Zack Fravel

010646947

Lab 1 : M 4:10 - 5:55 PM

Lab 4 - Simple Datapath

3/7/16

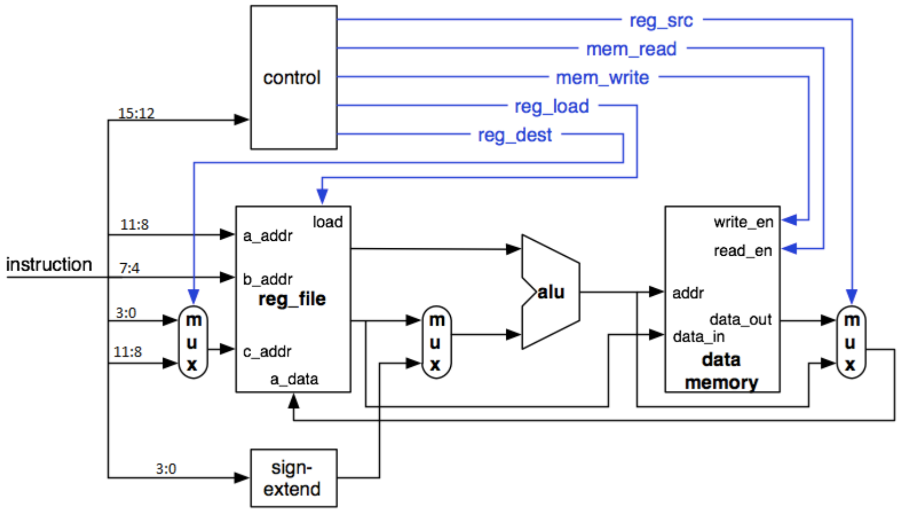
Introduction

The objective of lab 4 was to finally start to put the pieces we have been building together to form a simple datapath, which is the basis for our simple CPU we wish to implement. To do this, we will use the previous ALU and Register file components along with a few more we developed in this lab and others given to us. This was a two part lab, the first part allowing for ADD, ADDi, SUB, SUBi, AND, and OR operations to be performed. In the second part, we added functionality for LW and SW (load word and store word) operations. We were given a test bench and memory file to test our final design.

Approach

Since this was a two week lab, we had two main objectives to complete by the end. The first part was to add our control and sign extend units to allow the first six instruction sets to be able to be implemented. These are our basic arithmetic instructions, along with their immediate counterparts, as well as AND and OR logical. The sign extend unit allows the datapath to take in a four-bit signed number (instruction bits 3:0 or “offset”) and “extend” it into a sixteen-bit signed number. For example, “0011” would become “0000000000000011” and “1000” would become “1111111111111000.” This allows us to ADDi and SUBi. The design for this unit is fairly simple, basically we just took the most significant bit and replicated it twelve times and added it before the rest of the number. The VHDL for my sign_extend design is attached on the back of this report.

Another main part of this lab was adding a control unit to our datapath. The control unit takes in the opcode for our design ISA, which is instruction bits 15:12. This four bit opcode allows the control unit to, as the name implies, control the datapath to math the specified functionality. Our control unit is comprised of an opcode input and the rest outputs, mostly std_logic and a single std_logic_vector(1 downto 0). These outputs are the control signals for the required mux's in our design, as well as determine the register load, memory read/write, and of course the function of the ALU. Below is a diagram of the datapath which more clearly shows how the control unit works and changes the datapath. The VHDL is attached to the back. The way it works is basically we created case statements for the different opcode possibilities we want to account for and set the output signals to correctly match the datapath with the instruction.



In order to accommodate load word and store word operations, the data memory unit was given to us. The way the data memory works is it takes in memory from a file in the format of a 256 wide array of 8 bit numbers. The memory stores each 16 bit number we have in two different "memory slots" so when we load or store a word we are accessing 2 of the 256 slots, allowing for storage of 128 different 16 bit values. The memory unit also has a mem_dump functionality that dumps the contents of the memory to a file.

Finally, in order to put all this together we had to create a top-level entity named system that is used to connect all of our different units together in the layout shown above using port mapping. The process involved creating signals for all the different outputs that involved in the circuit design so we are able to “wire” them up to each component as necessary. Creating signals for these outputs also allows us to send certain information to multiple units at one time. The system entity has a 16 bit instruction input, as well as clock and reset inputs, and finally the mem_dump input. The VHDL for my system design is attached in the back. Once this was done, we had a fully functional simple datapath that allows for ADD, ADDi, SUB, SUBi, AND, OR, LW, and SW.

