3	19	57	39
mul	5	23	115	73
mul	7	29	203	CB
mul	11	31	341	155
add	34	57	91	5B
add	91	115	206	CE
add	206	203	409	199
add	409	341	750	2EE
add	17	19	36	24
add	36	23	59	3B
add	59	29	88	58
add	88	31	119	77
div	750	119	6	6
andi	31	0	0	0
addi	0	255	255	FF
sw	6			

Expected output values after loading the input values

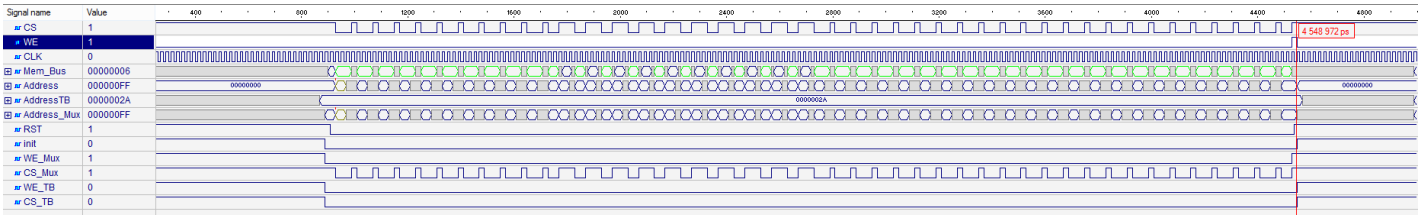
In part 2, the memory bus was removed and replaced with an input and output. Each of these signals has the same size as the original memory bus. The test bench was removed from accessing the memory because it would require additional control signals and produce a greater propagation delay from the memory to the CPU. Therefore, the memory must be preloaded with instructions. The PLL was programed to have a division of 50 which produced a clock of 1Mhz. The PLL is a component of the complete MIPS and its 1Mhz clock drives both the memory and the MIPS. The VHDL model has the new IO signals named relative to the memory.



