Jessica Atkinson
May 8, 2017
EE310

Lab 7 Instruction Fetch

Lab 7 Objective:

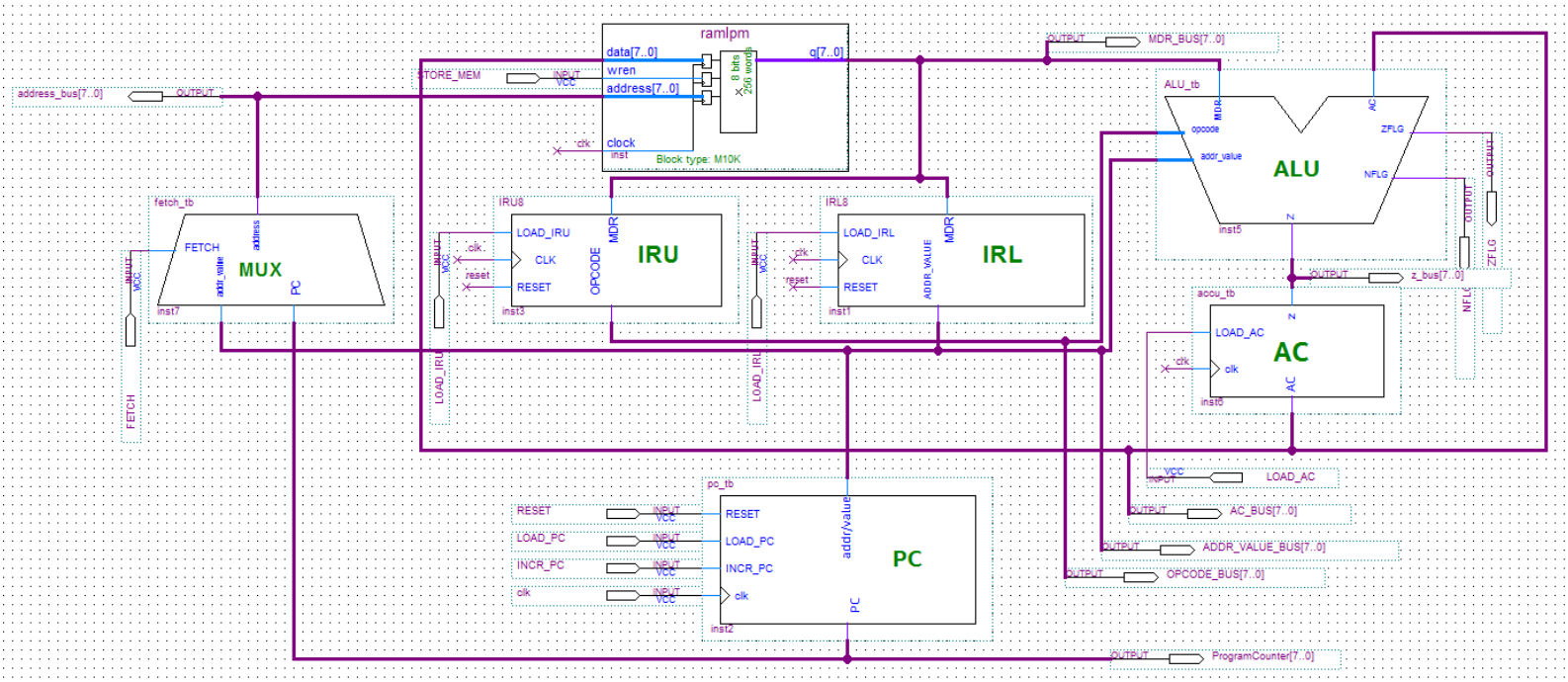
To use our previous labs 4, 5 and 6 to be able to fully integrate the ALU, RAM, registers and data paths of the microprocessor. To understand exactly what this circuit does by creating a force file that will transfer and store data accordingly. Then to create a control unit that will provide the inputs to the controls of: the opcode, system flags, reset and the system clock.

Activity 1: Hardware Design: Everything Except the Control Unit

Block Diagram:

The block diagram shown below is a combination of the code created from the previous labs. I altered the symbols for each file to match the given Block Diagram given in the Lab 7 PowerPoint. The code has also been modified in order to get all the control signals and data busses to communicate with each other. There are additional output pins to monitor their values while running the ModelSim Simulations.

bdf file



Memory Initialization File

The first twelve memory locations x”00” to x”0B” were initialized to the hexadecimal values of 02, E7, 03, 10, 09, 10, 10, 0A, 0E, 03, 04, FF. For the Memory initialization file the values had to converted into decimal representation as shown in the screenshot of the memory locations.

