CPU Control

Memory Interface and CPU Control
Fetch + Decode + Control + Datapath integration

Phillip Bradbury, Zach Toolson, Dan Willoughby

November 12, 2013

Contents

1 Overview

The combination of different modules makes the CPU able to decode assembly instructions, store data in memory, and perform operations. The decoder and controller deal with the parsing of instructions and setting the correct control signals. Dual port memory allows for the memory to be read and written to at the same time. The program counter and instruction FSM work together so that the CPU can execute the various instructions loaded into memory. The layout of the CPU can be seen in Figure 1. The red lines show the control signals, while the black show the flow of the data. Each component, described in detail below, plays an important part in the overall operation of the CPU.

2 Memory

2.1 Design

We designed our memory to meet the needs of our duck hunt game. We wanted two separate 16384x18-bit dual-port RAM blocks to allow us to have 4 ports for reading and writing. The reason why we chose to have two RAM blocks, instead of one RAM block(32 blocks of 1024 was the max of the nexys 3 block ram), was so that we could have audio in one RAM block, and game information in the other.

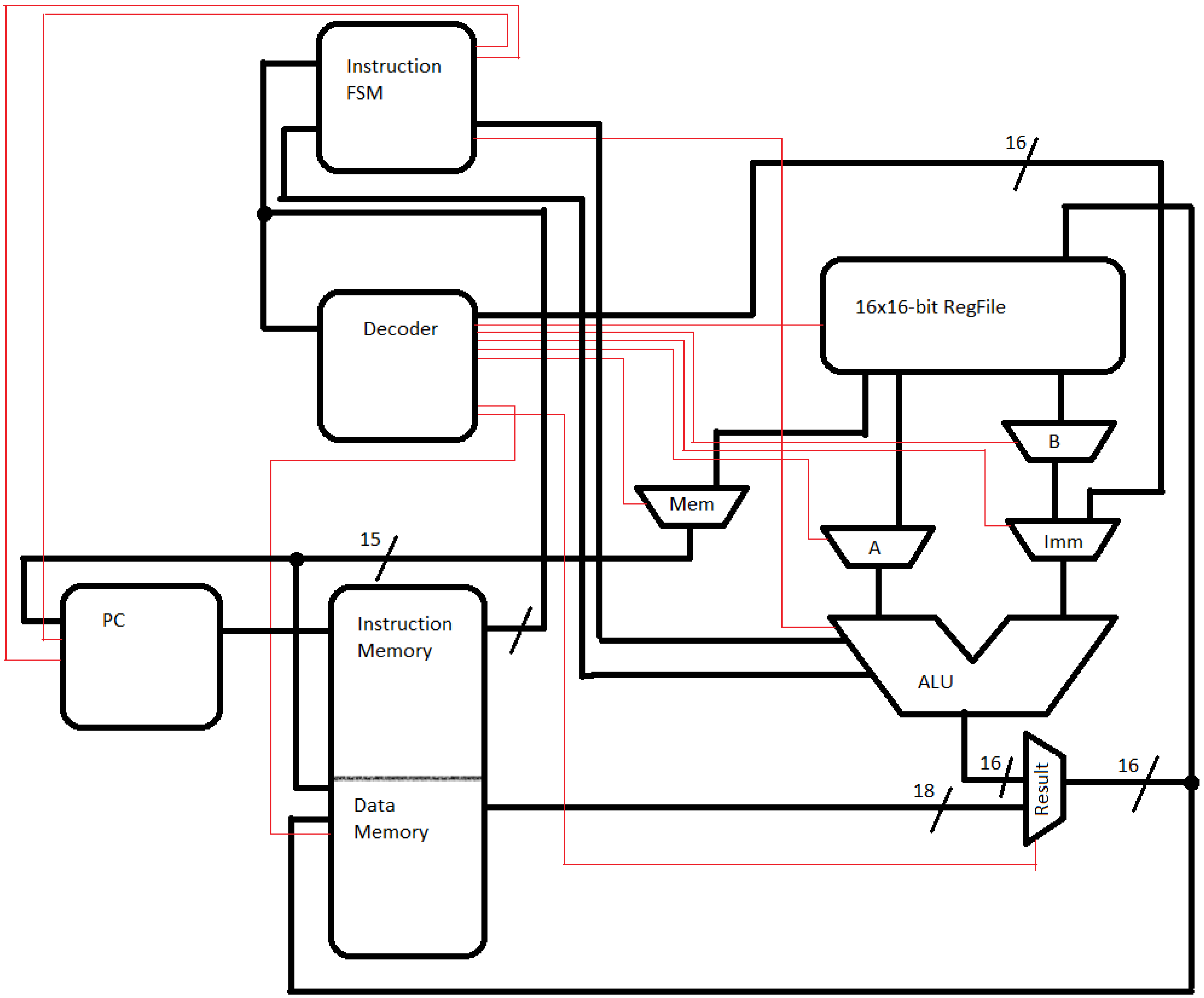
We designed our memory to be a synchronous dual-port block RAM. The dual-port memory was so that our program counter could continually read the assembly instructions of the program from one port, while still reading and writing real-time data for the duck hunt game. We established the convention that the top portion of a memory block was the assembly instructions, while the remaining was left for the game data.

2.2 Implementation

Our implementation first had a flawed design, but then we reworked it to get one that met our needs. Our initial implementation included 30 separate blocks of memory. Each block was 1024 registers and we wanted to tie 30 blocks together, leaving two blocks for audio. The idea behind having 30 blocks and 2 blocks, was so that we could have two ports for the 30 blocks and two ports for the remaining 2 blocks. The initial implementation had issues with the $readmemh command. Because of the instantiation issue we decided to redesign the memory from scratch.

The next implementation of our memory followed closely to the example code given in lab. The code includes two read/write ports, each having its own data in and data out bus. It also had a separate clock for each port. We decided when we instantiated the memory module, to pass in the same clock as an argument. Using the same clock for both memories had no negative effect on the function of the memory. The memory module used two parameters for the data and the address. The parameters were set to a default value in the module. When instantiating the module to use the programmer can then change the variable values if desired. We set two parameters to the values of data = 18, and addr = 14. These numbers are based on the Block-RAM design of the Spartan 6 board, which gave us 32 blocks of RAM, with each block having 18k bits. Xilinx tools requires a special syntax to declare the memory, which we followed using the reg [DATA-1:0] mem [(2**ADDR)-1:0] command. We then use the $readmemh command to

Figure 1: Schematic of CPU



read in a file and instantiate the memory with our game code at compile time. To implement our read/write memory design in Verilog we check if the memory should be writing or reading. When the positive edge of the clock hits, check to see if the memory is set to write. If it is, read the data out first before writing the incoming data, otherwise just read the data out of the address requested. This is a read-first memory design. With this second design, out memory worked properly and met the needs of our game design.

2.3 Troubleshooting

Initially, we had one 32 block RAM memory to use, but we wanted to have additional read/write ports. We started off by instantiating 30 blocks, but it didnt synthesize to one block RAM. It instead was synthesized to distributed RAM. Our next thought to overcome the distributed RAM problem was to instantiate 30 blocks separately and tie them together with logic (huge case statement to direct each memory address to the correct block). We realized when we implemented the memory with our CPU that we could not instantiate memory with the assembly code. The reason we were getting distributed RAM when we had 30 blocks, was because the Xilinx tools require certain syntax when instantiating memory. Among these syntax rules, was that memory parameters such as data and address. We also figured out with the help of the TA that the memory needed to be instantiated with a power of two address (reg [DATA-1:0] mem [(2**ADDR)-1:0];). We decided to have two 16 block memories giving us the extra ports that we wanted.

3 Instruction FSM

3.1 Design

The instruction FSM design was meant to give a life cycle for each assembly instruction. Depending on which instruction was read from memory, it took a longer or shorter amount of time to execute it. For example, a stor/load instruction can take up to 4 clock cycles while a non memory instruction takes only 3 clock cycles. We needed an FSM to account for these differences so that each instruction can work. We designed our FSM to be a moore finite state machine so the output only depended on the current state. The instruction FSM interacted with the PC counter with control signals. The FSM has three main functions. One function that the FSM did was set the PC counter to an address which was read from a register(branches and jump). Another function was to increment the PC counter(for the next assembly instruction). The last main function was to interact with the ALU flags. Each state in the FSM corresponds to its function described in 1. The next state logic of the FSM ensured that each instruction was allowed enough time to execute fully.

3.2 Implementation

The instruction FSM was implemented using a moore finite state machine design. We had present state logic, next state logic, and output logic each with its own associated always block statements. The present state logic had a sensitivity list to execute on the positive edge of the clock. If it was set to clear, then the present state would be set to fetch and the flags would be reloaded. Otherwise, the present state would be set to the next state. We also included some

Table 1: Description of instruction FSM states

State	Description
fetch	Initial state, sets up the address for Instruction Memory
decode	After getting the data-out from the Instruction Memory
alu	After decode state, then goes straight to fetch
stor1	State after decode which waits several cycles
stor2	When output recieved, sets up the instructions to push to the register
load1	State after decode which waits several cycles
load2	When output of memory is recievied signals the data was loaded
jump	Sets up the program counter with the new address

logic to fetch the flags from the ALU which made it so it took two cycles for the flags to actually change. We verified through testing that the flags were being set properly and it had no negative effect with our state machine.