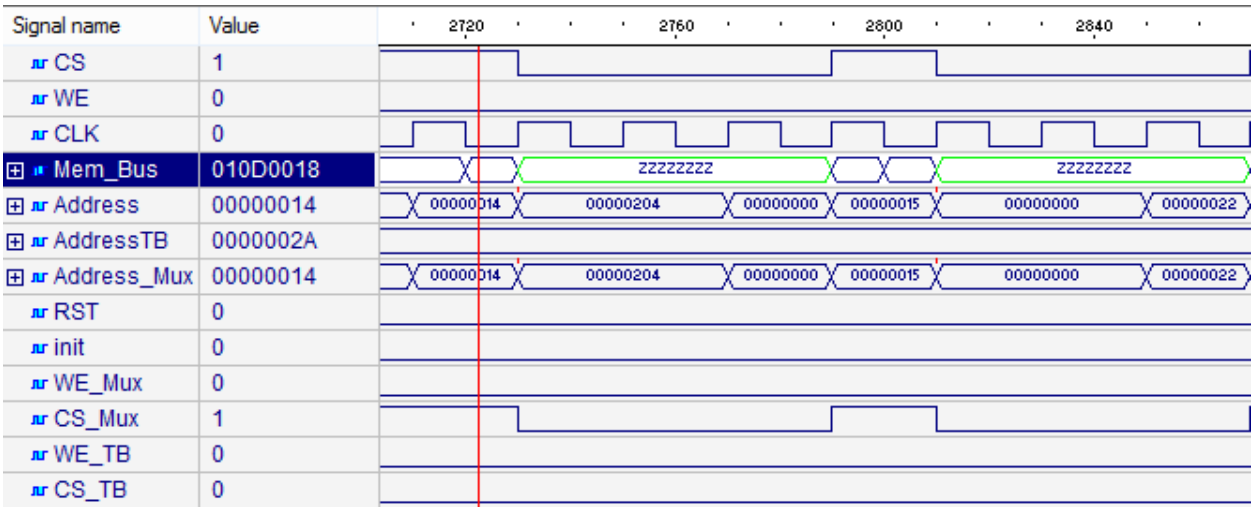
Simulation Results

Part1

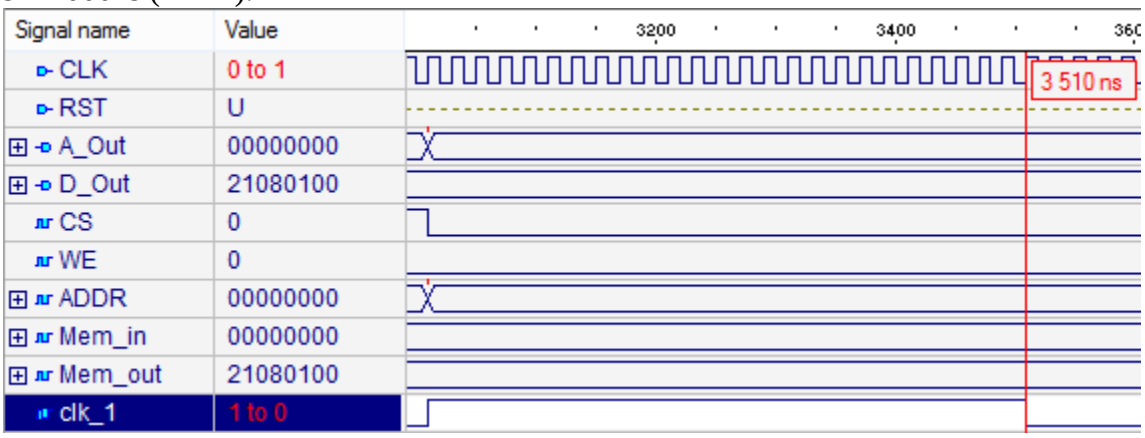
Here is the overview of the entire simulation from start to finish. Write enable (WE) goes high upon executing the last instruction which is store word (sw). The final result is shown on mem_bus and is 0x00000006. This is the correct value.



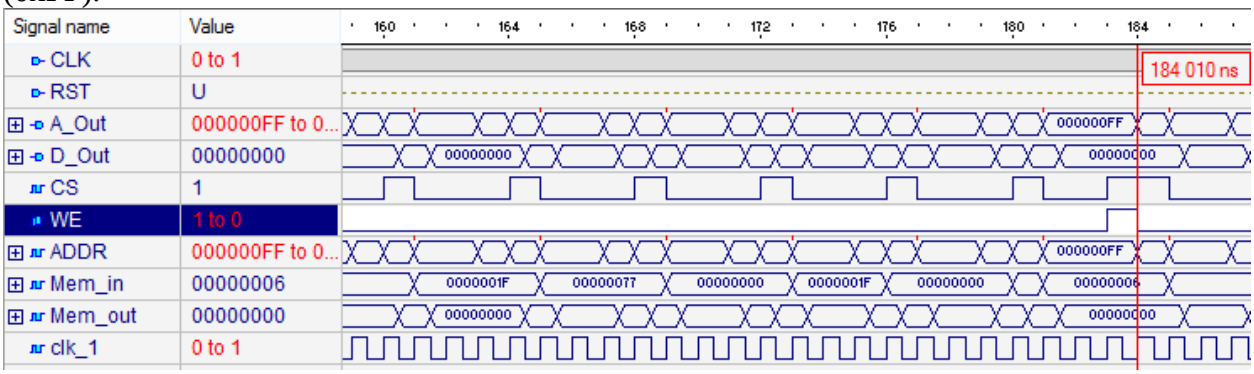
The 20th instruction (0x14) is multiplication of $t0, $t5 = 2*17 = 34 = 0x22. Since multiplication and division do not effect ordinary registers, the result will not be displayed on the mem_bus until mflo is executed. Once the 21st instruction (0x15) is executed, the last stage will write the lower half of the multiplication to mem_bus. This can be seen operating correctly below.



In part 2, A 1 Mhz clock was generated from a 50 Mhz clock. This new slower clock is called clk_1. The clock in the figure below rises at 3010ns and falls and 3510ns which is difference of 500ns. Since this is half of the cycle, we can calculate the total period of the new clock to be 2x500ns = 1000ns (1Mhz).



In the new model, the memory bus has been replaced with Mem_in and Mem_out. These signals are relative to the memory component. To verify that the new model is working correctly, the only value written to memory should be 0x00000006. The figure below shows the first time WE goes high and has the number 6 on the mem_in signal. The memory address is also correct (0xFF).



DE2 Board Setup and Observations

In part 2, the place and route summary shows a 3% usage on the logic elements and a 2% usage on the multipliers. I believe the total number of multipliers is 8 because a single 32bit multiplier requires 4 9bit multipliers. Since there are two instructions containing a multiply function (multu, mult) it consumes 8. This could be optimized to only consume 4 9 bit multipliers. The total number of pins is 20 because I mapped the lower part of the memory output to the 18 LEDs. The other 2 inputs are the Clock and KEY[0]. The button is mapped to reset.

