EE 3613 – Fall Semester – AY 2017-2018

Final Project Report (teams of 2)

Due 4 December 2017 11:59 PM on Blackboard

Instructions: convert this document to a PDF and include it in the zip file with the code.

|  |  |  |
| --- | --- | --- |
| **Name** | **PID** | **% contribution to the project** |
| Alex Tillman | p100201735 | 50 |
| Zachary Tumbleson |  | 50 |

**If each team member did not contribute 50%, explain the reasons below:**

**Bonus points attempted:**

One level CLA (+5): Yes/No

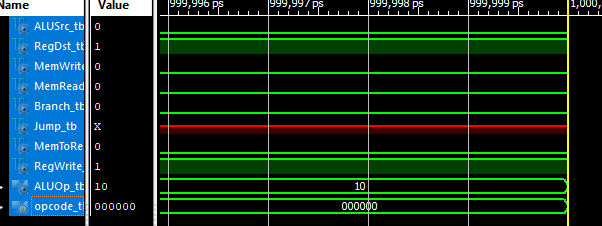
Two level CLA (+10): Yes/No

**Solutions (2-3 pages):**

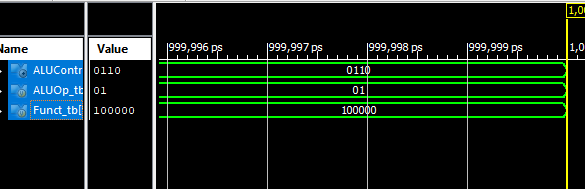
Briefly describe your solutions to all the 7 problems here. Discuss your methodology (e.g., the program your wrote for Problem 7) for each problem here and mention what worked and what didn’t. Include any waveforms from iSim where necessary.

I started working on the solution for problem 3 (Command and Control). I started by implementing a switch statement dependent on the opcode. The switch statement includes the opcodes for basic Mips functions. I created output registers to update the values in the always block. So at opcode = 6’b000000 for an R-type instruction will set the outputs.

|  |
| --- |
|  |
| RegDst = 1'b1; |
| ALUSrc = 1'b0; |
| MemToReg = 1'b0; |
| RegWrite = 1'b1; |
| MemRead = 1'b0; |
| MemWrite = 1'b0; |
| Branch = 1'b0; |
| ALUOp = 2'b10; |
| Jump = 1'bx; |
| end |

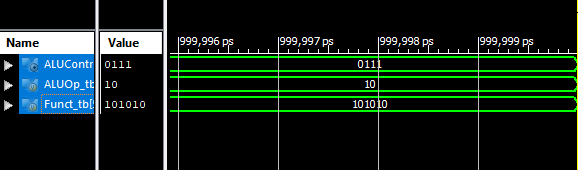


I then set the parameters to the correct values for I-type instructions including Load word and store word instructions. As well as two j type instructions Branch if Equal and Jump. After wiring up the module I created a test-bench for the control unit that sets the opcode to a 6 bit binary number and displays the output. after proving this module works i started to work on the ALUControl unit that branches off the control ALUOp wire. using another case statement I test the 2 bit input from ALUOp to control the settings for the 32 bit ALU. At ALUOp value 2’b00 the ALUControl is set to the value 0010 for ADD instruction. This is for Load Word and Store Word operations. For ALUOp value 2’b01 it sets the ALU actions to 0110 for subtract, this is used for the branch equal instruction.

For ALUOp value 2’b10 i implemented another case statement to test the value of the Funct input, this input determines what type of ALU operation will be done for the proper R-type instruction.

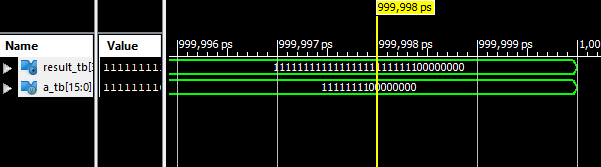
Example:

|  |  |
| --- | --- |
| case(Funct) |  |
|  | 6'b10\_0000: begin |
|  | ALUControl = 4'b0010; |
|  | end |
|  |  |



After this implementation I created another test bench for this process that’s sets the ALUOp and Funct values.