The next state logic always block determined the next state based off the the present state. For example, if the present state was fetch, the next state would be set to decode. Likewise, if the present state was decode, the next state would be determined by the decoded instruction. In this case, if it was a load instruction it would set to the load1 state(see 1). The rest of the next state states were determined in a similar fashion, always with the end result of the next state being set to the fetch state.

The output logic block set the control signals to the ALU and the PC counter. For certain instructions such as branch, the FSM checked the flags set by the ALU. If it was a BEQ instruction, the FSM would determine if the zero flag was high and set the appropriate control signals. We also included some logic for compare operations, so that the flags had ample time to be set.

3.3 Troubleshooting

While implementing the instruction FSM, we encountered several issues. Originally we had a design that did not include an instruction FSM. The instruction was decoded and executed in the same clock cycle. Our design worked for simple instructions such as and, sub, and, xor etc. But it failed for the load and store instructions. We thought we could have our assembler create duplicate load/store instructions to compensate the for time delay of the memory. We found, however, that we could not stop the program counter from incrementing to the next instruction. This was a problem, because when a load/store instruction was encountered, it would execute the next assembly instruction multiple times. This was resolved by creating the instruction FSM which allowed for instructions to have a state.

Another main issue we encountered was that was that we did not have enough states for our instructions. This was a problem because the compare operation required a different amount of time than a simple ALU instruction. As a result, the problem encountered was that the branch instructions were not working correctly. For example, a BEQ would always branch instead of only branching when they are equal (zero flag =1). We resolved the problem by creating additional states for the compare instructions. The additional states allowed for the flags to be set and read correctly for the branch or jump instructions.

Another main issue we encountered was with the load and store instructions not reading or loading the correct register. The incorrect register caused the output of the ALU to be undetermined (XXX). The undetermined output was an issue because we could not communicate with the reg file the appropriate value or address to be extracted from memory. We discovered that our datapath from memory that connected to the ALU-out datapath needed to have additional logic (needed a mux). The additional logic ensured that we were setting the memory data-out to the ALU-out datapath at the correct time. After we overcame these issues, the instruction FSM functioned properly.

4 Decoder

4.1 Design

The purpose of our decoder was to abstract away the decoding of the assembly instructions and send control signals to other components of the CPU. We had it separate from the instruction FSM so that it was easier to debug. Each instruction was parsed by 4 bit chunks. It would read the opcode, opcode extension, the register destination / source / address, and immediate value. We had the decoder deal with reading 2s complement immediate values. For example, if the instruction was addi we would extend the most significant bit sign to fill up the remaining 16 bits.

After the assembly instruction was parsed, it would set the control signals corresponding to the instructions opcode. The decoder was responsible for communicating with the reg file, the memory, and muxes that control the flow of the datapath.

4.2 Implementation

The implementation of the decoder required using a case statement to set each control signal. The register control signals needed to communicate to the reg file when to write and which register to read or write to or from, respectively. The first case statement was on the top 4 bits of the instruction (15:12). These 4 bits determine if the instruction was a register type instruction (add \$1, \$2), or any other instruction (immediate, shift, compare, branch, etc). We then had another case statement of the opcode extension bits (7:4) and it would set the opcode to communicate to the ALU which registers to read and load to the reg file.

The immediate instructions were similar to the register type instruction, except it would load in an immediate value. We had a case for each instruction that dealt with immediate values. The decoder would read in the immediate value from the instruction, extend the MSB if it was 2s complement, and enabled the mux that allowed for an external immediate value to be loaded into the ALU. The immediate value would come from the decoder and be sent to the B port of the ALU. The compare and move immediate instructions were done in a similar fashion. The shift instructions required no extra logic to decode and worked similar to the register type instruction that they correlated with.

The last case was when we had a memory or branch type instruction. We decided to have the branch, jump, load, and store instructions have the same opcode. The reason was because we did not think that we would have very many branch instructions. The way the decoder differentiated between the branch instructions and the actual memory instructions (load / store) was by the opcode extension. The load instruction was responsible for enabling a control signal that switched the ALU-out bus to the B data-out port of the memory. This allowed for the value stored in memory to be loaded directly into the datapath. The store instruction was responsible for setting the write enable signal to the memory module. Both the load and store instructions set the B port address to be read/loaded. The decoders job with the branch and load instruction was to tell the reg file which registers to read and load to. The 4 LSB (3:0) specified which type of condition the branch was (equal, not equal, jump). The instruction FSM dealt with reading the flags of the ALU and which address to jump to. With the case statement on each instruction, it allowed us to easily implement the decoder.

4.3 Troubleshooting

We encountered several issues while implementing the decoder. One of the main issues was that we had a typo in one of our case statements. The typo was that it was a case statement from 17:6 instead of 17:16. As a result, there was bizarre behavior that confused us for several hours. Luckily, we were able to resolve the issue.

Another issue we encountered with the decoder was that we werent setting the control signals to the correct values. When a control signal was the wrong value, it would either read the wrong register or not branch at the appropriate time. Since we had several signals on the output of our CPU, we were able to quickly determine which signal was incorrect. Overall, the decoder was just a large case statement, so the issues were resolved quickly.

5 Program Counter

5.1 Design

The goal for our program counter was simplicity. We moved some of the complicated things into our assembly code design and dealt with the problems there. This includes signed 2s complement addresses and signed offsets. The program counter was designed so that we could set it to any address. We also allowed for the PC counter to be incremented by any value. The conventions we established were that addresses used to set the program counter must be an unsigned number. By setting some conventions, we greatly simplified our program counter.

5.2 Implementation

The program counter would increase each clock cycle by the increment value provided. The instruction FSM communicated to the PC counter how much to increment by. The main purpose behind having the increment being externally controlled was so that the instruction FSM could tell the PC counter when to increment. For example, when we did not want the program counter to increment, we would set the increment value to zero. When we did want it to increment, we would set the value to one. The external increment value also allowed us to have a jump instruction which would set the increment value to however far we wanted the program counter to jump. The increment value could be positive or negative. We used an always block with the posedge of the clock in the sensitivity list. The program counter would increment the output address by the increment value given unless an address was being set. In this case, the output address would become the input address. The instruction FSM was responsible for all of the life cycle of instructions. Therefore, we could have a relatively simple program counter.

5.3 Troubleshooting

We experienced one main issue with the program counter. With our initial FSM described we could not prevent the program counter from incrementing before an instruction was executed (see 3.3). We attempted to have a program counter decrement to remain on the previous instruction, but this just complicated things more. Our solution was to add an instruction FSM that externally controlled the incrementation for assembly instructions.

6 CPU Datapath

6.1 Design

The CPU datapaths purpose was to instantiate all the modules, and to connect them all together. We instantiated each module, and then created wires to connect the inputs and outputs of the modules to each other. For example, we needed to connect the memory to the instruction FSM, decoder, and ALU. We abstracted the decoder and instruction FSM into its own module called the controller. The controller instantiated the required muxes / wires to communicate effectively between the decoder and instruction FSM. The program counter needed to be connected to the controller and the memory. Because we abstracted most of the details away into separate modules, the CPU datapath was relatively straightforward.

6.2 Implementation

The implementation of our CPU was instantiating modules and connecting them together. We instantiated the program counter, memory, and controller and then tied their inputs and outputs to their designated places. For documentation purposes, we declared each wire that connected the modules. We also had several outputs of different signals to aide us with debugging. The final addition to out CPU was a clock divider to allow enough time for instructions to be executed on the board. With all the modules connected, the CPU was able to execute a simple program.

6.3 Troubleshooting

Since the CPU datapath was relatively simple in design, we did not have any significant issues. We did however need to double check which wires we tied into each module. Refer to appendix F.2 for the assembly program used to test all phases required for the lab.

7 Waveforms of test operation

7.1 Phase i

Loads data from memory into RegFile. Refer to Figure. 2.

7.2 Phase ii

Performs a series of arithmetic and logical operations. Refer to Figure. 3.

7.3 Phase iii and Phase iv

Stores the result into memory and reloads the result from memory into the RegFile. Refer to Figure. 4.

7.4 Phase v and Phase vi

Performs arithmetic to set flags and uses flags to perform branches. We implemented a loop that calls a method that displays the numbers 1 to 5, each iteration. Refer to Figure. 5.

7.5 Phase vii

Writes the result into memory and displays the result. Refer to Figure. 6.

8 Conclusion

The results of Lab 3 and Lab 4 were that we were able to run our own assembly programs on our CPU. We found several errors as we were testing, which we were able to correct. It would have been near impossible to test the CPU without the different components seperated into different modules. The combination of all the modules formed a complex design, which could perform interesting operations. The CPU successfully running a program we wrote was very exciting and encouraging.

