Project Part 2

John Bergin

NUID: 002155500

Course: 3324 - 40456 - 202440

Introduction

Within this project, I have built off the ALU, Register, and RegFile modules from the prior portion of the project, to create a fully pipelined Risc-V-esque processor. To accomplish this, there are 3 intermittent registers between the 3 stages, and a few additional modules. These include a simple two to one multiplexor, a sign extension module, a controller, and the overarching cpu3 module to control the entire CPU. This project proved fundamental to my understanding of Pipelined CPUs, whose implementation will be seen in the following sections.

Implementation

The first part of the implementation process was to reuse the alu32.sv, register.sv, and regfile.sv files from the previous section. As they were detailed in my previous report, I will not be including them here.

The first additional module to be created is the **controller** module. This module decodes the input OpCode and Func (if necessary) returning two signals: ALUop and ALUsrc. The implementation can be seen in the image below:

```
//John Bergin
//002155500

`timescale 1ns / 1ps

module controller(input logic [5:0] ID_OpCode, ID_Func, output logic ID_ALUsrc, output logic [2:0] ID_ALUop);


//SET DEFAULT VALS
always_comb begin
//handle default case
    ID_ALUsrc = 1'b0;
    ID_ALUop = 3'b000;

     case(ID_OpCode)

     //Case for ADD and I format
     6'b000011: begin
        ID_ALUop = 3'b000;
        ID_ALUsrc = 1'b1;
     end

     //Case for AND and I format
     6'b001111: begin
        ID_ALUop = 3'b111;
        ID_ALUsrc = 1'b1;
     end


     //Case for R type Instrs.
     6'b000000: begin
     ID_ALUsrc = 1'b0;//always zero when OpCode 000000
        case(ID_Func)
           6'b000011:
            ID_ALUop = 3'b000; //ADD
           6'b000010:
           ID_ALUop = 3'b001; //SLL
           6'b000111:
            ID_ALUop = 3'b111; //AND
        endcase
     end

     default:begin
        //NO OP
        ID_ALUsrc = 1'b0;
        ID_ALUop = 3'b000;

     end
     endcase
end
endmodule

```

Figure 1: the controller.sv file

As shown, this file follows the following data, provided within this project instruction file:

Inputs		Outputs	
OpCode	Func	ALUop	ALUsrc
000011	x	000 (ADD)	1 (for I format)
001111	x	111 (AND)	1 (for I format)
000000	000011	000 (ADD)	0 (for R format)
000000	000010	001 (SLL)	0 (for R format)
000000	000111	111 (AND)	0 (for R format)

Figure 2: the opcode decoding chart

To accomplish this, a simple case statement is used to decide between the necessary opcodes, setting ALUop and ALUsrc to the corresponding outputs as shown by the chart. If the opcode happens to be 6'b000000, then there is another case statement for the Func, in which the outputs are then selected per figure 2. This concludes the controller implementation.

The next module that was instantiated was a simple **2 to 1 mux**, with a parameter such that the input and output data widths can change as it must be implemented twice. To accomplish this, I wrote the following code:

```
1  //John Bergin
2  //002155500
3
4  `timescale 1ns / 1ps
5
6  module mux2to1#(parameter size = 32)(input logic [size-1:0] in1, in0, input logic sel, output logic [size-1:0] out);
7  assign out= sel? in1: in0; //sel is 1, in1 chosen, else in0
8  endmodule
9
```

Figure 3: the mux2to1.sv file

As shown, this simple code elects to set the output data (out) to either in1 or in0, depending on whether sel is 1 or 0, respectively.

The next module that was created was a simple **sign extender** module, capable of sign extending a 16 bit value to 32 bits. Below is its implementation below:

```
1  //John Bergin
2  //002155500
3
4  `timescale 1ns / 1ps
5  module signext16to32(input logic [15:0] in, output logic [31:0] out);
6
7      always_comb begin
8          out = {{16{in[15]}}, in}; //append the input to 16 bits matching the MSB
9      end
10 endmodule
11
```

Figure 4: the sign extension module

As shown, this simple code appends the signal with 16 bits of the same value (1 or 0) as it's 16th bit, thus sign extending the value.