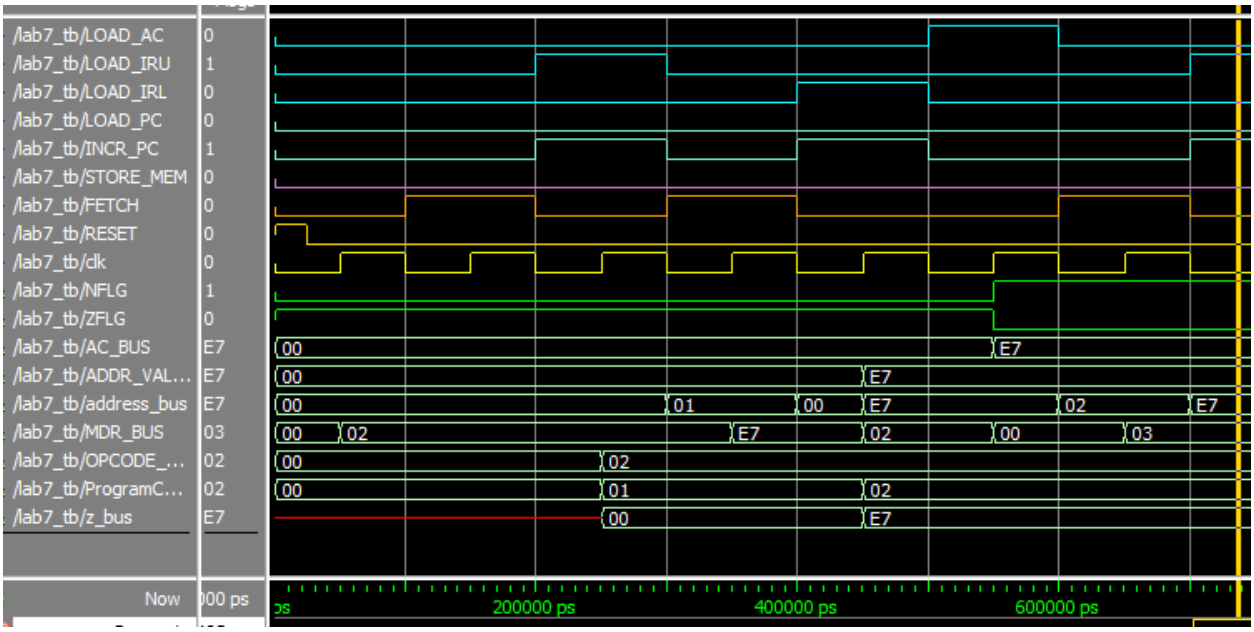
Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	2	231	03	16	09	16	16	10	.....
8	14	3	4	255	0	0	0	0	.....
16	0	0	0	0	0	0	0	0	.....
24	0	0	0	0	0	0	0	0	.....

Force File Act 1

```
############################
# EE310 Lab07 ACT1 Force File
# Demonstrate Control Signals
# Jessica Atkinson May 8, 2017
############################

# Restate Simulation when loaded
restart -force

# Reset and clear all registers
force RESET 1 0ns ;
force FETCH 0 0ns ;
force INCR_PC 0 0ns ;
force LOAD_PC 0 0ns;
force LOAD_IRL 0 0ns;
force LOAD_IRU 0 0ns;
force LOAD_AC 0 0ns;
force clk 0 0ns, 1 50ns -r 100ns;
force STORE_MEM 0 0ns;

# Run one clock cycle & turn off RESET
run 25ns;
force RESET 0 0ns;
run 75ns;

# Start repeat statements for the PrepU, FetchU, PrepL, FetchL state representations
# Made them repeat every 500ns (5 clock cycles)
force FETCH 1 0ns, 0 100ns, 1 200ns, 0 300ns -r 500ns;
force INCR_PC 0 0ns, 1 100ns, 0 200ns, 1 300ns, 0 400ns -r 500ns;
force LOAD_IRL 0 0ns, 1 300ns, 0 400ns -r 500ns;
force LOAD_IRU 0 0ns, 1 100ns, 0 200ns -r 500ns;

# From 400ns to 500ns of the repeat statements is when LOAD_AC, LOAD_PC and STORE_MEM will
execute. Following statements are asserted from the given instructions.
run 400ns;

force LOAD_AC 1 0ns;
run 100ns;

force LOAD_AC 0 0ns;
run 400ns;

force STORE_MEM 1 0ns;
run 100ns;

force STORE_MEM 0 0ns;
run 400ns;

force LOAD_AC 1 0ns;
run 100ns;

force LOAD_AC 0 0ns;
run 400ns;

force LOAD_PC 1 0ns;
run 100ns;

force LOAD_PC 0 0ns;
run 400ns;
#####################//end
```