Figure 2: Load data from memory into RegFile

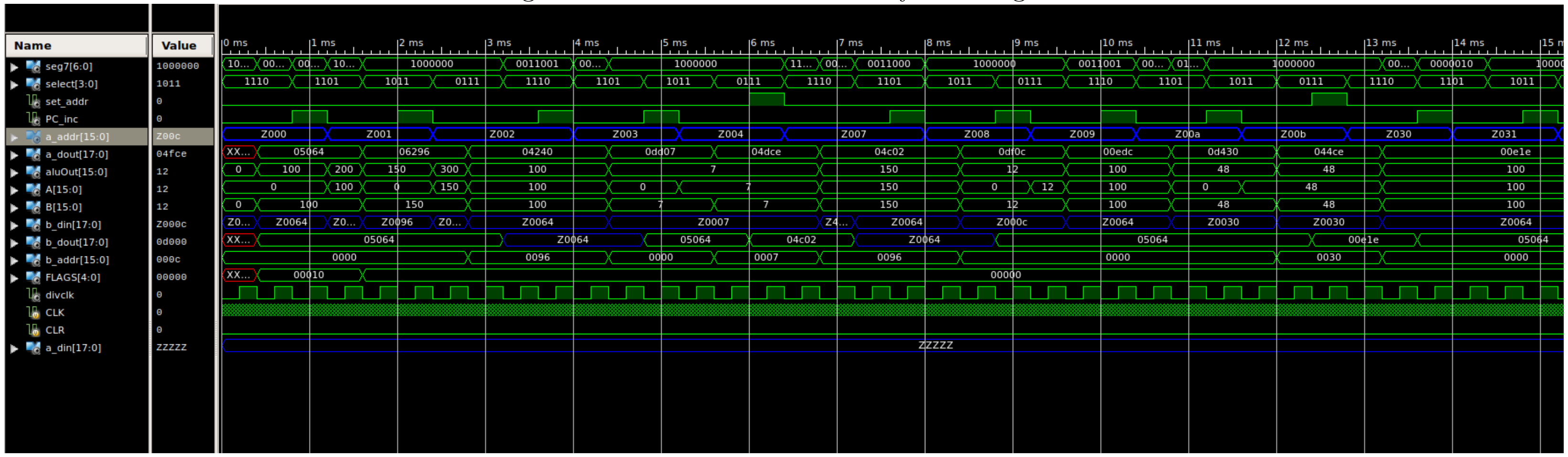


Figure 3: Performs a series of arithmetic and logical operations

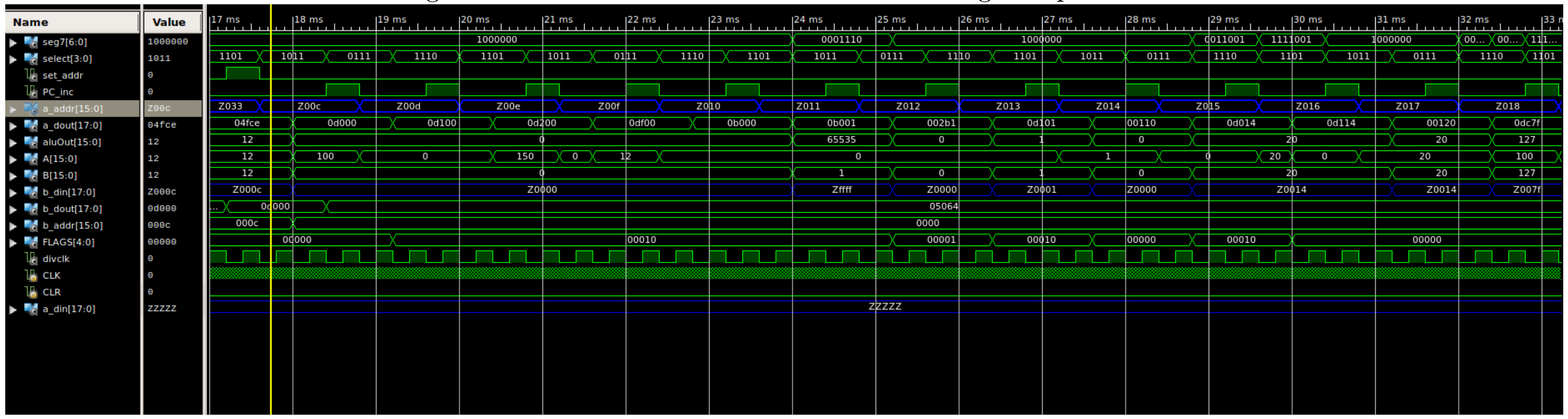
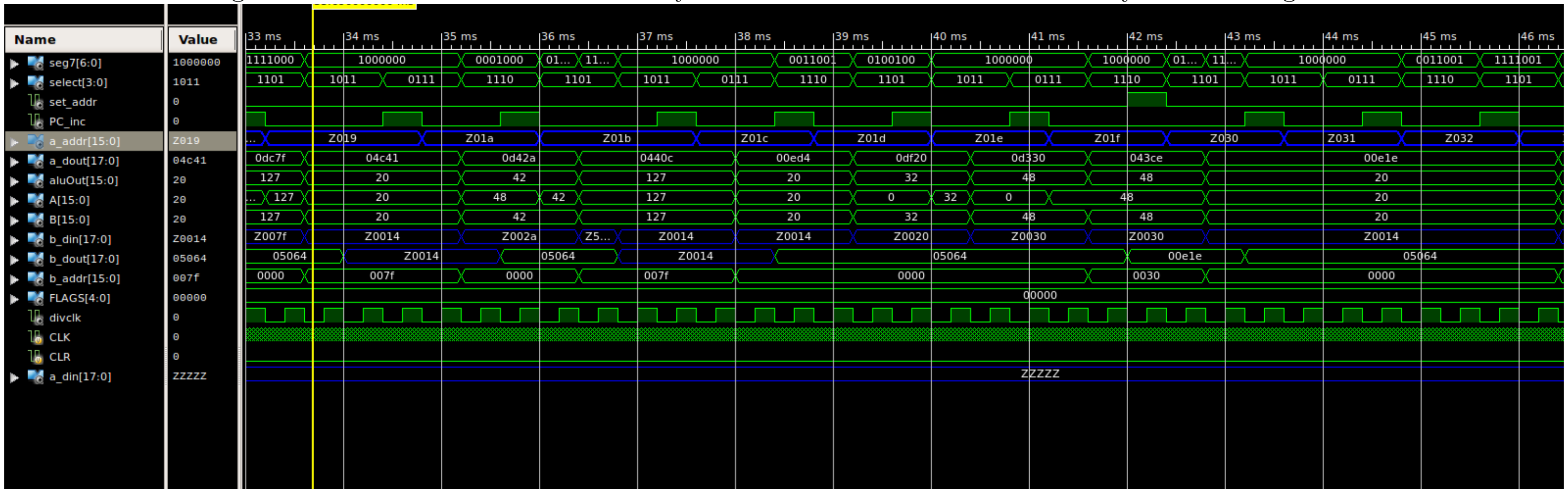
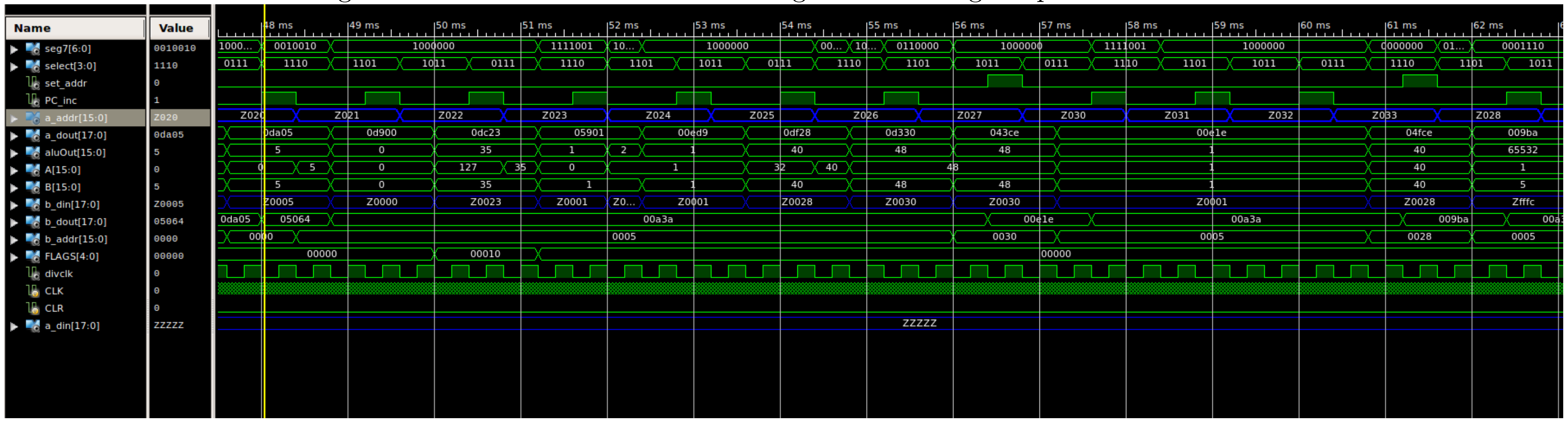


Figure 4: Stores the result into memory and reloads the result from memory into the RegFile.