Fitter Summary	
Fitter Status	Successful - Sun Mar 27 18:25:56 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	Lab4Part2
Top-level Entity Name	Complete_MIPS
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,388 / 114,480 (3 %)
Total combinational functions	3,370 / 114,480 (3 %)
Dedicated logic registers	238 / 114,480 (< 1 %)
Total registers	238
Total pins	20 / 529 (4 %)
Total virtual pins	0
Total memory bits	34,816 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	8 / 532 (2 %)
Total PLLs	1 / 4 (25 %)

Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	45.67 MHz	45.67 MHz	PLL1|altpll_component|auto_generated|pll1|clk[0]	

Part 2

Part 3

For the third part of this lab we were to verify the timing constraints with TimeQuest Timing Analyzer, then verify the design with SignalTap post-synthesis. One issue that we had was Quartus was interpreting an OpSave as a latch and thus seeing it as a clock when it should only seeing the phase lock loop clock. To get around this issue we made the multiplication and division operations asynchronous.

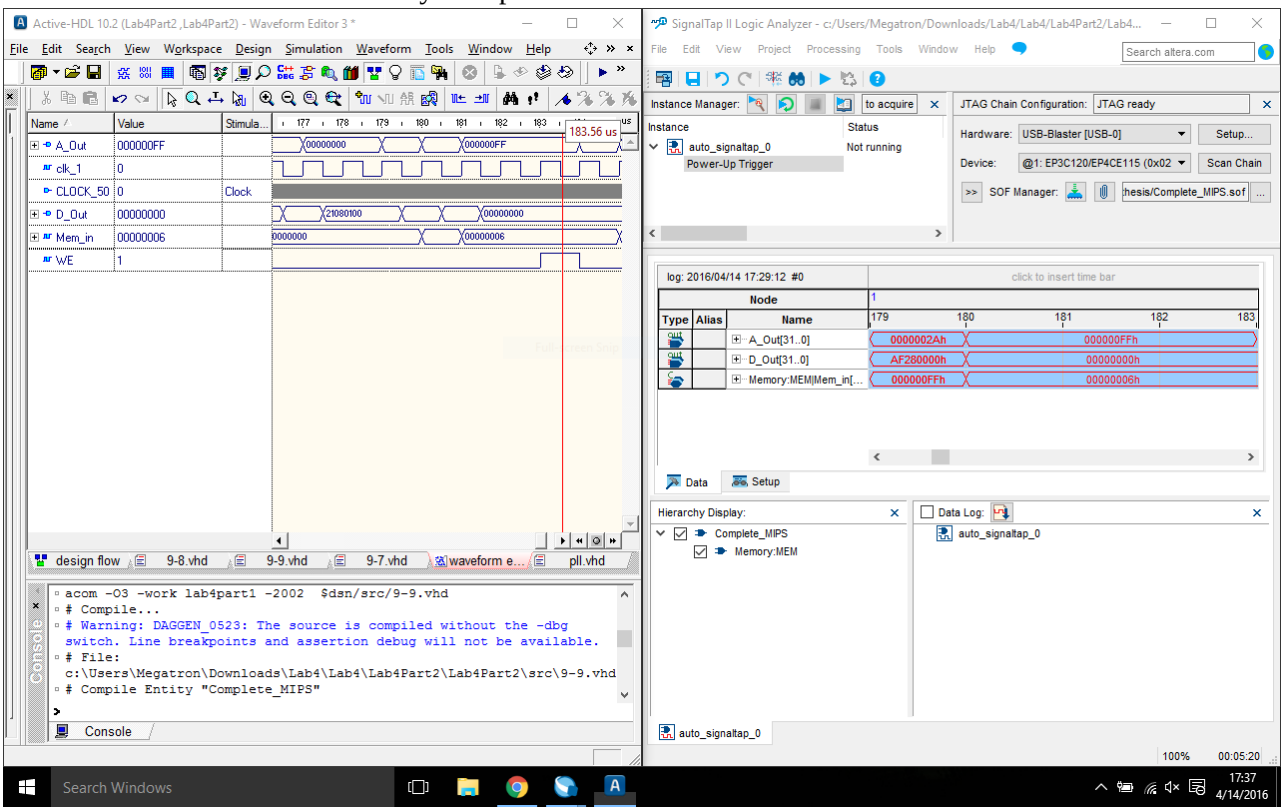
After verifying the operation was correct in Aldec we found that TimeQuest did not show the illusive OpSave.add clock. The figure below shows the a snippet from TimeQuest with the OpSave.add clock.