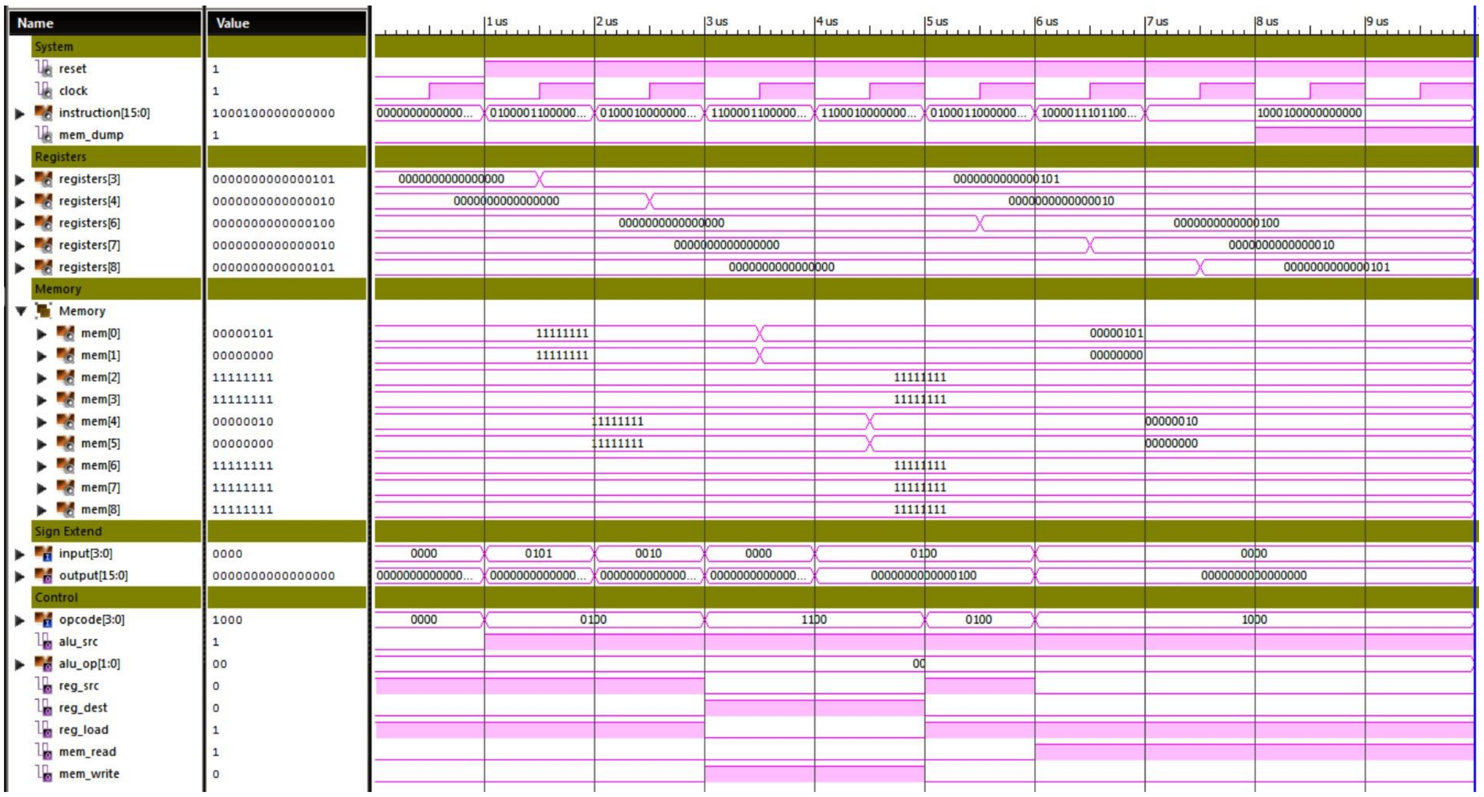
Experimentation

The first part of the lab, getting the first six functions to work, was fairly straight forward and didn’t require much debugging. The most confusing part of the lab was making sure I had not created any duplicate signals or naming errors in connecting all the units correctly in the system entity. The control unit was also a little tricky to get working initially. The main issue I was running into was visualizing the datapath correctly for LW and SW operations. However, after tracing through each instruction is became clear which signals needed to be set to what. The main thing to get right is which path the mux’s are allowing and making sure the ALU is performing immediate addition, because its calculating an address from an immediate value.

Once I worked out all the errors that were giving me false positive results, I was able to run the given test bench and show that the datapath works with the ISA provided as described in the lab.