Name	Value
seg7[6:0]	1000000
select[3:0]	1011
set_addr	0
PC_inc	0
a_addr[15:0]	Z019
a_dout[17:0]	04c41
aluOut[15:0]	20
A[15:0]	20
B[15:0]	20
b_din[17:0]	Z0014
b_dout[17:0]	05064
b_addr[15:0]	007f
FLAGS[4:0]	00000
divclk	0
CLK	0
CLR	0
a_din[17:0]	ZZZZZ

Figure 5: Performs arithmetic to set flags and uses flags to perform branches.



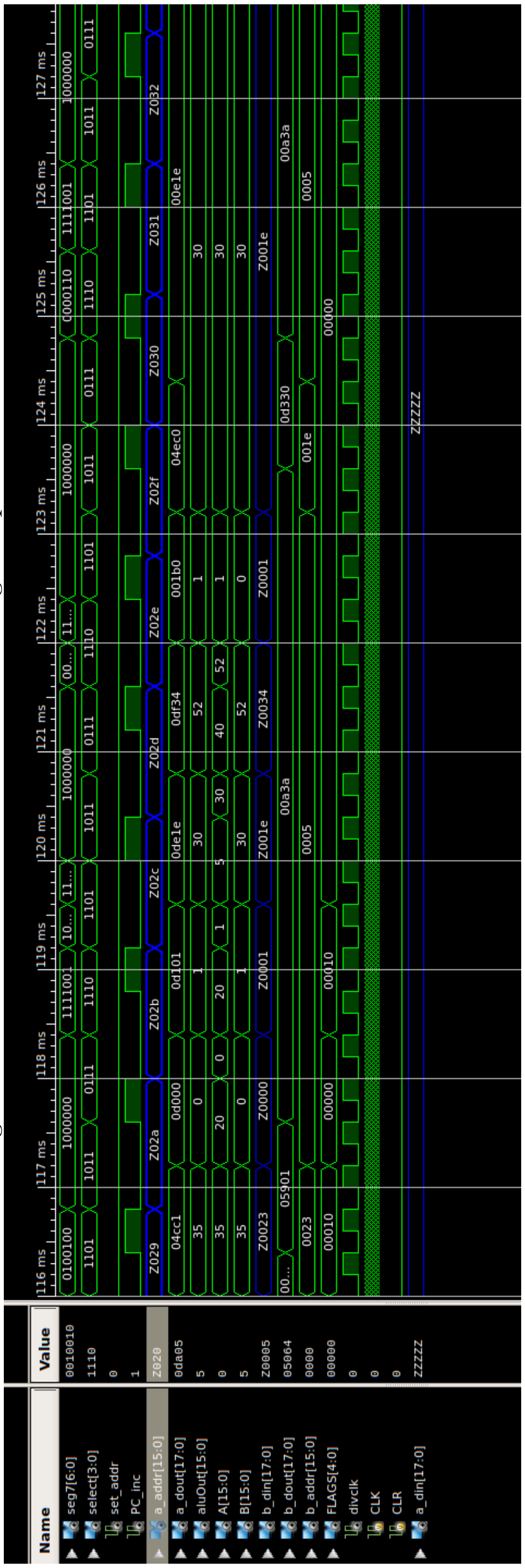


Figure 6: Perform a series of arithmetic and logical operations

Appendix

A Synthesis Results

A.1 Memory

```
Synthesizing Unit <memory>.
    DATA = 18
    ADDR = 14
    Found 16384x18-bit dual-port RAM <Mram_mem> for signal <mem>.
    Found 18-bit register for signal <b_dout>.
    Found 18-bit register for signal <a_dout>.
    Summary:
inferred   1 RAM(s).
inferred  36 D-type flip-flop(s).
inferred   1 Multiplexer(s).
```

A.2 Instruction FSM

```
Synthesizing Unit <instruction FSM>.
    Found 5-bit register for signal <FLAGS>.
    Found 3-bit register for signal <PS>.
    Found finite state machine <FSM 0> for signal <PS>.
    ------------------------------------------------------------
    | States             | 8                     |
    | Transitions        | 13                    |
    | Inputs             | 4                     |
    | Outputs            | 7                     |
    | Clock              | CLK (rising edge)     |
    | Reset              | CLR (positive)        |
    | Reset type         | synchronous           |
    | Reset State        | 000                   |
    | Encoding           | auto                  |
    | Implementation     | LUT                   |
    ------------------------------------------------------------
    Summary:
    inferred   5 D-type flip-flop(s).
    inferred   7 Multiplexer(s).
    inferred   1 Finite State Machine(s).
```

A.3 Decoder

```
Synthesizing Unit <decoder>.
    Found 8-bit 15-to-1 multiplexer for signal <inst[15]_PWR_9_o_wide_mux_37_OUT>
    Found 4-bit 4-to-1 multiplexer for signal <_n0144> created at line 304.
    Summary:
    inferred  50 Multiplexer(s).
```

```
Unit <decoder> synthesized.
```

A.4 Program Counter

```
Synthesizing Unit <program_counter>.
    Found 14-bit register for signal <out_addr>.
    Found 14-bit adder for signal <out_addr[13]_GND_2_o_add_1_OUT> created at lin
    Summary:
    inferred   1 Adder/Subtractor(s).
    inferred  14 D-type flip-flop(s).
    inferred   1 Multiplexer(s).
```

B Memory

B.1 Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:16:09 10/31/2013
// Design Name:
// Module Name:    memory_testing_block
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module memory#(
    parameter DATA = 18,
    parameter ADDR = 14
) (
    // Port A
    input   wire                a_clk,
    input   wire                a_wr,
    input   wire    [ADDR-1:0]  a_addr,
    input   wire    [DATA-1:0]  a_din,
    output  reg     [DATA-1:0]  a_dout,

    // Port B
    input   wire                b_clk,
    input   wire                b_wr,
    input   wire    [ADDR-1:0]  b_addr,
    input   wire    [DATA-1:0]  b_din,
    output  reg     [DATA-1:0]  b_dout
);

// Shared memory
reg [DATA-1:0] mem [(2**ADDR)-1:0];
initial begin
$readmemh("cpu_inst.mem", mem);
end
// Port A
always @(posedge a_clk) begin
    a_dout      <= mem[a_addr];
    if(a_wr) begin
        a_dout      <= a_din;
        mem[a_addr] <= a_din;
    end
end

// Port B
always @(posedge b_clk) begin
    b_dout      <= mem[b_addr];
    if(b_wr) begin
        b_dout      <= b_din;
        mem[b_addr] <= b_din;
    end
end
```

```
endmodule
```

B.2 FSM

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:52:36 10/10/2013
// Design Name:
// Module Name:    memory_FSM
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module memory_FSM(
    input clk,
    input clr,
    input [4:0] ext_input,
 input button,
    output [6:0] seg7,
    output [3:0] select,
 output [14:0] addr0,
   output [14:0] addr1,
 output [15:0] data0,
 output [15:0] data1
    );

// wire [15:0]addr0,addr1;
wire w0,w1;
//wire [15:0]data0,data1;
wire [17:0] out0, out1;
wire [15:0] display;
// input clk,
// input clr,
// input [5:0] ext_input,
// output reg[14:0] addr0, addr1,
// output reg w0, w1,
// output reg[15:0] data0, data1
memory_FSM_fib fsm(clk,clr,ext_input,addr0,addr1,w0,w1,data0,data1);


// input CLK, CLR, w0, w1,
// input [14:0] addr0, addr1,
// input [17:0] data0, data1,
// output reg [17:0] out0, out1
/*
 // Port A
    input   wire                a_clk,
    input   wire                a_wr,
    input   wire    [ADDR-1:0]  a_addr,
    input   wire    [DATA-1:0]  a_din,
    output  reg     [DATA-1:0]  a_dout,

    // Port B
    input   wire                b_clk,
    input   wire                b_wr,
    input   wire    [ADDR-1:0]  b_addr,
    input   wire    [DATA-1:0]  b_din,
    output  reg     [DATA-1:0]  b_dout*/
memory mem_block(clk,w0,addr0,data0,out0,clk,w1,addr1,data1,out1);

// input [15:0] a0,
// input [15:0] a1,
// input sel,
// output reg [15:0] b
mux2_to_1_16bit muxX(out0,out1,button,display);

SSD_decoder decoder(clk, clr, display, seg7, select);


endmodule
```