The final portion of this implementation, and the trickiest, is the **cpu3** module. This overarching module takes in the clk input, the ibus, and outputs the write data to the register file. Thus, its implementation is very important as it uses all prior instantiated modules. This must also decode the following input manner:

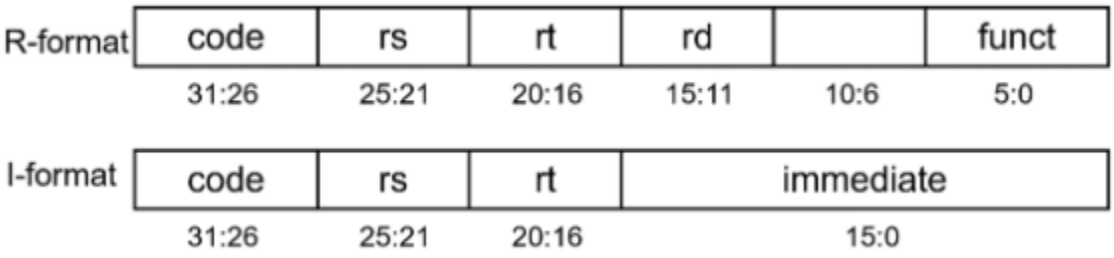


Figure 5: the ibus input formats

To accomplish this, the following code was created:

```
1  //John Bergin
2  //002155500
3
4  `timescale 1ns / 1ps
5
6  module cpu3(input logic [31:0] ibus, input logic clk, output logic [31:0] WD);
7
8  //create necessary logic for each interconnection between registers, modules, inputs/outputs
9  logic ID_ALUsrc, EX_ALUsrc, V, Z, Cout;
10 logic [2:0] ID_ALUop, EX_ALUop;
11 logic [4:0] ID_rd, ID_rt, ID_rs, WR, ID_WR, EX_WR;
12 logic [5:0] ID_Func, ID_OpCode;
13 logic [15:0] ID_immv16;
14 logic [31:0] ID_RD1, ID_RD2, ID_immv32, EX_RD1, EX_RD2, EX_WD, EX_imm, ALU_opB;
15
16 //Create IFID registers:
17 register #(5) IFID_rd (.clk(clk), .D(ibus[15:11]), .Q(ID_rd));//store rd signal
18 register #(5) IFID_rt (.clk(clk), .D(ibus[20:16]), .Q(ID_rt));//store rt signal
19 register #(5) IFID_rs (.clk(clk), .D(ibus[25:21]), .Q(ID_rs));//store rs signal
20 register #(6) IFID_OpCode (.clk(clk), .D(ibus[31:26]), .Q(ID_OpCode));//store OpCode signal
21 register #(6) IFID_Func (.clk(clk), .D(ibus[5:0]), .Q(ID_Func));//store Func, signal
22 register #(16) IFID_Imm16 (.clk(clk), .D(ibus[15:0]), .Q(ID_immv16));//store immv16 signal
23
24 //Create IDEX registers
25 register #(1) IDEX_ALUsrc(.clk(clk), .D(ID_ALUsrc), .Q(EX_ALUsrc)); //store ALUsrc signal
26 register #(3) IDEX_ALUop(.clk(clk), .D(ID_ALUop), .Q(EX_ALUop)); //store ALUop signal
27 register #(5) IDEX_WR(.clk(clk), .D(ID_WR), .Q(EX_WR)); //store WR signal
28 register #(32) IDEX_RD1(.clk(clk), .D(ID_RD1), .Q(EX_RD1)); // store RD1 signal
29 register #(32) IDEX_RD2(.clk(clk), .D(ID_RD2), .Q(EX_RD2)); // store RD2 signal
30 register #(32) IDEX_Imm32(.clk(clk), .D(ID_immv32), .Q(EX_imm)); //store imm signal
31
32 //Create EXMEM registers
33 register #(32) EXMEM_WD(.clk(clk), .D(EX_WD), .Q(WD)); //store WD signal
34 register #(5) EXMEM_WR(.clk(clk), .D(EX_WR), .Q(WR)); //store WR signal
35
36 //module instantiations for the ID stage
37 controller ctrl (.ID_OpCode(ID_OpCode), .ID_Func(ID_Func), .ID_ALUsrc(ID_ALUsrc), .ID_ALUop(ID_ALUop));//create controller module
38 mux2to1 #(5) ID_mux(.in0(ID_rd), .in1(ID_rt), .sel(ID_ALUsrc), .out(ID_WR)); //create EX mux
39 signext16to32 signExt (.in(ID_immv16), .out(ID_immv32)); //create sign extension module
40 regfile RegFile (.RR1(ID_rs) , .RR2(ID_rt), .WRM(WR), .WD(WD), .clk(clk), .RD1(ID_RD1), .RD2(ID_RD2)); //regfile instantiation
41
42 //module instantiations for the EX. stage
43 alu32 ALU (.a(EX_RD1), .b(ALU_opB), .ALUop(EX_ALUop), .d(EX_WD), .Cout(Cout), .V(V), .Z(Z)); //ALU assignment
44 mux2to1 #(32) EX_mux (.in0(EX_RD2), .in1(EX_imm), .sel(EX_ALUsrc), .out(ALU_opB)); //EX mux assignment (32 bits)
45 endmodule
46
```