Results

The test bench given to us initializes the memory to all “11111111” and shows functionality with ADDi, SW, and LW. The test bench adds immediate numbers in registers r3 and r4, stores r3 and r4 in memory slots M[0] and M[4] respectively, does another ADDi to r6, and performs a LW on of r6 to r7 and r0 to r8. Finally, the test bench sets mem_dump to one and prints the memory contents to a file. The input and output memory files are also attached to the back. Below is the simulation waveform results for the test bench.



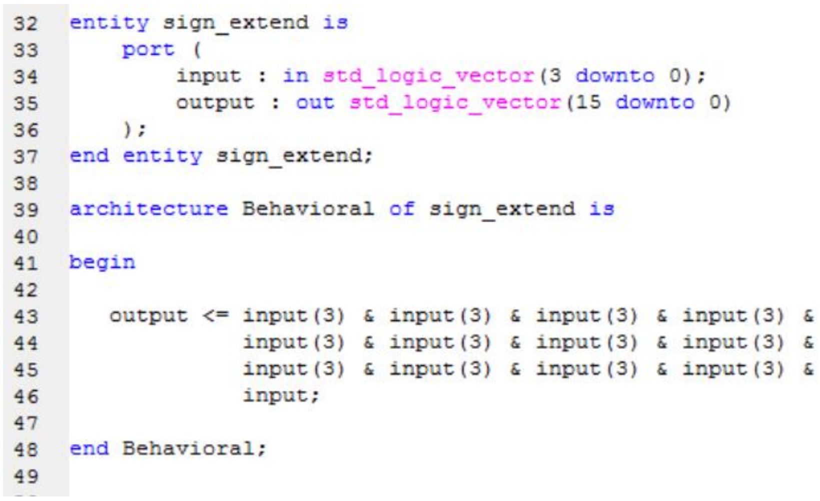
```
driver : process is
begin
    -- reset the system
    reset <= '0'; instruction <= x"0000"; wait for tick;
    reset <= '1';

    -- ADDI r3, r0, 5    (r3 = 5)
    instruction <= x"4305"; wait for tick;

    -- ADDI r4, r0, 2    (r4 = 2)
    instruction <= x"4402"; wait for tick;

    -- SW r3, 0(r0)      (M[0] = 5)
    instruction <= x"C300"; wait for tick;

    -- SW r4, 4(r0)      (M[4] = 2)
    instruction <= x"C404"; wait for tick;

    -- ADDI r6, r0, 4    (r6 = 4)
    instruction <= x"4604"; wait for tick;

    -- LW r7, 0(r6)      (r7 = 2)
    instruction <= x"8760"; wait for tick;

    -- LW r8, 0(r0)      (r8 = 5)
    instruction <= x"8800"; wait for tick;

    -- Dump the final data memory into a file
    mem_dump <= '1';
    wait;
end process driver;
```

It can be seen that each time the alu_op signal changes the operation of the datapath changes as well. This is determined by the opcode in the instruction set used by our control unit. Along with each different instruction, if you take a look at the control waveforms you can see the mux and ALU controllers change with each instruction.

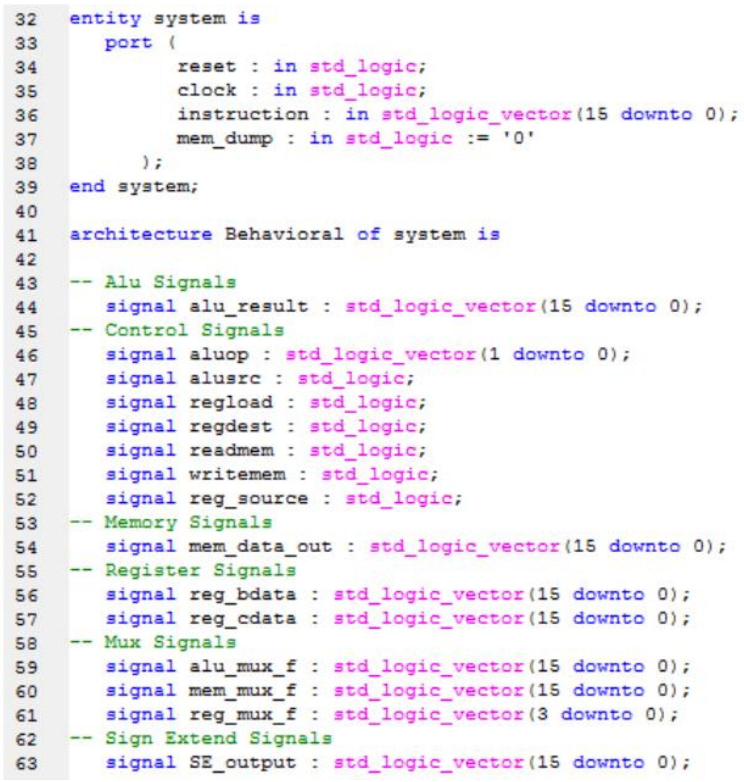
Conclusion

The test bench above shows that the simple datapath works as described in the original lab. We have now implemented most of what we need for the ISA, only a few instruction types remain to have a fully functional simple CPU. This lab was useful in showing how we are actually putting all the pieces together and how to create an actual circuit in the design tools. Along with this, the lab also exposed me to the use of case-statements for the control unit, which is very useful for implementing an instruction set.