Slow 1200mV 85C Model Setup Summary

	Clock	Slack	End Point TNS
1	MIPS:CPU|OpSave.add	-121.992	-5817.669
2	PLL1|altpll_component|auto_generated|pll1|clk[0]	-7.617	-220.047

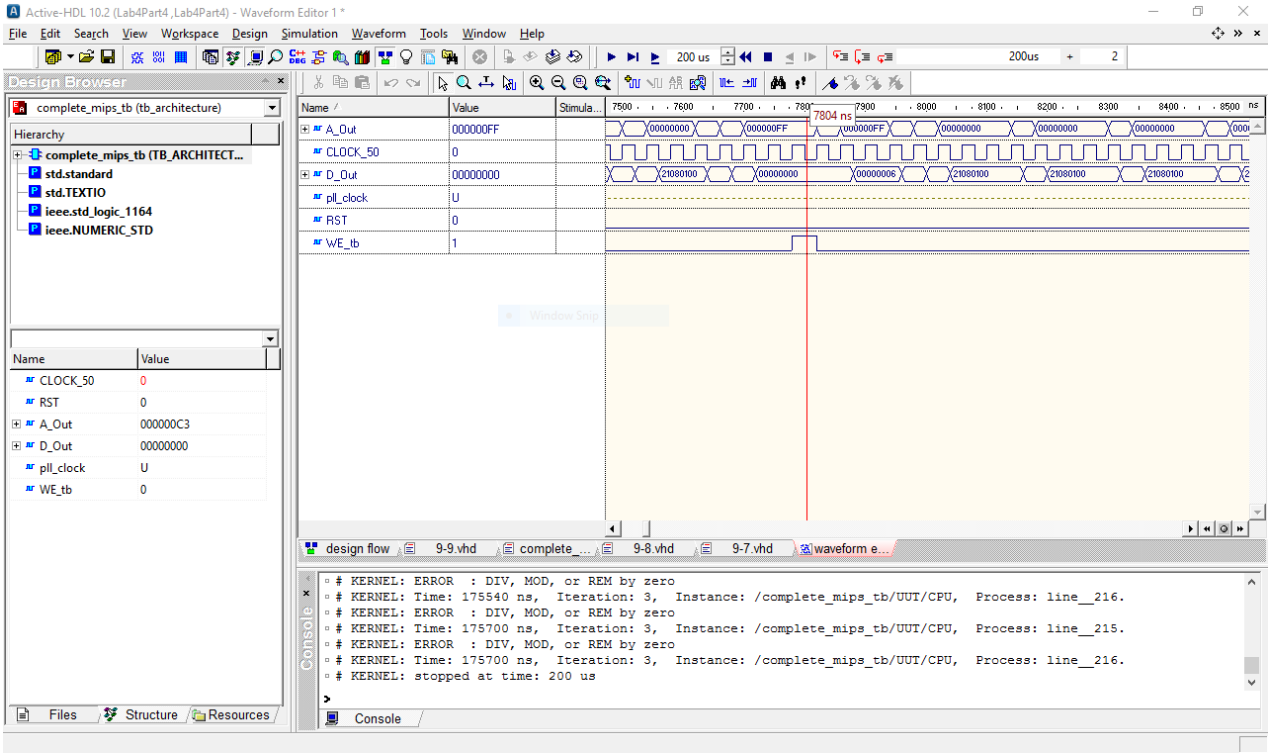
After setting multicycle paths through the div and mult operations we were able get the slack to a more manageable number.

With SignalTap after synthesis it took some time to look for the WE signal to go high for the value to go to the memory. This gave us an issue because of what Quartus can synthesize out, in this case the WE signal was being synthesize out. After comparing the Aldec simulation and the SignalTap collected data we were able to verify the operation of the MIPS CPU.