Figure 6: the cpu3.sv module

As seen, the first step is to instantiate a tremendous amount of logic segments, which interconnect the registers with other modules, and decode the ibus. Next, the ibus is decoded, with logics code, rs, rt, rd, funct, and imm. These allow for the data to be passed further into the cpu3 module accordingly. These values are then passed into the IFID register, under the corresponding ID_XXX values (for example rt becomes ID_rt). From the register, these are then passed into the various modules of the ID section, which outputs a next host of logic signals to be passed to the IDEX register, including going through the controller, passing RD1 and RD2, sign extending the immediate, and determining the WR value.

Next, they are passed into the IDEX register. From this register, the corresponding logic signals are passed into the APU and the mux to decide the ALU input (either the immediate

or RD2. Next, the Write Register and Write Data are then passed to the EXMEM register, where they are passed to the regfile on the next corresponding cycle to be written.

To implement this in code, many logic signals were created to interconnect the modules. These modules were then instantiated with the corresponding logic signals, and finally the ibus signal was decoded into the corresponding signals as shown above. This was then ran on the cpu3_testbench. The following success message was received:

```
  ibus=xxxxxx xxxxx xxxxx xxxxx xxxxxxxxxxx for instruction          13
Time=     280.0ns
  clk=1
Time=     290.0ns
  clk=0
  Testing output operand for instruction          11
    Your WD =    00000000000000000000000000010011
    Correct WD = 00000000000000000000000000010011
---------YOU DID IT!! SIMULATION SUCCESFULLY FINISHED----------
INFO: [USF-XSim-96] XSim completed. Design snapshot 'cpu3_testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 7: the success message

The following was the first waveform output:

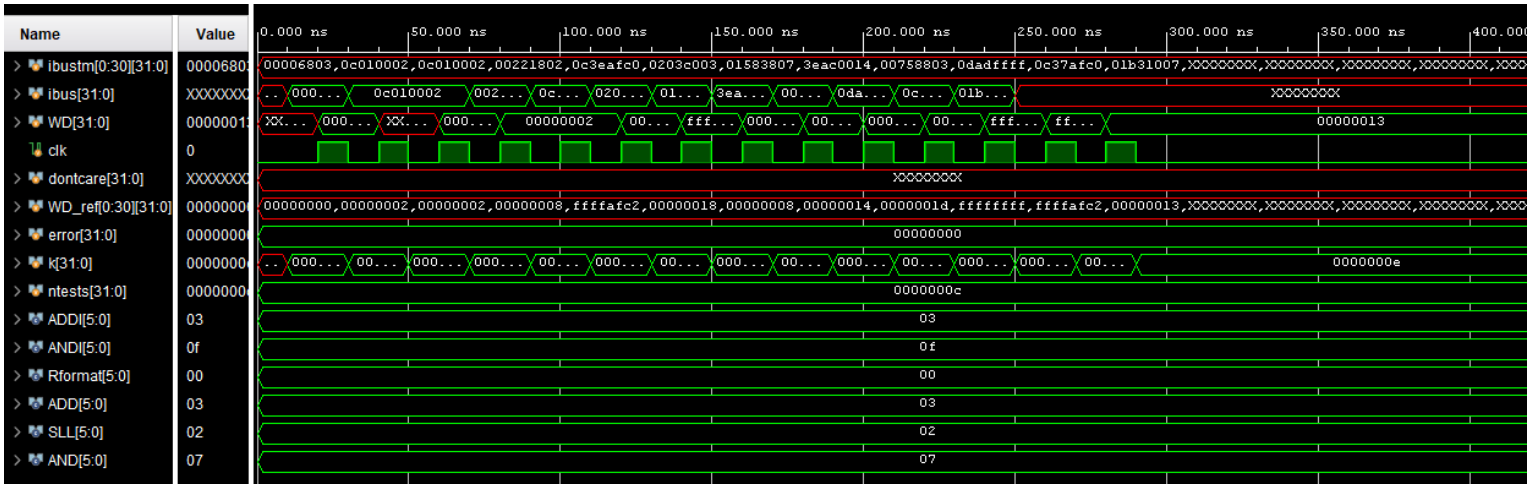


Figure 8: the first waveform output

Next, the clk and WD signals were added, yielding the following output tests 6, 7, and 8:

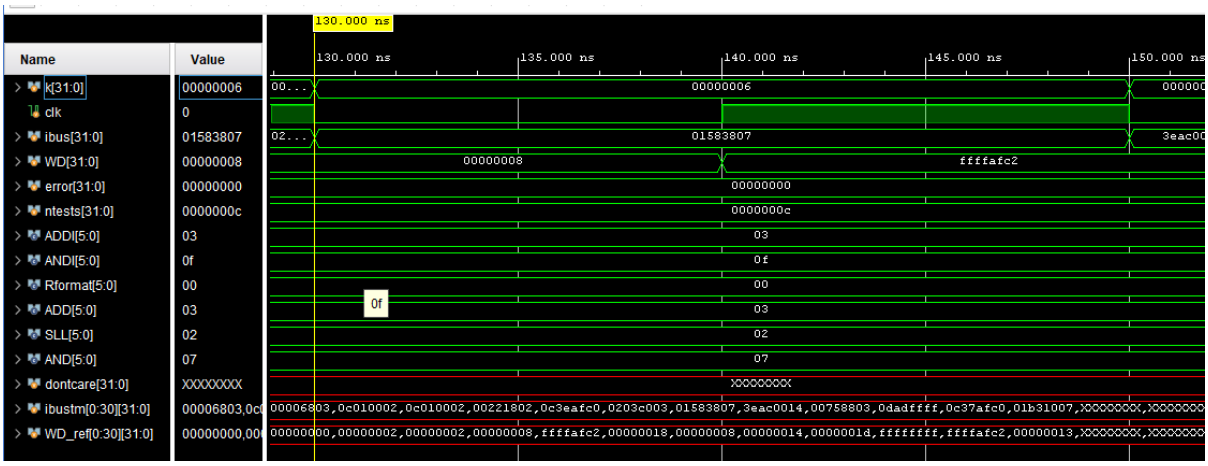