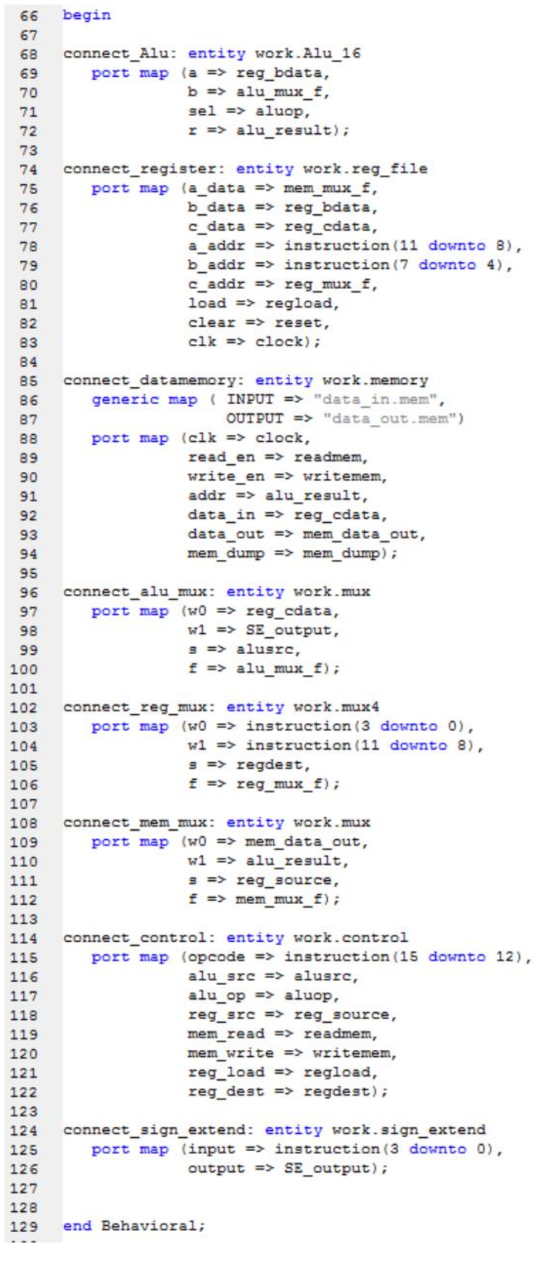
VHDL



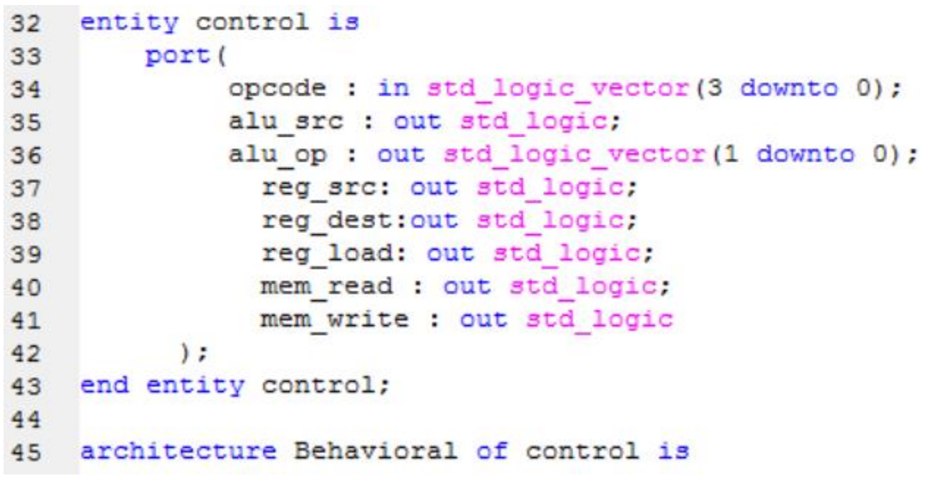
```vhdl
entity sign_extend is
    port (
        input : in std_logic_vector(3 downto 0);
        output : out std_logic_vector(15 downto 0)
    );
end entity sign_extend;

architecture Behavioral of sign_extend is

begin

   output <= input(3) & input(3) & input(3) & input(3) &
             input(3) & input(3) & input(3) & input(3) &
             input(3) & input(3) & input(3) & input(3) &
             input;

end Behavioral;