B.3 Memory Load test

```
`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   09:35:14 11/02/2013
// Design Name:   memory
// Module Name:   /home/dan/Documents/xilinx_projects/CPU/memory_test.v
// Project Name:  CPU
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: memory
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module memory_test_file_readin;

// Inputs
reg a_clk;
reg a_wr;
reg [13:0] a_addr;
reg [17:0] a_din;
reg b_clk;
reg b_wr;
reg [13:0] b_addr;
reg [17:0] b_din;

// Outputs
wire [17:0] a_dout;
wire [17:0] b_dout;

// Instantiate the Unit Under Test (UUT)
memory uut (
.a_clk(a_clk),
.a_wr(a_wr),
.a_addr(a_addr),
.a_din(a_din),
.a_dout(a_dout),
.b_clk(b_clk),
.b_wr(b_wr),
.b_addr(b_addr),
.b_din(b_din),
.b_dout(b_dout)
);
integer i;
initial begin
// Initialize Inputs
a_clk = 0;
a_wr = 0;
a_addr = 0;
a_din = 0;
b_clk = 0;
b_wr = 0;
b_addr = 0;
b_din = 0;


// Wait 100 ns for global reset to finish
#100;

//Check that the reading was correct
for(i=0; i< 10; i=i+1)
begin
a_addr = i;
#2;

$display("A_ADDR: %0d, A_DOUT: %16b", a_addr, a_dout);


end
$display("done");

end

always begin
#1 a_clk = ~a_clk;
b_clk = ~b_clk;
end

endmodule
```

B.4 Memory test

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   09:35:14 11/02/2013
// Design Name:   memory
// Module Name:   /home/dan/Documents/xilinx_projects/CPU/memory_test.v
// Project Name:  CPU
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: memory
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module memory_test;

// Inputs
reg a_clk;
reg a_wr;
reg [13:0] a_addr;
reg [17:0] a_din;
reg b_clk;
reg b_wr;
reg [13:0] b_addr;
reg [17:0] b_din;

// Outputs
wire [17:0] a_dout;
wire [17:0] b_dout;

// Instantiate the Unit Under Test (UUT)
memory uut (
.a_clk(a_clk),
.a_wr(a_wr),
.a_addr(a_addr),
.a_din(a_din),
.a_dout(a_dout),
.b_clk(b_clk),
.b_wr(b_wr),
.b_addr(b_addr),
.b_din(b_din),
.b_dout(b_dout)
);
integer i;
initial begin
// Initialize Inputs
a_clk = 0;
a_wr = 0;
a_addr = 0;
a_din = 0;
b_clk = 0;
b_wr = 0;
b_addr = 0;
b_din = 0;

// Wait 100 ns for global reset to finish
#100;

a_wr = 1;
b_wr = 1;
// Add stimulus here
a_addr = 0;
a_din = 18'b000000000000000010;

for(i=0; i< (2 ** 14) /2; i=i+1)
begin
a_addr = i;
a_din = i;
b_addr = i + (2 ** 14)/2;
b_din =  i + (2 ** 14)/2;
#2;
end
```

```
a_wr = 0;
b_wr = 0;
//Check that the reading was correct
for(i=0; i< (2 ** 14) / 2; i=i+1)
begin
#2;
a_addr = i;
b_addr = i + (2 ** 14)/2;
#2;
if(i != a_dout)
begin
$display("A_ADDR: %0d, A_DOUT: %0d", a_addr, a_dout);
end
if(i + (2 ** 14)/2 != b_dout)
begin
$display("B_ADDR: %0d, B_DOUT: %0d",b_addr, b_dout);
end
end
$display("done");

end

always begin
#1 a_clk = ~a_clk;
b_clk = ~b_clk;
end

endmodule
```

C Controller (Decoder and FSM)

C.1 Instruction FSM Module

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
////////////////////////////////////////////////////////////////////////////////
// A note about testing the instruction_FSM.  The FLAGS from the previous
// operation are not saved in this module until the fetch state completes
// This means that the life-cycle of an instruction begins in the decode
// state then through whatever next states there are (alu, load1, stor1,
// jump, etc. and ends at fetch.)  For testing, the FLAGS must be set for
// 2 clock cycles before they can change.  Lifecycle for a non-memory
// instruction is 3 clock cycles.  Lifecycles for a memory instruction is
// 4 clock cycles.
module instruction_FSM ( CLK, CLR, inst, _FLAGS, FLAGS, PC_inc, JAddrSelect, loadReg);
input CLK, CLR;
input [17:0] inst;
input [4:0] _FLAGS; // Input flags from the ALU
output [4:0] FLAGS; // stored output flags for use with later Instructions
output reg PC_inc, JAddrSelect, loadReg;

parameter MEM = 4'b0100;
parameter LOAD_1 = 4'b0000;
parameter STOR_1 = 4'b0100;

// MEM is the opcode, JCOND and SCOND are the secondary codes, JUC through BLE are stored in bits [3:0]
// JCOND will be absolute memory jumps
// SCOND will be relative memory jumps
parameter JCOND = 4'b1100; // JCOND uses unsigned comparison for BEQ through BLT
//parameter SCOND = 4'b0100; // SCOND is the same as the MEM and is a signed comparison for BEQ through BLE
parameter JUC = 4'b1110; // JUC jumps directly
parameter BEQ = 4'b0000;
parameter BNEQ = 4'b0001;
parameter CMP_1 = 4'b0000;
parameter CMP_2 = 4'b1011;
parameter CMPI = 4'b1011;
parameter CMPUI = 4'b1110; // Replaces MULI

parameter fetch = 3'b000; // fetch is the initial state, set up the address for Instruction Memory
parameter decode = 3'b001; // decode is the state after getting the data_out from the Instruction Memory
parameter alu = 3'b010; // ALU is a state after decode which says go straight to fetch;
parameter stor1 = 3'b011; // stor1 is a state after decode which says wait several cycles
parameter stor2 = 3'b100; // waits for the output, then sets up the instructions to push to the register
parameter load1 = 3'b101; // load1 is a state after decode which says wait several cycles
parameter load2 = 3'b110; // waits for the output of memory to say that the data was loaded
parameter jump = 3'b111; // set up the program counter with the new address

reg [2:0] NS;
reg [2:0] PS;
reg [4:0] FLAGS;

// Present State Logic
```

```
always @ (posedge CLK) begin
if (CLR) begin PS <= fetch; FLAGS <= _FLAGS; end
else begin
PS <= NS;
if (PS == fetch) FLAGS <= _FLAGS;
else FLAGS <= FLAGS;
end
end

// Next State Logic
always @ (*) begin
case (PS)
fetch: begin NS <= decode; end
decode: begin
case (inst[15:12])
MEM: begin
case (inst[7:4])
LOAD_1: begin
NS <= load1;
end
STOR_1: begin
NS <= stor1;
end
JCOND: begin
NS <= jump;
end
default: begin NS <= fetch; end
endcase
end
default: begin
NS <= alu;
end
endcase
end
alu: begin
NS <= fetch;
end
load1: begin
NS <= load2;
end
load2: begin
NS <= fetch;
end
stor1: begin
NS <= stor2;
end
stor2: begin
NS <= fetch;
end
jump: begin
NS <= fetch;
end
default: begin
NS <= fetch;
end
endcase
end

// Output Logic
always @ (*) begin
PC_inc <= 1'b0;
JAddrSelect <= 1'b0;
loadReg <= 1'b0;
case (PS)
fetch: begin end // Probably need to set the loadReg[4] to 0 so we don't modify the Registers
decode: begin end // Probably need to set the loadReg[4] to 0 so we don't modify the Registers
alu: begin PC_inc <= 1'b1; loadReg <= 1'b1;
case (inst[15:12])
CMP_1: begin
if (inst[7:4] == CMP_2) loadReg <= 1'b0;
end
CMPI: loadReg <= 1'b0;
CMPUI: loadReg <= 1'b0;
default: loadReg <= 1'b1;
endcase
end
load1: begin end // Probably need to set the loadReg[4] to 0 so we don't modify the Registers
load2: begin PC_inc <= 1'b1; loadReg <= 1'b1; end
stor1: begin end // Probably need to set the loadReg[4] to 0 so we don't modify the Registers
stor2: begin PC_inc <= 1'b1; loadReg <= 1'b0; end
jump: begin
//JAddrSelect <= 1'b1;
// C, L, F, Z, N
case (inst[3:0])
JUC: begin JAddrSelect <= 1'b1; end
BEQ: begin
// Remember that the Registers will not be updated until the next clock cycle,
// so the NOP instruction that gets sent to the ALU will not update the flags
// until the next posedge CLK
// Since this is a BEQ, then if the Z Flag is 1, the arguments
// to the CMP operation were equal
if (FLAGS[1] == 1'b1) begin JAddrSelect <= 1'b1; loadReg <= 1'b0; end
else PC_inc <= 1'b1;
end
BNEQ: begin
// Since this is a BNEQ, then if the Z Flag is 0, the arguments
// to the CMP operation were not equal
if (FLAGS[1] == 1'b0) begin JAddrSelect <= 1'b1; loadReg <= 1'b0; end
else PC_inc <= 1'b1;
end
// other functionality can be added easily here to check the other flags for the GE, GT, LE, and LT
// Branches.
default: begin PC_inc <= 1'b1; JAddrSelect <= 1'b0; loadReg <= 1'b0; end
endcase
end
default: begin
PC_inc <= 1'b1;
end
endcase
end
endmodule
```