Figure 9: the start time for Index 6

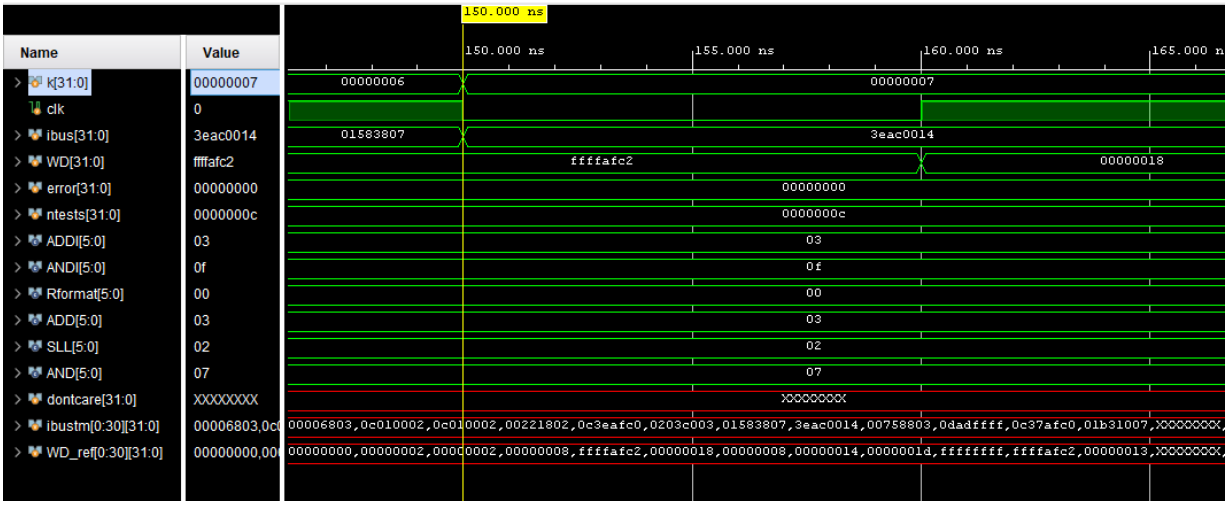


Figure 10: the start time for Index 7 and end time for Index 6

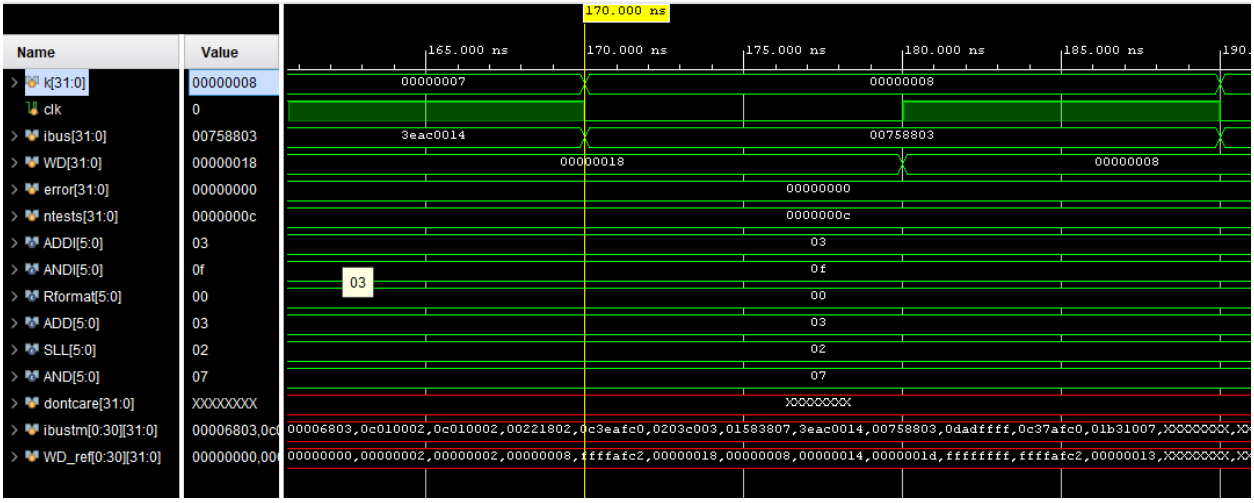


Figure 11: the start time for Index 8 and end time for Index 7

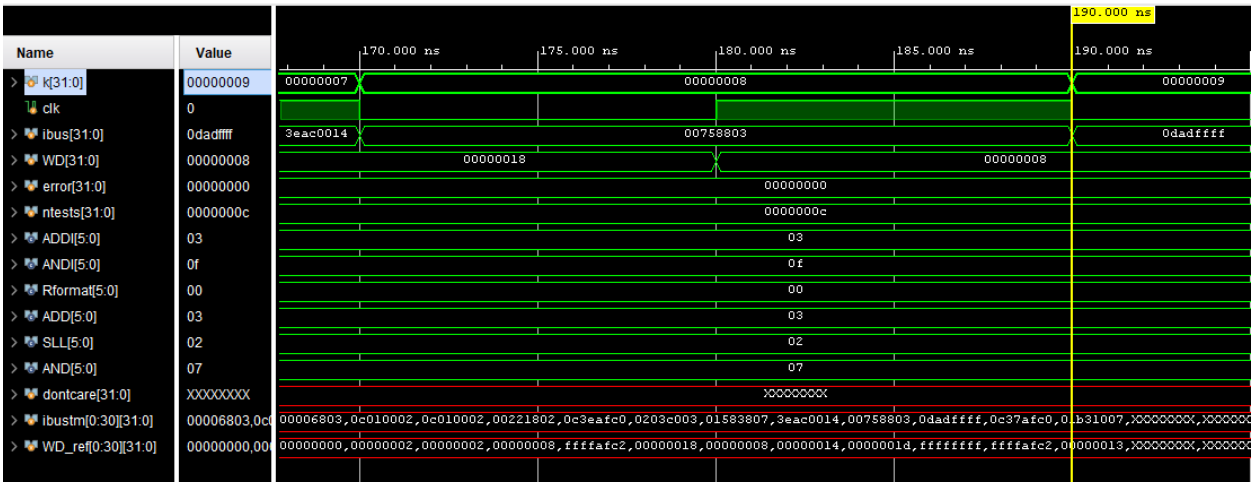


Figure 12: Index 8 end time

As shown above, Index 6 begins at 130 ns, ending at 150 ns where the ibus value is 0x01583807. This reflects the instruction AND R7, R10, R24. The expected WD value is 0x00000008, calculated through a bitwise AND operation of values 0x10 (in R10) and 0x08 (in R24).

Index 7 begins at 150 ns, ending at 170 ns where the ibus value is 0x3EAC0014. This reflects the instruction ANDI R12, R21, #0014. The expected WD value is 0x00000014 which is simply the immediate value.

Index 8 begins at 170.00 ns and ends at 190.00 ns where the ibus value is 0x00758803, which represents ADD R17, R3, R21. The expected WD value is 0x0000001D, which is calculated by adding 0x08 (in R3) and 0x15 (in R21).

These tests confirm that the WD signal is accurate and expected, showing the correct behavior from the entire pipelined module. It is relevant to note the 2-cycle delay seen by the registers to account for the 2 stages in between reading and writing.

Finally, 5 more testbench instructions were added, and their implementation and results can be shown below. As shown in figure 13 below, the testbench has a specific format to allow for easy instruction implementation. The data ibustm serves to connect to ibus, such that the opcode can be denoted by the necessary operation, depending on whether or not it is I or R type. If it is R format, then 3 registers are used (immediate only uses 2 along with the immediate value), then the end (Funct) contains the operation, either AND, ADD, or SLL. This allows for simplistic implementation and can be seen in the sample 5 new instructions seen below. Then, the expected WD value (WD_ref) is compared with the actual WD, to ensure it is correct. If not, it executes and error and cites which instruction

caused the error. The code is commented with the corresponding operations performed and necessary registers:

