Quinn Roemer

Engineering – 303

Lab 15

5/15/2017

Introduction/Description

In this lab, I was supposed to implement a more complicated assembly language program for our single cycle computer. This new assembly language program performed the same task as the one I created in lab 14 but performs it more efficiently. Also, in this lab, I was supposed to attempt to make a factorial program. This program would take a number and output its factorial answer.

Design

Part 0 – Multiplier with Comparison

The first circuit that I created was a hard wired assembly language program that we could plug into our single cycle computer from lab 14. This program performs the same task as the program we created in lab 14. However, it performs this task much more efficiently. For example, since our single cycle computer lacks a multiplier we must use software to multiply numbers. This is done by adding a number to zero a certain number of times. For instance, the calculation 5*4 equals 20. This calculation could also be done by adding 5 to zero 4 times. To make this circuit more efficient it essentially switches the values inside of its registers until it has the smaller number as its counter variable.

Here is the Verilog code for the circuit.

```
52  always@(Address)
53  begin
54      case(Address)
55      /* Multiplies two given inputs A * B = P
56
57      A goes in R0
58      B goes in R1
59      P goes in R2
60      Jump address goes in R3
61
62      For efficiency, first compare A and B, swap smaller number into A if required
63      R2 is used for the comparison
64      R3 is used as a temporary location for the swap
65
66      bitshifting and bitwise OR is used to load a 4 bit jump address to R3.
67      */
68
69      //Setup
70         0: IR <= {LOAD, R0, NULL, NULL};     //Load A to R0
71         1: IR <= {LOAD, R1, NULL, NULL};     //Load B to R1
72         2: IR <= {SUB, R2, R0, R1};       //Subtract A - B, put in R2
73         //If A < B then Branch to MultInit (binary 7)
74         3: IR <= {BRN, 3'b000, R2, 3'b111};
75      //Swap
76         4: IR <= {MOVA, R3, R0, NULL};       //Move R0 to R3
77         5: IR <= {MOVA, R0, R1, NULL};       //Move R1 to RO
78         6: IR <= {MOVA, R1, R3, NULL};       //Move R3 to R1
79      //MultInit
80         //Load 3 bits of jump address (14) to R3
81         7: IR <= {LDI, R3, NULL, 3'b001};
82         8: IR <= {SFTL, R3, NULL, R3};       //Shift left
83         9: IR <= {SFTL, R3, NULL, R3};       //Shift left
84         10: IR <= {SFTL, R3, NULL, R3};      //Shift left
85         //Load 3 more bits of jump address to R2
86         11: IR <= {LDI, R2, NULL, 3'b110};
87         //Combine R2 and R3 bits into R3
88         12: IR <= {OR, R3, R3, R2};
89         13: IR <= {LDI, R2, NULL, ZERO};     //Set P to 0
90      //Loop
91         //If A = 0 then brance to Done (18)
92         14: IR <= {BRZ, 3'b010, R0, 3'b010};
93         15: IR <= {ADD, R2, R2, R1};         //Add B to P
94         16: IR <= {DEC, R0, R0, NULL};       //Decrement A
95         17: IR <= {JMP, NULL, R3, NULL};     //Jump to loop
96      //Done
97         18: IR <= {ST, NULL, NULL, R2};      //Output answer
98         default
99            IR <= 255;
100     endcase
101  end
102  endmodule
```

Part 1 – Factorial Program

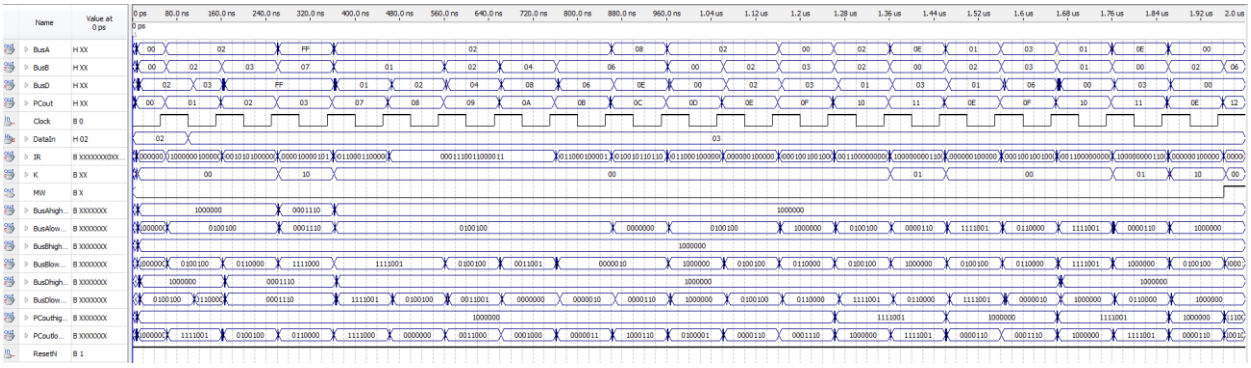
In the last section of the lab, we were told to attempt to create a factorial program for this computer. The fact that this computer only has 4 available registers severely complicates this task. After spending several hours brainstorming I came up with some code that should work. This code is capable of performing a factorial calculation. Note, while testing this code it only worked for the numbers 1, 2, and 3. However, I believe that this code is capable of performing factorial calculation up to 5. This code is extremely inefficient in its design. For example, this code uses the answer of the previous calculation as a counter for the next calculation. If I were to enter 5 as the number to do the calculation on it would take a little over 800 clock cycles to output the answer. Because of this, I found this code to be difficult to test. If I were to use only branch statements in the code and branch on negative I could probably make the code hugely more efficient. As I would have one more register to play around with.