Waveform

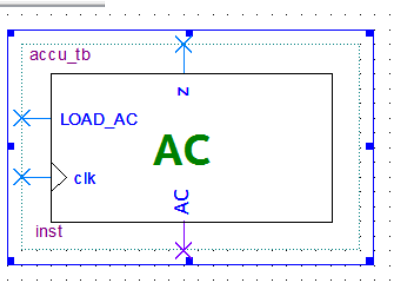
Below is the waveform produced from the compiled BDF and force file. I ran into some iteration issues at 750ns with an error. I've tried to determine the cause of this issue and couldn't seem to get rid of the error. For the first seven and a half clock cycles it appears that all the data is correctly being stored and transferred among the data busses on the RISING clock edge while the control lines only change on the FALLING clock edge giving the control signals and data busses 50ns to avoid timing issues.. ALU and FETCH are not controlled by the clock and update their data busses on both, rising and falling clock edges. I am not sure if this is what is causing the issue or if it is within the original code of the files.



VHDL Code

Accumulator

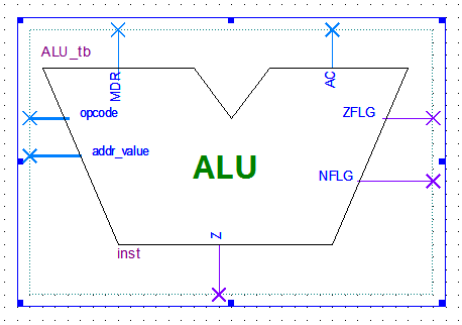
```
1  ----------------------------
2  --EE310 Lab07
3  --Accumulator
4  ----------------------------
5
6   --import library
7   library ieee;
8   use ieee.std_logic_1164.all;
9
10 entity accu_tb is
11    port( z : in std_logic_vector(7 downto 0);
12       LOAD_AC, clk : in std_logic;
13       AC : out std_logic_vector(7 downto 0));
14 end entity accu_tb;
15
16 architecture behavior of accu_tb is
17 begin
18    process(z, clk, LOAD_AC)   --sensitivity list
19    begin
20       IF LOAD_AC = '1' then          --ELSIF LOAD_AC (SW8) is on
21          IF clk'event AND clk = '1' then   --IF rising edge of clk
22             AC <= z;                --store z bus data into the AC register
23          END IF;
24       END IF;
25    end process;
26 END behavior;
```