```
--------------------------------------------------------------------------------------------
  ibus=000000 00010 10111 00010 00000000111 for instruction          14
Time=     300.0ns
  clk=1
Time=     310.0ns
  clk=0
  Testing output operand for instruction          12
    Your WD =    00000000000000000000000000000000
    Correct WD = 00000000000000000000000000000000
--------------------------------------------------------------------------------------------
  ibus=000000 00010 00000 01111 00000000011 for instruction          15
Time=     320.0ns
  clk=1
Time=     330.0ns
  clk=0
  Testing output operand for instruction          13
    Your WD =    11111111111111111010000011000000
    Correct WD = 11111111111111111010000011000000
--------------------------------------------------------------------------------------------
  ibus=000000 10111 00000 10111 00000000111 for instruction          16
Time=     340.0ns
  clk=1
Time=     350.0ns
  clk=0
  Testing output operand for instruction          14
    Your WD =    00000000000000000000000000000010
    Correct WD = 00000000000000000000000000000010
--------------------------------------------------------------------------------------------
  ibus=xxxxxx xxxxx xxxxx xxxxx xxxxxxxxxxx for instruction          17
Time=     360.0ns
  clk=1
Time=     370.0ns
  clk=0
  Testing output operand for instruction          15
    Your WD =    00000000000000000000000000010011
    Correct WD = 00000000000000000000000000010011
--------------------------------------------------------------------------------------------
  ibus=xxxxxx xxxxx xxxxx xxxxx xxxxxxxxxxx for instruction          18
Time=     380.0ns
  clk=1
Time=     390.0ns
  clk=0
  Testing output operand for instruction          16
    Your WD =    00000000000000000000000000000000
    Correct WD = 00000000000000000000000000000000
---------YOU DID IT!! SIMULATION SUCCESFULLY FINISHED----------
INFO: [USF-XSim-96] XSim completed. Design snapshot 'cpu3_testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 13: the success message after testing

```
112
113     //NEW TESTS SEEN BELOW//
114
115     // ADDI R13, R13, 1 (R13 currently holds value -1)
116     // ---------
117     //         opcode source1   dest      Immediate...
118     ibustm[12]={ADDI, 5'b01101, 5'b01101, 16'h0001};
119     WD_ref[12]=32'h00000000;
120
121
122     // ANDI R23, R23, #A0C0
123     // ---------
124     //         opcode source1   dest      Immediate...
125     ibustm[13]={ANDI, 5'b10111, 5'b10111, 16'hA0C0};
126     WD_ref[13]=32'hFFFFA0C0;
127
128
129     // AND R2, R2, R23
130     // --------
131     //          opcode   source1   source2   dest      unused     Function...
132     ibustm[14]={Rformat, 5'b00010, 5'b10111, 5'b00010, 5'b00000, AND};
133     WD_ref[14]=32'h00000002;
134
135     // ADD R15, R2, R0
136     // ----------
137     //         opcode   source1   source2   dest      unused     Function...
138     ibustm[15]={Rformat, 5'b00010, 5'b00000, 5'b01111, 5'b00000, ADD};
139     WD_ref[15]=32'h00000013;
140
141
142     // AND R23, R23, R0
143     // ---------
144     //          opcode   source1   source2   dest      unused     Function...
145     ibustm[16]={Rformat, 5'b10111, 5'b00000, 5'b10111, 5'b00000, AND};
146     WD_ref[16]=32'h00000000;
147
148
149     ntests = 17;
```