C.2 Controller Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
// Module Name:    controller_integrated
//
//////////////////////////////////////////////////////////////////////////////////
module controller_integrated(
input CLK, CLR,
input [17:0] inst, external_din,
output w1,  // w1 is writeToMemory
output [15:0] addr1, // A[14:0] is addr1 for the memory module
output [15:0] data1,
output [4:0] stored_flags,
output [15:0] A, B, aluOut, // for debugging
//output [3:0] readRegA, loadReg, // for debugging
output PC_inc, JAddrSelect

    );

wire [7:0] OP;
wire [15:0] Imm;
wire [3:0] readRegA, readRegB, loadReg, memAddr; // loadReg; // readRegA,
wire selectImm, selectResult;
wire [15:0] r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, r8out, r9out, r10out, r11out, r12out, r13out, r14out, r15out;
reg [15:0] enWrite, reset;
wire [15:0] RegB;
wire [4:0] FLAGS;

instruction_FSM controller( CLK, CLR, inst, FLAGS, stored_flags, PC_inc, JAddrSelect, LR);

ALU _alu(stored_flags[4], A, B, OP, aluOut, FLAGS[4], FLAGS[3], FLAGS[2], FLAGS[1], FLAGS[0]);

decoder _decoder(inst, OP, Imm, selectImm, selectResult, w1, memAddr, readRegA, readRegB, loadReg);

RegFile _regfile(CLK, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1, data1,
 data1, enWrite, reset, r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, r8out, r9out, r10out, r11out, r12out, r13out, r14out, r15out);

mux16_to_1_16bit MemSelect(r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, r8out,
r9out, r10out, r11out, r12out, r13out, r14out, r15out, memAddr, addr1);

mux16_to_1_16bit ASelect(r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, r8out,
r9out, r10out, r11out, r12out, r13out, r14out, r15out, readRegA, A);

mux16_to_1_16bit BSelect(r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, r8out,
r9out, r10out, r11out, r12out, r13out, r14out, r15out, readRegB, RegB);

mux2_to_1_16bit ImmMux(RegB, Imm, selectImm, B);
mux2_to_1_16bit ResultMux(aluOut, external_din[15:0], selectResult, data1);

always @ (*) begin
if (CLR == 1'b1) begin reset <= 16'b1111111111111111; enWrite <= 16'b0; end
else begin
reset <= 16'b0000000000000000;
if (LR == 0) enWrite <= 16'b0000000000000000;
// loadReg is a 5-bit address/enable vector.  The lower four
// bits indicate the address of the register to be enabled, but the
// MSB indicates whether writing is enabled.  This MSB is active-Low,
// meaning that when it is low, writing is enabled for the register
// addressed by the lower 4 bits.  When it is high, writing is disabled.
// This decision was for ease of reading and implementation.
else
case(loadReg) // decodes which register to enable
0: begin enWrite <=  16'b0000000000000001; reset <= 16'b0; end
1: begin enWrite <=  16'b0000000000000010; reset <= 16'b0; end
2: begin enWrite <=  16'b0000000000000100; reset <= 16'b0; end
3: begin enWrite <=  16'b0000000000001000; reset <= 16'b0; end
4: begin enWrite <=  16'b0000000000010000; reset <= 16'b0; end
5: begin enWrite <=  16'b0000000000100000; reset <= 16'b0; end
6: begin enWrite <=  16'b0000000001000000; reset <= 16'b0; end
7: begin enWrite <=  16'b0000000010000000; reset <= 16'b0; end
8: begin enWrite <=  16'b0000000100000000; reset <= 16'b0; end
9: begin enWrite <=  16'b0000001000000000; reset <= 16'b0; end
10: begin enWrite <= 16'b0000010000000000; reset <= 16'b0; end
11: begin enWrite <= 16'b0000100000000000; reset <= 16'b0; end
12: begin enWrite <= 16'b0001000000000000; reset <= 16'b0; end
13: begin enWrite <= 16'b0010000000000000; reset <= 16'b0; end
14: begin enWrite <= 16'b0100000000000000; reset <= 16'b0; end
15: begin enWrite <= 16'b1000000000000000; reset <= 16'b0; end
default: begin enWrite <= 16'b0; reset <= 16'b0; end
endcase
end
end
endmodule
```

C.3 Decoder Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
// Module Name:    controller
// Description: controller turns the 8-bit Imm value into a 16-bit immediate value
//      Signed instructions will have a sign-extended immediate value, Unsigned
//      instructions will have eight 0's added to Imm[15:8].  LUI will shift the
//   8-bit Imm value up to Imm[15:8] and add eight 0's to fill Imm[7:0].
//
//////////////////////////////////////////////////////////////////////////////////
module decoder(
    input [17:0] inst,
    output reg [7:0] op,
 output reg [15:0] Imm, // Imm needs to be sign-extended or 0-extended when applicable
 output reg selectImm, selectResult, w1, //e1,
 output reg [3:0] memAddr,
 output reg [3:0] readRegA,
 output reg [3:0] readRegB,
 output reg [3:0] loadReg
    );

parameter RTYPE = 4'b0000;
//parameter ADD_0 = 4'b0000;
parameter ADD_1 = 4'b0101;
parameter ADDI = 4'b0101;
//parameter ADDU_0 = 4'b0000;
parameter ADDU_1 = 4'b0110;
parameter ADDUI = 8'b0110;
//parameter ADDC_0 = 4'b0000;
parameter ADDC_1 = 4'b0111;
//parameter ADDCU_0 = 4'b0000;
parameter ADDCU_1 = 4'b0100;
parameter ADDCUI = 4'b1010; // Replaces SUBCI
parameter ADDCI = 4'b0111;
//parameter SUB_0 = 4'b0000;
parameter SUB_1 = 4'b1001;
parameter SUBI = 4'b1001;
//parameter CMP_0 = 4'b0000;
parameter CMP_1 = 4'b1011;
parameter CMPI = 4'b1011;
parameter CMPUI = 4'b1110; // Replaces MULI
//parameter AND_0 = 4'b0000;
parameter AND_1 = 4'b0001;
//parameter OR_0 = 4'b0000;
parameter OR_1 = 4'b0010;
//parameter XOR_0 = 4'b0000;
parameter XOR_1 = 4'b0011;
//parameter NOT_0 = 4'b0000;
parameter NOT_1 = 4'b1111;
parameter SHIFT = 4'b1000;
//parameter LSH_0 = 4'b1000;
parameter LSH_1 = 4'b0100;
//parameter LSHI_0 = 4'b1000;
parameter LSHI_1 = 4'b0000; // Imm is unsigned
//parameter RSH_0 = 4'b1000;
parameter RSH_1 = 4'b1100;
//parameter RSHI_0 = 4'b1000;
parameter RSHI_1 = 4'b0001; // Imm is unsigned
//parameter ALSH_0 = 4'b1000;
```

```
parameter ALSH_1 = 4'b0101; // Interprets RSrc as Unsigned
//parameter ARSH_0 = 4'b1000;
parameter ARSH_1 = 4'b1101; // Interprets RSrc as Unsigned

parameter MEM = 4'b0100;
parameter LOAD_1 = 4'b0000;
parameter STOR_1 = 4'b0100;
parameter LUI = 4'b1111;
// MOV_0 = 4'b0000;
parameter MOV_1 = 4'b1101;
parameter MOVI = 4'b1101;

//parameter LOAD = 2'b10; // This is a read instruction and reads the value in memory[RAddr] and stores in RDest
//parameter STOR = 2'b11; // This is a write instruction and writes the value in RDest to memory[RAddr]

// MEM is the opcode, JCOND and SCOND are the secondary codes, JUC through BLE are stored in bits [3:0]
// JCOND will be absolute memory jumps
// SCOND will be relative memory jumps
parameter JCOND = 4'b1100; // JCOND uses unsigned comparison for BEQ through BLT
//parameter SCOND = 4'b0100; // SCOND is the same as the MEM and is a signed comparison for BEQ through BLE
parameter JUC = 4'b1110; // JUC jumps directly
parameter BEQ = 4'b0000;
parameter BNEQ = 4'b0001;
//parameter BGT = 4'b0110;
//parameter BLT = 4'b0111;
//parameter BGE = 4'b1101;
//parameter BLE = 4'b1100;


always @ (*) begin
w1 <= 1'b0;
selectResult <= 1'b0;
selectImm <= 1'b0;
Imm <= 16'b0000000000000000;
//e1 <= 1'b0;
op <= 8'b00000000;
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
memAddr <= 4'b1010;
if (inst[17:16] == 2'b00) begin
case (inst[15:12])
RTYPE: begin
case (inst[7:4])
ADD_1: begin
op <= {RTYPE, ADD_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
ADDU_1: begin
op <= {RTYPE, ADDU_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
ADDC_1: begin
op <= {RTYPE, ADDC_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
ADDCU_1: begin
op <= {RTYPE, ADDCU_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
SUB_1: begin
op <= {RTYPE, SUB_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
CMP_1: begin
op <= {RTYPE, CMP_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
AND_1: begin
op <= {RTYPE, AND_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
OR_1: begin
op <= {RTYPE, OR_1};
```

```
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
NOT_1: begin
op <= {RTYPE, NOT_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
XOR_1: begin
op <= {RTYPE, XOR_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
MOV_1: begin
op <= {RTYPE, MOV_1};
readRegA <= inst[3:0];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
ADDI: begin
op <= {ADDI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7:0]};
end
ADDUI: begin
op <= {ADDUI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {8'b00000000, inst[7:0]};
end
ADDCUI: begin
op <= {ADDCUI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {8'b00000000, inst[7:0]};
end
ADDCI: begin
op <= {ADDCI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7:0]};
end
SUBI: begin
op <= {SUBI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7:0]};
end
CMPI: begin
op <= {CMPI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7], inst[7:0]};
end
CMPUI: begin
op <= {CMPUI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {8'b00000000, inst[7:0]};
end
MOVI: begin
op <= {MOVI, inst[7:4]};
```

```
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {8'b00000000, inst[7:0]};
end
LUI: begin
op <= {LUI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {inst[7:0], 8'b00000000};
end
SHIFT: begin
case (inst[7:4])
LSH_1: begin
op <= {SHIFT, LSH_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
LSHI_1: begin
op <= {SHIFT, LSHI_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {12'b000000000000, inst[3:0]};
end
RSH_1: begin
op <= {SHIFT, RSH_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
RSHI_1: begin
op <= {SHIFT, RSHI_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= {12'b000000000000, inst[3:0]};
end
ALSH_1: begin
op <= {SHIFT, ALSH_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
ARSH_1: begin
op <= {SHIFT, ARSH_1};
readRegA <= inst[11:8];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
MEM: begin
// LOAD and STOR take two cycles to complete, so duplicate the LOAD instruction, and use
// a NOP instruction on the STOR instruction
case (inst[7:4])
LOAD_1: begin
// LOAD takes the value stored in mem[RAddr] and loads it into RDest
// where RAddr = inst[3:0] and RDest = inst[11:8]
op <= {RTYPE, OR_1};
memAddr <= inst[3:0];
readRegA <= inst[3:0];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
// selectResult selects the value out of the memory
selectResult <= 1'b1;
//e1 <= 1'b1;
end
STOR_1: begin
// STOR takes the value stored in RDest and stores it in mem[RAddr]
// where RAddr = inst[11:8] and RDest = inst[3:0]
op <= {RTYPE, OR_1};
memAddr <= inst[11:8];
readRegA <= inst[3:0];
readRegB <= inst[3:0];
loadReg <= inst[11:8];
```

```
w1 <= 1'b1;
//e1 <= 1'b1;
end
JCOND: begin
// These should all essentially be the same
memAddr <= inst[11:8];
case (inst[3:0])
JUC: begin
// get the address out of inst[11:8] and throw it up to the PC counter
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
BEQ: begin
// Check the flags after a CMP operation
// get the address out of inst[11:8] and throw it up the PC counter if
// Z <= 1

op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
BNEQ: begin
// Check the flags after a CMP operation
// get the address out of inst[11:8] and throw it up to the PC counter if
// Z <= 0;

op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
// BCOND: begin
//
// end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
else begin
case (inst[17:16])
2'b11: begin
// might be useful to always load the result into the same register so we can use
// all 16 bits for the LI.
op <= {LUI, inst[7:4]};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
selectImm <= 1'b1;
Imm <= inst[15:0];
end
default: begin
op <= {RTYPE, OR_1};
readRegA <= inst[11:8];
readRegB <= inst[11:8];
loadReg <= inst[11:8];
end
endcase
end
end

endmodule
```