ALU

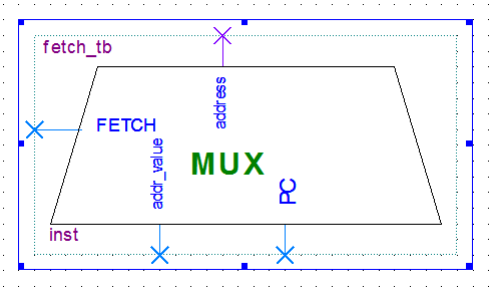
In the ALU

```
1   ----------------------------------------
2   -- EE310 Lab05 ALU
3   -- Implements the instruction set
4   -- for the uP3 microprocessor in Lab 4
5   --
6   -- Author: Jessica Atkinson, NAU/CUPT EE
7   ----------------------------------------
8
9   --declaring libraries used
10  library ieee;
11  use ieee.std_logic_1164.all;
12  use ieee.numeric_std.all;
13  library altera_mf;
14  use altera_mf.altera_mf_components.all;
15
16  --Entity
17  entity ALU_tb is
18     port( AC_IN, MDR_IN, VALUE_IN, OPCODE_IN : in std_logic_vector(7 downto 0);
19           Z_OUT : out std_logic_vector(7 downto 0);
20          --LOAD_PC, STORE_MEM : out std_logic
21          ZFLG, NFLG : out std_logic);
22  end ALU_tb;
23
24  architecture behav of ALU_tb is
25     SIGNAL  OPCODE_REG, VALUE_OUT, AC_out, MDR_out : std_logic_vector(7 downto 0);
26     SIGNAL AC_SIGNED : signed(7 downto 0);
27     -- INSTRUCTION VARIABLES
28     SIGNAL Z, NOP,  LOAD, LOADI, CLR, ADD, SUBT, NEG, NOT, ANDD, ORR, XOR, SHL, SHR : std_logic_vecto
29     SIGNAL STORE, JUMP, JNEG, JPOSZ, JZERO, JNZER : std_logic;
30     signal ADDI, SUBTI : std_logic_vector(8 downto 0);
31     signal MEM_STORE, PC_LOAD : std_logic;
32  begin
33        --LOADING TMP REGISTER TO OUTPUTS
34        Z_OUT <= Z;
35        AC_OUT <= AC_IN;
36        MDR_OUT <= MDR_IN;
37        VALUE_OUT <= VALUE_IN;
38        OPCODE_REG <= OPCODE_IN;
39        AC_SIGNED <= signed(AC_IN);
40  --    STORE_MEM <= MEM_STORE;
41  --    LOAD_PC <= PC_LOAD ;
42        ZFLG <= '1' WHEN (AC_IN = x"00") ELSE '0';
43        NFLG <= '1' WHEN (AC_SIGNED(7) = '1') ELSE '0';
44        ---------------------------------
45        -- CREATING THE INSTRUCTION SET
46        NOP <= Z;                                          --00
47        LOAD <= MDR_IN;                                    --01
48        LOADI <= VALUE_IN;                                 --02
49        STORE <= '1';                                      --03
50        CLR <= x"00";                                      --04
51        ADD <= std_logic_vector(to_signed(to_integer(signed(AC_IN)) + to_integer(signed(MDR_IN)), 8));
52        ADDI <= std_logic_vector(('0' & UNsigned(AC_IN)) + ('0' & UNsigned(VALUE_IN)));
53        SUBT <= std_logic_vector(to_signed(to_integer(signed(AC_IN)) - to_integer(signed(MDR_IN)),8));
54        SUBTI <= std_logic_vector(('0' & UNsigned(AC_IN)) - ('0' & UNsigned(VALUE_IN)));
55        NEG <= std_logic_vector(to_signed(to_integer(signed(CLR)) - to_integer(signed(MDR_IN)), 8));
56        NOT <= NOT MDR_IN;                                 --0A
57        ANDD <= (AC_IN AND MDR_IN);                        --
51        ADD <= std_logic_vector(to_signed(to_integer(signed(AC_IN)) + to_integer(signed(MDR_IN)), 8));
52        ADDI <= std_logic_vector(('0' & UNsigned(AC_IN)) + ('0' & UNsigned(VALUE_IN)));
53        SUBT <= std_logic_vector(to_signed(to_integer(signed(AC_IN)) - to_integer(signed(MDR_IN)),8));
54        SUBTI <= std_logic_vector(('0' & UNsigned(AC_IN)) - ('0' & UNsigned(VALUE_IN)));
55        NEG <= std_logic_vector(to_signed(to_integer(signed(CLR)) - to_integer(signed(MDR_IN)), 8));
56        NOT <= NOT MDR_IN;                                 --0A
57        ANDD <= (AC_IN AND MDR_IN);
58        ORR <= (AC_IN OR MDR_IN);
59        XOR <= (AC_IN XOR MDR_IN);                         --0D
60        SHL <= std_logic_vector( unsigned(AC_IN) sll to_integer(unsigned(VALUE_IN(2 downto 0))));
61        SHR <= std_logic_vector( unsigned(AC_IN) srl to_integer(unsigned(value_IN(2 downto 0))));
62        JNEG <= '1' WHEN (AC_SIGNED(7) = '1') ELSE '0';    --11  BELOW ZERO
63        JPOSZ <= '1' WHEN (JNEG='0') ELSE '0';             --12  >=0
64        JZERO <= '1' WHEN (AC_IN = CLR) ELSE '0';          --13  IF AC EQUALS 00 RETURN 1
65        JNZER <= '1' WHEN (AC_IN /= CLR) ELSE '0';         --14  IF AC != 00 THEN RETURN 1
66
67  --    -- EXECUTING THE INSTRUCTION SET
68        MEM_STORE <= STORE when (OPCODE_REG = X"03") else '0';
69        JUMP <= '1' WHEN (OPCODE_IN = x"10") ELSE '0';
70        -- OPCODE AFFECTING Z
71        with OPCODE_REG select
72           Z <= NOP when x"00",
73                LOAD when x"01",
74                LOADI when x"02",
75                CLR when x"04",
76                ADD when x"05",
77                ADDI(7 downto 0) when x"06",
78                SUBT when x"07",
79                SUBTI(7 downto 0) when x"08",
80                NEG when x"09",
81                NOT when x"0A",
82                ANDD when x"0B",
83                ORR when x"0C",
84                XOR when x"0D",
85                SHL when x"0E",
86                SHR when x"0F",
87                NOP when OTHERS;
88        -- OPCODE AFFECTING LOAD_PC
89        with OPCODE_REG select
90           PC_LOAD <= JUMP when x"10",
91                      JNEG when x"11",
92                      JPOSZ when x"12",
93                      JZERO when x"13",
94                      JNZER when x"14",
95                       '0' when OTHERS;
96  end behav;
97
```