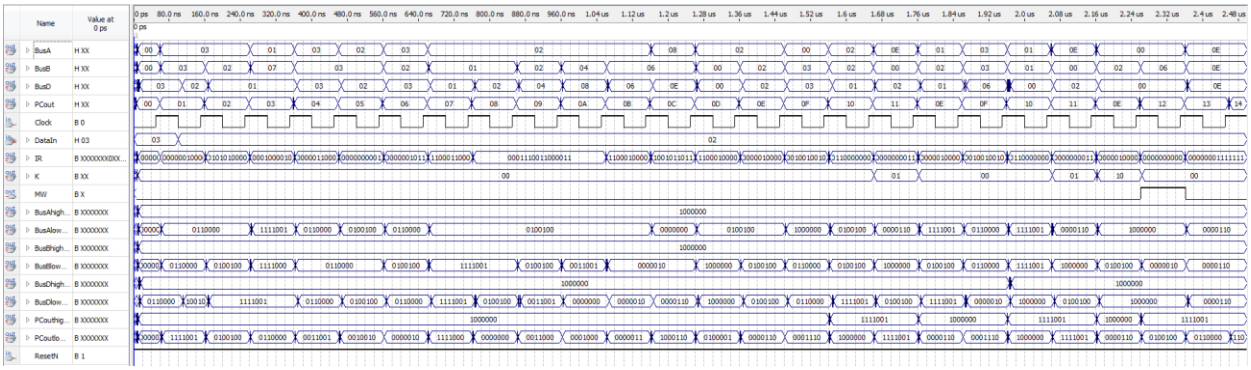
Here is the Verilog code for the circuit.

```
52  always@(Address)
53  begin
54     case(Address)
55
56     //Setup
57        0: IR <= {LOAD, R0, NULL, NULL};    //Loading the factoral number into register zero.
58        1: IR <= {MOVA, R1, R0, NULL};      //Moving the number in register zero to register one.
59        2: IR <= {DEC, R1, R1, NULL};       //Decrementing the number in register one.
60        3: IR <= {LDI, R2, NULL, ZERO};     //Loading zero into register two.
61        4: IR <= {LDI, R3, NULL, 3'b110};   //Loading the jump address six into register three.
62        5: IR <= {BRZ, 3'b001, R1, 3'b111}; //If the number in register one is zero the program will jump to line 15.
63     //Multiplication Loop
64        6: IR <= {BRZ, 3'b001, R0, 3'b010}; //If the number in register zero is zero the program will jump to line 10.
65        7: IR <= {ADD, R2, R2, R1};         //Adding the number in register two and one and storing the result in register two.
66        8: IR <= {DEC, R0, R0, NULL};       //Decrementing the number in register zero.
67        9: IR <= {JMP, NULL, R3, NULL};     //Jumping to line 6.
68     //Decrement/Setup for next loop
69        10: IR <= {MOVA, R0, R2, NULL};     //Moving the number in register two to register zero.
70        11: IR <= {DEC, R1, R1, NULL};      //Decrementing the number in register one.
71        12: IR <= {LDI, R2, NULL, ZERO};    //Loading zero to register two.
72        13: IR <= {BRZ, 3'b001, R1, 3'b111};//If the number in register one is zero the code will branch to line 15.
73        14: IR <= {JMP, NULL, R3, NULL};    //Jumping to line 6.
74     //Store
75        15: IR <= {ST, NULL, NULL, R0};     //Storing the answer held in register zero.
76     default
77        IR <= 255;
78     endcase
79  end
80  endmodule
```