Figure 14: the updated testbench

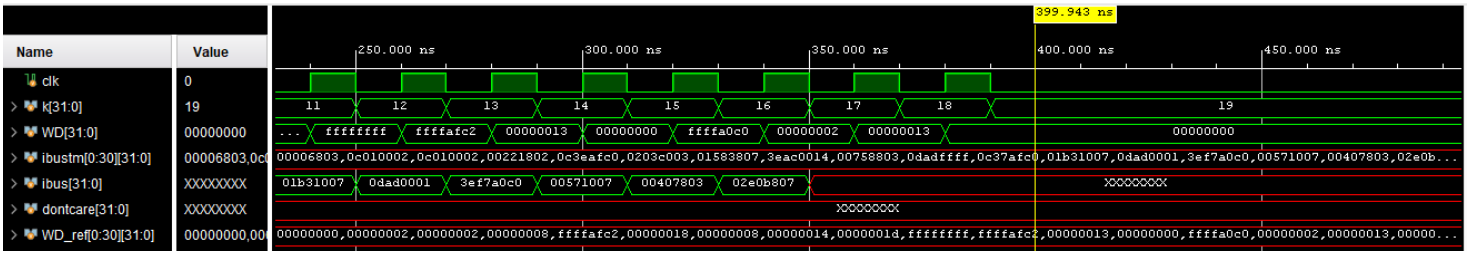
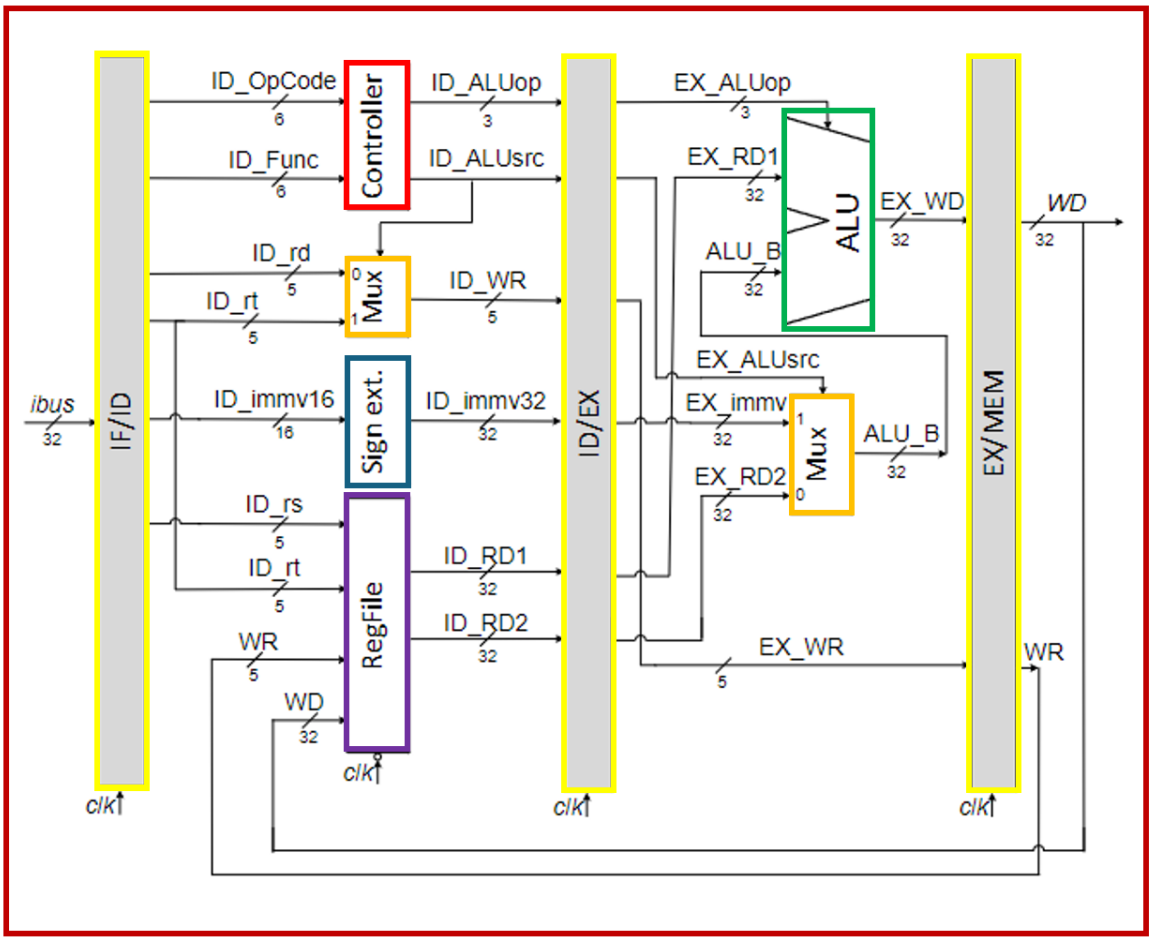


Figure 15: the output waveform with the additional 5 tests

Block Diagram

As shown below, this is the block diagram for this entire implementation. The “KEY” seen to the right details the specific module corresponding to each block, in a color-coordinated fashion.



KEY

- mux2to1.sv
- regfile.sv
- controller.sv
- signext16to32.sv
- regfile.sv
- alu32.sv
- cpu3.sv

Conclusion

This project consisted of creating a pipelined CPU capable of decoding 32-bit instructions to perform simple operations such as AND, ADD, and SLL. This proved fruitful in my learning experience of this course as it showed hands-on how the pipelined CPU functions. Regarding future modifications to this project, as I mentioned in the previous section is the possibility of additional operations for the ALU to perform. This would allow even more instructions to occur which would be helpful for the learning process. Aside from this, this project was beneficial to my learning experience as a whole.