Fetch

This was a simple multiplexer that chose the PC bus or the ADDR_VAL bus based on the value of FETCH.



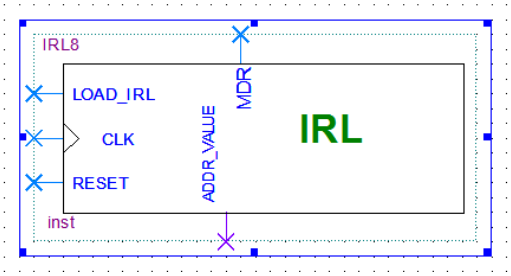
FETCH_tb.vhd

```
1  ----------------------------------------
2  --EE310 lab 07
3  -- Multiplexer - Fetch
4  --Jessica Atkinson
5  ----------------------------------------
6
7  --importing library
8  library ieee;
9  use ieee.std_logic_1164.all;
10
11 entity fetch_tb is
12    --declaring variable properties
13    port( FETCH : in std_logic;
14       addr_value, pc : in std_logic_vector(7 downto 0);
15       address : out std_logic_vector(7 downto 0));
16 end entity fetch_tb;
17
18 architecture behavior of fetch_tb is
19 begin
20    address <= PC when (FETCH = '1') else addr_value;
21 END behavior;
```

Instruction Registers

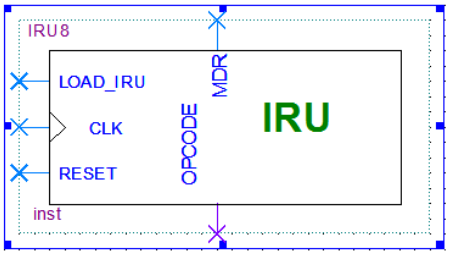
For the instruction registers, initially, I had created one VHDL file that held both the IRL and IRU. For this lab, I took that code and separated it into the following two files to represent the Upper and Lower Instruction register separately.

IRL.vhd




```
1  ------------------------
2  -- EE310 Lab#7
3  -- Jessica Atkinson
4  -- NAU
5  ------------------------
6  -- Instruction Register Lower
7  ------------------------
8
9  --declaring libraries used
10 library ieee;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13 library altera_mf;
14 use altera_mf.altera_mf_components.all;
15
16 entity IRL8 is
17    port( LOAD_IRL, CLK, RESET : in std_logic;
18       MDR : in std_logic_vector(7 downto 0);
19       ADDR_VALUE : out std_logic_vector(7 downto 0));
20 end IRL8;
21
22 architecture behav of IRL8 is
23 begin
24    process(CLK, LOAD_IRL, MDR, RESET)
25    begin
26       if RESET = '1' then
27          ADDR_VALUE <= x"00";
28       --if CLK on FALLING edge and LOAD_IRU =1 then MDR = OPCODE
29       elsif clk'event AND clk = '0' then
30          if (LOAD_IRL = '1') then
31             ADDR_VALUE <= MDR;
32          end if;
33       end if;
34    end process;
35 end behav;
```