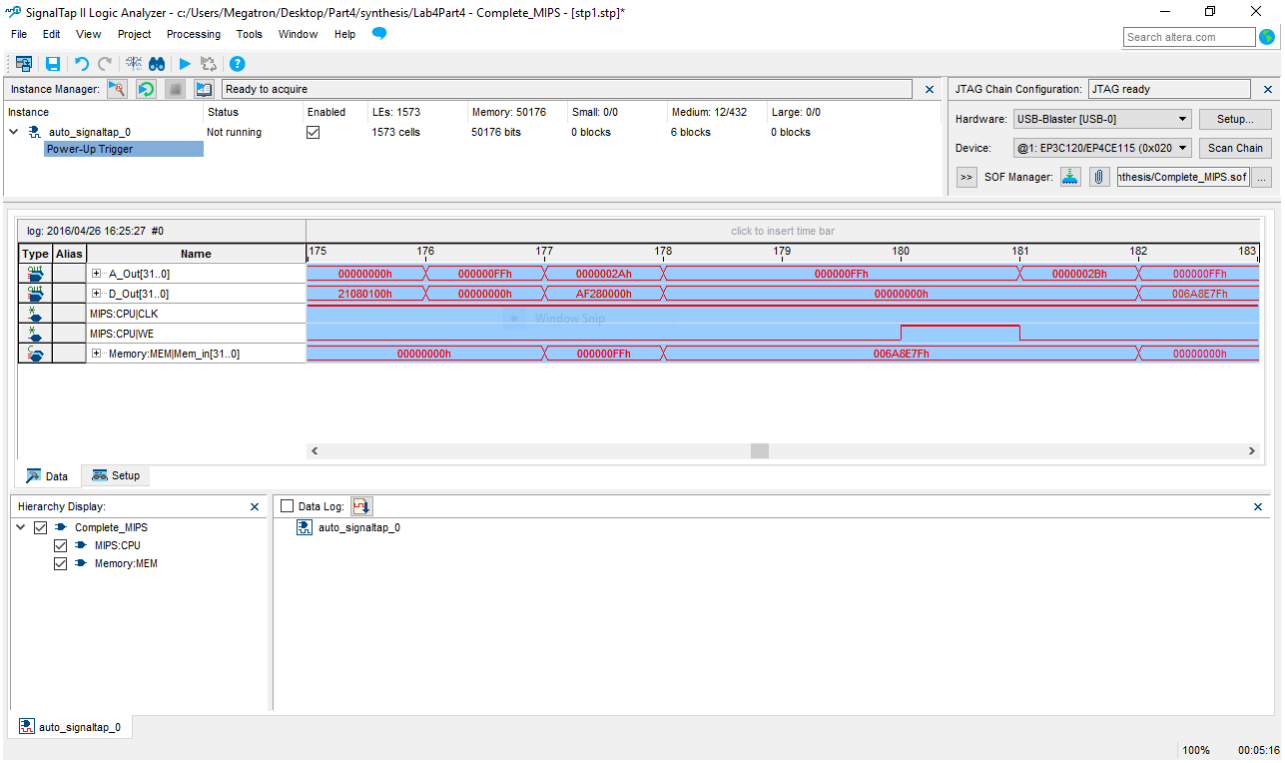
Part 4

In the final part of this assignment we were to up the operating frequency from 1Mhz to 50Mhz. Initially we had a phase lock loop defined that would use the 50Mhz and bring it down to 1Mhz. For the design of this part we deleted the phase lock loop and used the 50 Mhz clock to see how the processor would operate, and if any wait statements would be needed to give the multiplication and division time to operate. We were able to verify that our design for the multiplication and division immediately.



After verifying the design in Aldec we took it to Quartus to verify the slack and verify the operation post synthesis. Like for Part 3 of the MIPS processor it took a little but after adjusting the multicycle paths through the div and mult operations we were able to get the slack closer to 0.

With in SignalTap we were able to verify the operation was correct, unlike Part 3 the WE signal was not getting sythesized out. We were able to locate that signal when the correct value was getting written mem_in.



Analysis

Ultimately we were able to get the MIPS processor to work, it took us a weekend to get the processor running and us to verify the correct output. Trying to fixed the Opsave.add false clock that was being generated took longer, but by making and mult and div operations asynchronous we were able to resolve that issue. The utilization on the FPGA was 4% when verifying the functionality with SignalTap, and 3% with out. This is due to SignalTap needing to be compiled into the already existing design.

Summary and Conclusion

For this lab we were to implement a 32-bit MIPS processor, that was not pipelined on to the DE2-115 FPGA. From this template in the textbook we were to add mult, multu, div, divu, mflo, mfhi. Where the mflo and mfhi we storing the 64-bit value of the mult, and div operations. Initially getting the multiplication and the division functioning was pretty painless, however trying to track down a new clock signal that was getting created. Once the stray clock was managed getting the design to work with a 1Mhz and 50 Mhz clock was straight forward. When it came time for checking the timing requirements in TimeQuest but adding a couple multicycle paths for the div and mult operations lowered the required slack for the design to work. Verification in SignalTap was easily verified with the correct values for the memory location and the correct mathematical value going into a_out and d_out respectivly.