Once the control was implemented i moved on to Problem 2 (Bits and Pieces) and started by implementing a 16bit to 32bit sign extension unit. You have to basically concatenate the upper 16 bits of the result value with the MSB of the input value, with the rest of the [15:0]bit input.



The 32bit 2to1 Mux operates when the input op changes. depending on the binary signal from op, the output result will equal the value from input a or value from input 2. Because the code for both the

|  |  |
| --- | --- |
| case(op) |  |
|  | 1'b0: result = a; |
|  | 1'b1: result = b; |
|  | endcase |

|  |  |
| --- | --- |
| begin |  |
|  | if (op ==0) |
|  | result = a; |
|  | else |
|  | result = b; |
|  | end |

For the 5Bit2To1 Mux i used a if statement instead.

The next unit i worked on following the Bits and pieces is the Problem 5 (Program Counter and Friends). These are simple solutions

ProgramCounter:

|  |  |
| --- | --- |
| always @(posedge Clk) |  |
|  | begin |
|  | assign PCOut = PCIn; |
|  | end |

PCAdder:

|  |  |
| --- | --- |
| always @(\*) |  |
|  | begin |
|  | assign PCOut = PCIn +4; |
|  | end |
|  | endmodule |

LeftShifterTwoBits:

This modules assigns the output value as a concatenation of the upper 26 bits replaced by the input valueIN and the lower 2 bits with 2 zeroes.

|  |  |
| --- | --- |
| always @(\*) |  |
|  | begin |
|  | assign ValueOut = {ValueIn, 2'b00}; |
|  | end |

LeftShifterTwoBitsDiscared:

Very similar to the normal left shifter except both input and output are 32 bits, You concatenate the input of lower 30bits of ValueIN into the upper 30 bits of valueOUt and once again add 2’b00 to the lower two bits.

|  |  |
| --- | --- |
| always @(\*) |  |
|  | begin |
|  | assign ValueOut = {ValueIn[29:0], 2'b00}; |
|  | end |

BEQAdder: This module assigns the output value to be the sum of value1 the jumpaddress + value2 the PC and 4;

|  |  |
| --- | --- |
| always @(\*) |  |
|  | begin |
|  | assign ValueOut = ValueIn1 + ValueIn2 + 4; |
|  | end |

My final implementation was the register-file module as part of Problem 4 and my partner finished the rest of the memory section.

I created the Registerfile by creating an array of size 32, of[31:0]bit reg called Registers and filled the registers with trash values.

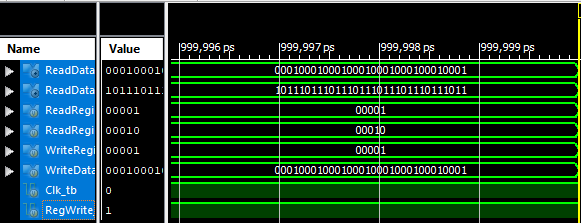
|  |  |
| --- | --- |
| initial |  |
|  | begin |
|  | Registers[0] <= 32'h00000000; |
|  | Registers[1] <= 32'hAAAAAAAA; |
|  | Registers[2] <= 32'hBBBBBBBB; |
|  | Registers[3] <= 32'h00000000; |

At the positive clock edge if the RegWrite input == 1 then WriteDAta input is written to the Register array at location giving by the input WriteRegister input :

|  |  |
| --- | --- |
| if (RegWrite ==1) |  |
|  | begin |
|  | Registers[WriteRegister] <=WriteData; |
|  | end |

At the negative clock edge you set ReadData1 to the Register at the location indicated by input ReadRegister1 and do the same for ReadData2 at location ReadRegister2:

|  |  |
| --- | --- |
| begin |  |
|  | ReadData1 <= Registers[ReadRegister1]; |
|  | ReadData2 <= Registers[ReadRegister2]; |
|  | end |
|  |  |



My partner Worked on the implementations for problem(s) 1 including the CLA EC, data memory and instruction memory modules and did most of the CPU integration connecting the modules.