IRU.vhd



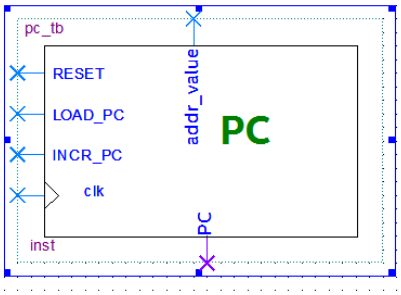

```
----------------------
-- EE310 Lab#7
-- Jessica Atkinson
-- NAU
----------------------
-- Instruction Register UPPER
----------------------

--declaring libraries used
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library altera_mf;
use altera_mf.altera_mf_components.all;

entity IRU8 is
   port( LOAD_IRU, CLK, RESET : in std_logic;
      MDR : in std_logic_vector(7 downto 0);
      OPCODE : out std_logic_vector(7 downto 0));
end IRU8;

architecture behav of IRU8 is
begin
   process(CLK, LOAD_IRU, MDR, RESET)
   begin
      if RESET = '1' then
         OPCODE <= x"00";
      --if CLK on FALLING edge and LOAD_IRU =1 then MDR = OPCODE
      elsif (LOAD_IRU = '1') then
         if clk'event AND clk = '1' then
            OPCODE <= MDR;
         end if;
      end if;
   end process;
end behav;
```

Program Counter

The program counter increments by one to execute the data stored in the memory location value. This is how the 16-bit data can be stored in four states at which PrepU/L and FetchU/L occur. Storing the upper and lower byte in two separate memory locations in order to store the next bit by incrementing the PC by one.

This is the file I had the most errors with. The code created for the previous lab would only hold x“00” and increment by one to x”01” then back down to x”00” in a loop.

pc_tb.vhd




```
----------------------------------
-- EE310   lab07
-- Program Counter
-- Jessica Atkinson
----------------------------------

--import libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library altera_mf;
use altera_mf.altera_mf_components.all;


entity pc_tb is
   --declaring i/o ports
   port( RESET, clk : in std_logic;
      addr_value : in std_logic_vector(7 downto 0);      --ir_lower
      LOAD_PC, INCR_PC : in std_logic;
      --ProgramCounter : out std_logic_vector(7 downto 0)); --pc bus
      pc_out : out std_logic_vector(7 downto 0)); --pc bus
end entity pc_tb;

architecture behavior of pc_tb is
   --signal pc_out : std_logic_vector(7 downto 0);     -- TMP VARs for increments
   signal pc_store : std_logic_vector(7 downto 0);
begin
   pc_out <= pc_store;
   process(clk, RESET, addr_value, INCR_PC, LOAD_PC, pc_store)
   begin
      IF RESET = '1' then                    --IF reset active
         --pc_out <= "00000000";                  --clear register
         pc_store <= "00000000";              --clear store
      ELSIF clk'event AND clk = '1' then  --ELSIF clk rising edge triggered
         IF LOAD_PC = '1' then                  --IF LOAD_PC active
            --pc_out <= addr_value;         --load addr_value
            pc_store <= addr_value;
         ELSIF INCR_PC = '1' then           --IF INCR_PC active
            --pc_out <= pc_store + 1;          --increment + 1 PER KEY0 ACTIVE
            pc_store <= std_logic_vector(unsigned(pc_store) + 1);


         END IF;
      END IF;
   end process;
END behavior;
```