C.4 Controller Test

```
`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   18:33:05 11/03/2013
// Design Name:   controller_integrated
// Module Name:   /home/dan/Documents/xilinx_projects/CPU/controller_integrated_test.v
// Project Name:  CPU
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: controller_integrated
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module controller_integrated_test;

// Inputs
reg CLK;
reg CLR;
reg [17:0] inst;
reg [17:0] external_din;

// Outputs
wire w1;
wire [15:0] addr1;
wire [15:0] data1;
wire [4:0] FLAGS;
wire [15:0] B;
wire [15:0] aluOut;
wire [3:0] readRegA, loadReg;
wire PC_inc;
wire JAddrSelect;

// Instantiate the Unit Under Test (UUT)
controller_integrated uut (
.CLK(CLK),
.CLR(CLR),
.inst(inst),
.external_din(external_din),
.w1(w1),
.addr1(addr1),
.data1(data1),
.FLAGS(FLAGS),
.B(B),
.aluOut(aluOut),
.readRegA(readRegA),
.loadReg(loadReg),
.PC_inc(PC_inc),
.JAddrSelect(JAddrSelect)
);

initial begin
// Initialize Inputs
CLK = 0;
CLR = 0;
inst = 0;
external_din = 18'b0;

CLR = 1;
#3 CLR = 0;
// Wait 100 ns for global reset to finish
#100;

inst = 16'h6aff; //        addui 255, $10
#6;
if(data1 != 16'b0000000011111111)
begin
$display("ERROR1: data1 was %0d; expected 255", data1);
end

inst = 16'hfb01; // lui 1, $11
#6;

inst = 16'h0a2b; // or $10, $11
#5;

inst = 16'h6a00;  // addui 0, $10
#6;
if(addr1 != 16'b0000000111111111)
```

```
begin
$display("ERROR2: Addr1 was %0d; expected 511", addr1);
end
      #6;
inst =  16'hd114; // movi 20, $1
#6;

inst = 16'h0232; //xor $2, $2
#6;
if(data1 != 16'b0)
begin
$display("ERROR3: data1 was %0d; expected 0", data1);
end

inst = 16'h522a; // addi 42, $2
#6;

inst = 16'h0212; // and $2, $2
#6;
if(data1 != 8'h2a) // 42
begin
$display("ERROR3.1: data1 was %0d; expected 42", data1);
end

inst =  16'hde63; // movi 99, $14
#6;

inst = 16'h4142; // stor $1, $2
#6;
if(addr1 != 16'b0000000000010100)
begin
$display("ERROR5: Addr1 was %0d; expected 20", addr1);
end

inst = 16'h0212; // and $2, $2
#6;
if(data1 != 8'h2a) // 42
begin
$display("ERROR5.1: data1 was %0d; expected 42", data1);
end

inst =  16'hdf64; // movi 100, $15
#6;


inst =  16'hd114; // movi 20, $1
#6;
inst = 16'h4201;  //load $2, $1
#6;
if(addr1 != 16'b0000000000010100)
begin
$display("ERROR6: addr1 was %0d; expected 20", addr1);
end

inst = 16'h0212; //xor $2, $2
// Add stimulus here
#6;
$finish;
end

always
#1 CLK = ~CLK;

endmodule
```

C.5 Instruction FSM Test

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
////////////////////////////////////////////////////////////////////////////////

module instruction_FSM_test;

// Inputs
reg CLK;
reg CLR;
reg [15:0] inst;
reg [4:0] _FLAGS;

// Outputs
wire PC_inc;
wire JAddrSelect;
wire loadReg;
wire [4:0] FLAGS;

// Instantiate the Unit Under Test (UUT)
instruction_FSM uut (
.CLK(CLK),
.CLR(CLR),
.inst(inst),
._FLAGS(_FLAGS),
.PC_inc(PC_inc),
.JAddrSelect(JAddrSelect),
.loadReg(loadReg)
);

initial begin
// Initialize Inputs
CLK = 0;
CLR = 0;
inst = 0;
_FLAGS = 0;

// Wait 100 ns for global reset to finish
#8;
// Add stimulus here
// ADD r1 r0
inst = 16'b0000000101010000;
#6;
// ADDC r2 r1
inst = 16'b0000001001110001;
#6;
// CMP r3 r15
inst = 16'b0000001110111111;
#6;
// RSH r1 r2
inst = 16'b1000000111000010;
#6;
// LUI r1 16
inst = 16'b1111000100010000;
#6;
// JUC r5
inst = 16'b0100010111001110;
#6;
// BEQ r1 // Should Jump after 3 cycles (6ns)
_FLAGS = 5'b00010;
inst = 16'b0100000111000000;
#4;
_FLAGS = 5'b00000;
#2;
// BEQ r1 // Should NOT Jump after 3 cycles (6ns)
_FLAGS = 5'b00000;
inst = 16'b0100000111000000;
#4;
_FLAGS = 5'b00010;
#2;
// BNEQ r14 // Should Jump after 3 cycles (6ns)
_FLAGS = 5'b00000;
inst = 16'b0100111011000001;
#4;
_FLAGS = 5'b00010;
#2;
// BNEQ r14 // Should NOT Jump after 3 cycles (6ns)
_FLAGS = 5'b00010;
inst = 16'b0100111011000001;
#4;
_FLAGS = 5'b00000;
#2;
// LOAD r15 r4
inst = 16'b0100111100000100;
#8;
// STOR r15 r4
inst = 16'b0100111101000100;
#8;
$finish;
end

always begin CLK = ~CLK; #1; end

endmodule
```

C.6 Decoder Test

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
```

```
//
////////////////////////////////////////////////////////////////////////////////

module decoder_tb;

// Inputs
reg [17:0] inst;

// Outputs
wire [7:0] op;
wire [15:0] Imm;
wire selectImm;
wire selectResult;
wire w1;
wire [3:0] readRegA;
wire [3:0] readRegB;
wire [3:0] loadReg;

// Instantiate the Unit Under Test (UUT)
decoder uut (
.inst(inst),
.op(op),
.Imm(Imm),
.selectImm(selectImm),
.selectResult(selectResult),
.w1(w1),
.readRegA(readRegA),
.readRegB(readRegB),
.loadReg(loadReg)
);

initial begin
// Initialize Inputs
inst = 0;

// Wait 100 ns for global reset to finish
#100;

// Add st;imulus here
inst = 16'b001101000100010100;
#10;
inst = 16'b000110001111001000;
#10;
inst = 16'b000100000101000011;
#10;
inst = 16'b000000010000110100;
#10;
inst = 16'b000100001000000001;
#10;
inst = 16'b000110001000000001;
#10;
inst = 16'b001001001000001010;
#10;
$finish;
end

endmodule
```