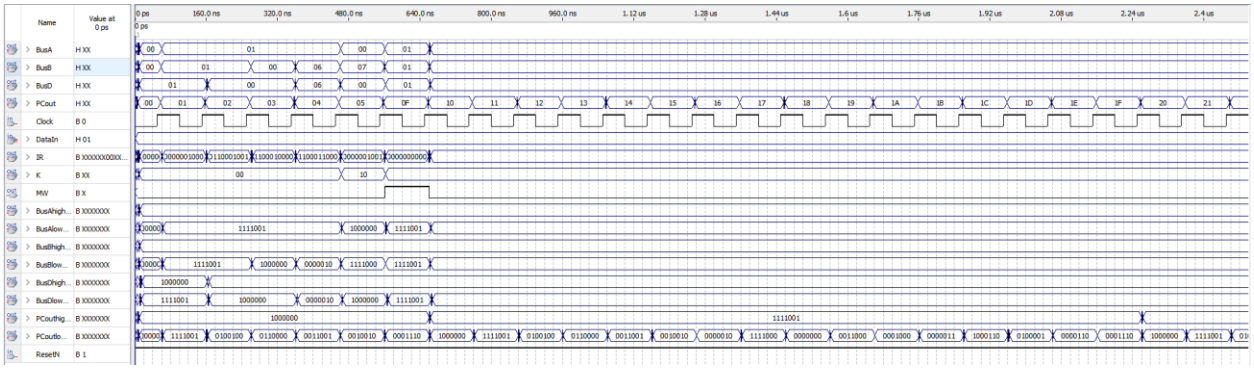
Testing

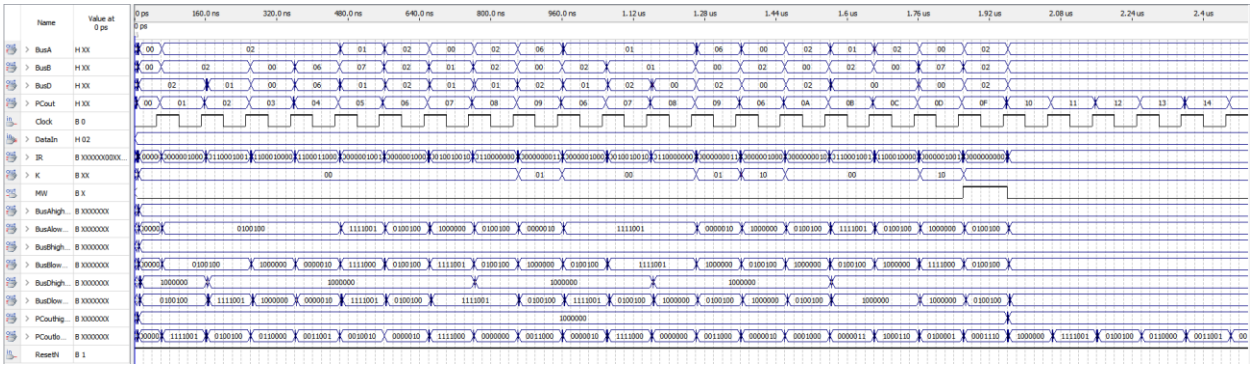
While testing the Multiplier with Comparison I encountered no problems and it performed as expected for every single input combination.

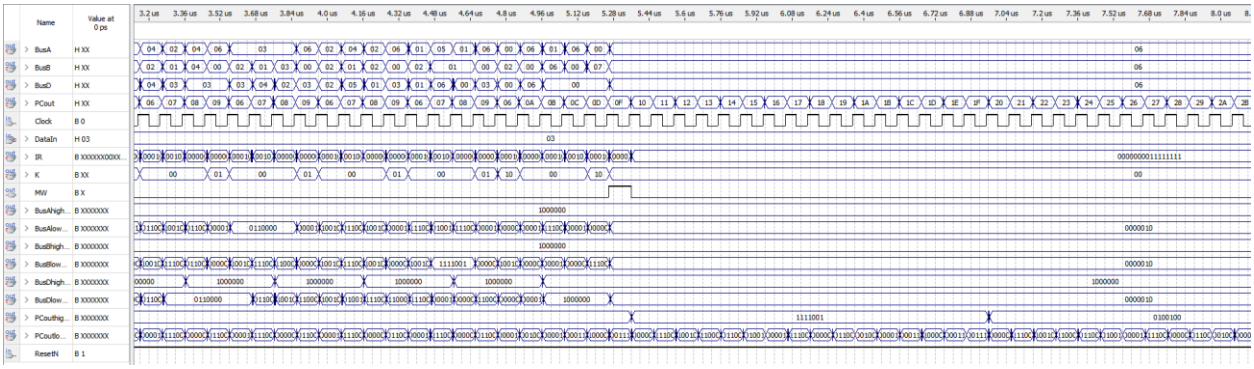




When testing the factorial program I ran into a few problems. For example, because the code takes so long to perform its task simulating the circuit proved difficult at times. I was able to successfully get correct answers for the numbers 1, 2, and 3. Whenever I tried to test either 4 or 5 Quartus would fail to simulate. Here are the waveforms for 1, 2, and 3 in that order.







Conclusion

In this lab, I learned how to program our single cycle computer. I learned the syntax and rules that apply when you try to create a program in assembly language. This proved to be an eye-opening experience for me as I have always stuck to high-level languages. As this is the last lab for this course I find myself being a little bummed that it is over. I have greatly enjoyed this class and all the experience I have received with simple and complex circuits. I will differently be taking an assembly language course next semester. Overall, I enjoyed this lab and feel like I can confidently create assembly programs for my single cycle computer.