Activity 2: Control Unit for Instruction Fetch

control_unit.vhd

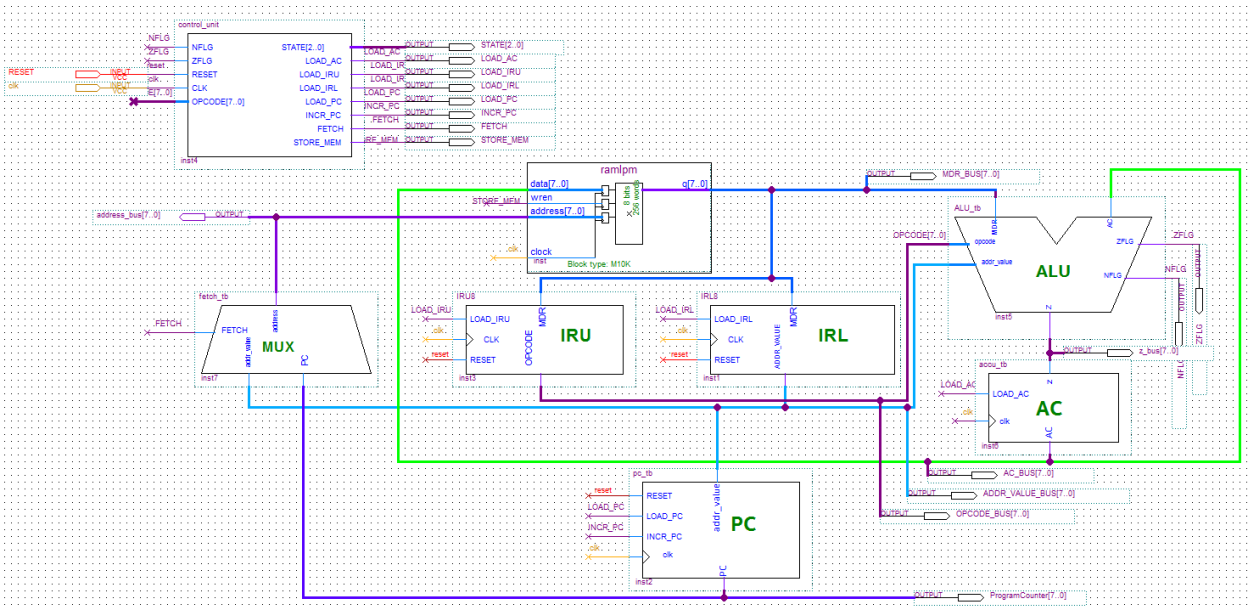
Control Unit

The control unit was straight forward on how to structure the VHDL code. Declaring the state_types then variables that will indicate what state it is at. While alternating various signals to go off for each state, there is an if statement that will determine if the third bit in the STATE variable is a 1. If it is a one then it returns the states to the start if not it moves to the next. Once reaching the last state, FetchL it will skip the start state unless reset is being pressed and move on to PrepU.

```
----------------------------------
-- EE310 Control Unit
-- Created: May 8, 2018
-- Creator: Jessica Atkinson
-- 5-State Finite State Machine
----------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
   port( NFLG, ZFLG, RESET, CLK : in std_logic;
      OPCODE : in std_logic_vector(7 downto 0);
      STATE : BUFFER std_logic_vector(2 downto 0);
      LOAD_AC, LOAD_IRU, LOAD_IRL, LOAD_PC, INCR_PC, FETCH, STORE_MEM : out std_logic);
end control_unit;

architecture behavior of control_unit is
   type state_type is (start, PrepU, FetchU, PrepL, FetchL);
   signal present_state, next_state, LAST_STATE : state_type;
   signal AC_LOAD, MEM_STORE : STD_LOGIC;
begin

   sync_proc:
   process(CLK, RESET)
   begin
      --
      if RESET = '1' then
         present_state <= start;
      elsif (clk'event and CLK = '0') then
         present_state <= next_state;
      end if;
   end process;

   comb_proc:
   process(present_state, next_state)
   begin

      case present_state is
                                 -- start
         when start =>
            next_state <= PrepU;
            STATE <= "100";
            FETCH <= '0';
            LOAD_IRL <= '0';
            LOAD_IRU <= '0';
            INCR_PC <= '0';

                                 -- PrepU
         when PrepU =>
            STATE <= "000";
            FETCH <= '1';
            LOAD_IRL <= '0';
            INCR_PC <= '0';
            LOAD_AC <= '0';
            if STATE(1) = '1' then
               next_state <= START;
            else
               next_state <= FetchU;
            end if;
                                 -- FetchU
         when FetchU =>
            STATE <= "001";
            FETCH <= '0';
            LOAD_IRU <= '1';
            INCR_PC <= '1';
            if STATE(1) = '1' then
               next_state <= start;
            else
               next_state <= PrepL;
            end if;
                                 -- PrepL
         when PrepL =>
            STATE <= "010";
            FETCH <= '1';
            LOAD_IRU <= '0';
            INCR_PC <= '0';
            if STATE(2) = '1' then
               next_state <= start;
            else
               next_state <= FetchL;
            end if;
                                 -- FetchL
         when FetchL =>
            STATE <= "011";
            FETCH <= '0';
            LOAD_IRL <= '1';
            INCR_PC <= '1';
            if STATE(2) = '1' then
               next_state <= start;
            else
               next_state <= PrepU;
            end if;
      end case;
   end process;
end behavior;
```

Block Diagram File

I added the new control box above the original block diagram and labeled the bus lines so they will update accordingly and not have too many wires to follow.