D Program Counter

D.1 Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:48:52 10/29/2013
// Design Name:
// Module Name:    program_counter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module program_counter(
    input CLK,
    input CLR,
 input set_addr,
 input [13:0] in_addr,
 input PC_inc,
    output reg[13:0] out_addr
    );

 always @(posedge CLK) begin
if (CLR)
out_addr <= 14'b0;
else begin
if (set_addr)
out_addr <= in_addr;
else
out_addr <= out_addr + PC_inc;
end
 end


endmodule
```

D.2 Test

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   11:04:03 10/29/2013
// Design Name:   program_counter
// Module Name:   /home/dan/Documents/xilinx_projects/CPU/program_counter_test.v
// Project Name:  CPU
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: program_counter
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module program_counter_test;

// Inputs
reg CLK;
reg CLR;
reg set_addr;
reg [15:0] in_addr;

// Outputs
wire [15:0] out_addr;

// Instantiate the Unit Under Test (UUT)
program_counter uut (
.CLK(CLK),
.CLR(CLR),
.set_addr(set_addr),
.in_addr(in_addr),
.out_addr(out_addr)
);

initial begin
// Initialize Inputs
CLK = 0;
CLR = 0;
set_addr = 0;
in_addr = 0;

// Wait 100 ns for global reset to finish
#100;
CLR = 1;
#3 CLR = 0;
#20;
in_addr = 16'b0000000000000100;
set_addr= 1;
#2 set_addr =0;




// Add stimulus here
```

```
end

always
#1 CLK = ~CLK;

endmodule
```

E CPU

E.1 Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:09:46 10/29/2013
// Design Name:
// Module Name:    CPU
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module CPU(
    input CLK,
    input CLR,
 input button,
    output [6:0] seg7,
    output [3:0] select,
 output PC_inc, set_addr, divclk,
 output [15:0] A, B, aluOut, // for debugging
 output [17:0] a_din, a_dout, b_din, b_dout, // for debugging
 output [15:0] a_addr, b_addr, // for debugging
 output [4:0] FLAGS
    );

reg divclk;
reg [16: 0] count = 0;
reg newCLK;

// clk divider
  always@(posedge CLK, posedge CLR) begin
if (CLR)
divclk <= 0;
else begin
if (count == 100000) begin // set to 100000 for every 1 ms
count <= 0;
divclk <= divclk + 1'b1;
end
else begin
count <= count + 1;
end
end
  end

 always @ (*)
begin
case (divclk)
1'b0: begin newCLK = 1; end
1'b1: begin newCLK = 0; end
endcase
end

wire [15:0] Imm;
wire [3:0] readRegA, readRegB, loadReg;
wire selectImm, selectResult;
wire [17:0] /*inst,*/ out1;
wire [14:0] pc_load_addr = 0;
wire [13:0] /* pc_addr,*/ c_addr, d_addr;

// b_addr is the output of the controller_integrated, which comes from RegA input to
// the ALU.
program_counter counter(newCLK,CLR,set_addr, b_addr[13:0], PC_inc, a_addr[13:0]);

memory asm_RAM (newCLK, 1'b0, a_addr, a_din, a_dout, newCLK, b_wr, b_addr, b_din, b_dout);
//memory game_RAM (CLK, 1'b0, c_addr, c_din, c_dout, CLK, 1'b0, d_addr, d_din, d_dout);
```

```
/*module controller_integrated(
input CLK, CLR,
input [17:0] inst, external_din,
output w1,  // w1 is writeToMemory
output [15:0] addr1, // A[14:0] is addr1 for the memory module
output [15:0] data1,
output [4:0] FLAGS,
output [15:0] A, B, aluOut, // for debugging
//output [3:0] readRegA, loadReg, // for debugging
output PC_inc, JAddrSelect);*/
controller_integrated controller(newCLK,CLR,a_dout,b_dout,b_wr,/*b_addr,*/b_addr, b_din[15:0],FLAGS, A,B,aluOut, PC_inc, set_addr);

//mux2_to_1_16bit muxX(aluOut,a_addr,button,display);
SSD_decoder decoder(newCLK, CLR, aluOut, seg7, select);

endmodule
```

E.2 Test

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
////////////////////////////////////////////////////////////////////////////////

module CPU_test;

/*module CPU(
    input CLK,
    input CLR,
 output PC_inc, set_addr,
 output [15:0] A, B, aluOut, // for debugging
 output [17:0] a_din, a_dout, b_din, b_dout, // for debugging
 output [15:0] a_addr, b_addr, // for debugging
 output [4:0] FLAGS
    );*/

// Inputs
reg CLK;
reg CLR;
reg button;

// Outputs
wire [6:0] seg7;
   wire [3:0] select;
wire PC_inc;
wire set_addr;
wire [15:0] A;
wire [15:0] B;
wire [15:0] aluOut;
wire [17:0] a_din;
wire [17:0] a_dout;
wire [17:0] b_din;
wire [17:0] b_dout;
wire [15:0] a_addr;
wire [15:0] b_addr;
wire [4:0] FLAGS;

// Instantiate the Unit Under Test (UUT)
CPU uut (
.CLK(CLK),
.CLR(CLR),
.button(button),
.seg7(seg7),
.select(select),
.PC_inc(PC_inc),
.set_addr(set_addr),
.divclk(divclk),
.A(A),
.B(B),
.aluOut(aluOut),
.a_din(a_din),
.a_dout(a_dout),
.b_din(b_din),
.b_dout(b_dout),
.a_addr(a_addr),
.b_addr(b_addr),
.FLAGS(FLAGS)
);

initial begin
// Initialize Inputs
CLK = 0;
CLR = 0;

// Wait 100 ns for global reset to finish
```

```
#100;
CLR = 1;
#9 CLR = 0;

// Add stimulus here

end

always
#1 CLK = ~CLK;


endmodule
```

F Assembly Test Code

F.1 Load and Store

```
movi 20, $1
addui 200, $3
stor $1, $3
xor $4, $4
load $2, $1
addui 1, $2
subi 10, $2
and $2, $2
```

F.2 Demo Code

```
# 1. Load data from memory into the regfile
addi 100, $0
addui 150, $2
stor $2_a150, $0_v100 # Value 100 into address 250
movi 7, $13
JUC $13
XOR $10, $10
XOR $10, $10
load $12_v100, $2_a150 # Load from address in $2, and put that value into $1
movi Arithmetic, $r15
mov $12, $a14
movi Display, $4
JUC $4

# 2. Perform a series of arithmetic andlogical operations (set flags)

Arithmetic: movi 0, $0
movi 0, $1
movi 0, $2
movi 0, $15
cmpi 0, $0 # This means it is equal, aka set the zero flag
cmpi 1, $0 # This should be not equal
cmp $1, $2 # This should be equal
movi 1, $1
and $0, $1 # and 0*1, should be zero
movi 20, $0
movi 20, $1
or $0, $1 # value is 20
movi 127, $12

# 3. Store the result in memory
Store: stor $12_Raddr, $1 # Store value 20 in address 127 ($15)
movi 42, $4

# 4. Reload the result from memory into the regfile
load $4, $12_Raddr #reg4 should have value 20
mov $4, $a14
movi Flags, $r15
movi Display, $3
JUC $3

# 5. Perfrom arithmetic to set flags
Flags: movi 5, $10 #break look when i =5, currently line 24
movi 0, $9_i # loop counter
movi Loop, $12
Loop: addi 1, $9
# set up function call
mov $9, $a14 # set the argument address to display
movi Loop_return, $r15
movi Display, $3
JUC $3
Loop_return: cmp $10, $9
bneq $12

# 6. Use the flags to do branches (BEQ, BNEQ, JUC)
```

```
movi 0, $0
movi 1, $1
movi 30, $14
movi End, $r15
cmp $0, $1
beq $14 # shouldn't branch
# 7. Write the result into memory and display at the output

Display: and $14, $14
and $14, $14
and $14, $14
JUC $r15

End: xor $2, $2
```