Force File

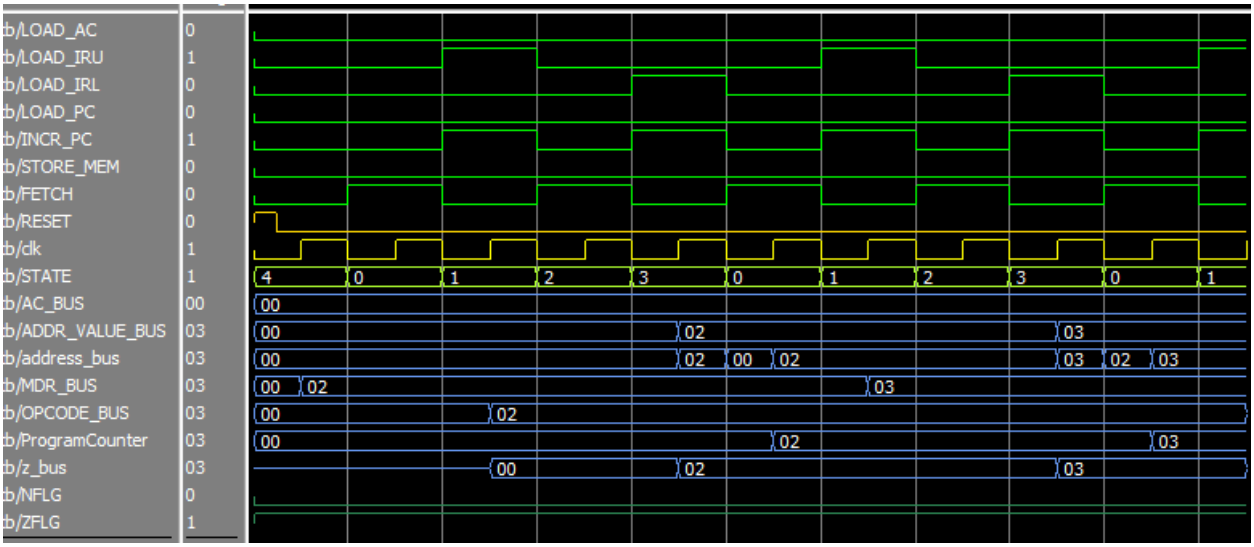
The force file didn't have many operations. This is because many of the inputs are coming from a different location and do not need to be set. Therefore the 8-bit OPCODE input comes from the OPCODE_BUS[7..0]. The ZFLG and NFLG are controlled from the output of the ALU. This leaves the clock and reset as the controls that can be altered in the force file. Therefore, I put the clock in a 100ns cycle alternating from 0 to 1 every 50ns. Setting reset to '1' for the first 25ns and zero for the rest of the simulation.

```
##############################
# EE310 Lab07
# Force File
# Demonstrate Control Signals
##############################

restart -force

# Reset and clear all registers
force RESET 1 0ns ;
force clk 0 0ns, 1 50ns -r 100ns;

# Set Reset=0 then run
run 25ns;
force RESET 0 0ns;
run 1075ns;
##########################//end
```

Waveform

This waveform below shows the STATE values starting at '100' := RESET then it cycles from '000' PrepU to '001' FetchU to '010' PrepL to '011' FetchL. This cycle is set to repeat and only include the 4th state if and only if the reset button is being pressed.

The inputs are in green and orange. The outputs include the STATE signal in lime-green and the busses in blue and the NFLG and ZFLG in green at the bottom. For this waveform the iteration limit error (vsim-3601) reached at 1050 ns in the middle of the 11th clock cycle during the FetchU state. I have written numerous versions of the force file and all of them stop on the rising edge of the FetchU state.



The table below shows what the signals should be during each state as well as their state value.

	1	2	3	4	5
	Start	**PrepU**	**FetchU**	**PrepL**	**FetchL**
RESET	1	0	0	0	0
STATE(3)	100	000	001	010	011
LOAD_AC	0	0	0	0	0
LOAD_IRU	0	0	1	0	0
LOAD_IRL	0	0	0	0	1
LOAD_PC	0	0	0	0	0
INCR_PC	0	0	1	0	1
FETCH	0	1	0	1	0
STORE_MEM	0	0	0	0	0

Conclusion

This lab showed me how to connect the previous labs to make a basic microprocessor. I ran into many issues including having to change the active RESET from '0' to '1' in all the registers as well as changing a few clock edges to activate on the rising edge. My program counter's VHDL has the most issues. Overall, this lab was helpful to realize how each part contributed to the microprocessor and what control lines operate different registers and update different data busses.