```



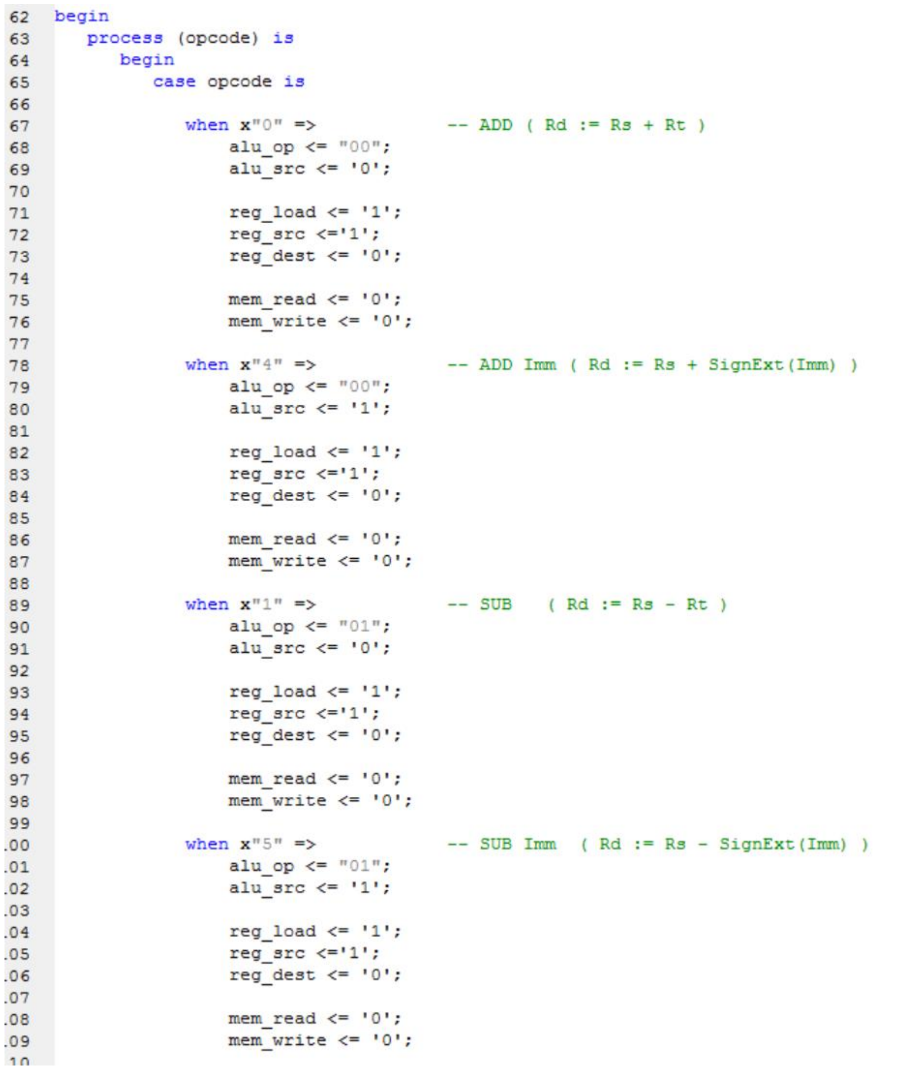
```vhdl
entity system is
   port (
         reset : in std_logic;
         clock : in std_logic;
         instruction : in std_logic_vector(15 downto 0);
         mem_dump : in std_logic := '0'
      );
end system;

architecture Behavioral of system is

-- Alu Signals
   signal alu_result : std_logic_vector(15 downto 0);
-- Control Signals
   signal aluop : std_logic_vector(1 downto 0);
   signal alusrc : std_logic;
   signal regload : std_logic;
   signal regdest : std_logic;
   signal readmem : std_logic;
   signal writemem : std_logic;
   signal reg_source : std_logic;
-- Memory Signals
   signal mem_data_out : std_logic_vector(15 downto 0);
-- Register Signals
   signal reg_bdata : std_logic_vector(15 downto 0);
   signal reg_cdata : std_logic_vector(15 downto 0);
-- Mux Signals
   signal alu_mux_f : std_logic_vector(15 downto 0);
   signal mem_mux_f : std_logic_vector(15 downto 0);
   signal reg_mux_f : std_logic_vector(3 downto 0);
-- Sign Extend Signals
   signal SE_output : std_logic_vector(15 downto 0);
```



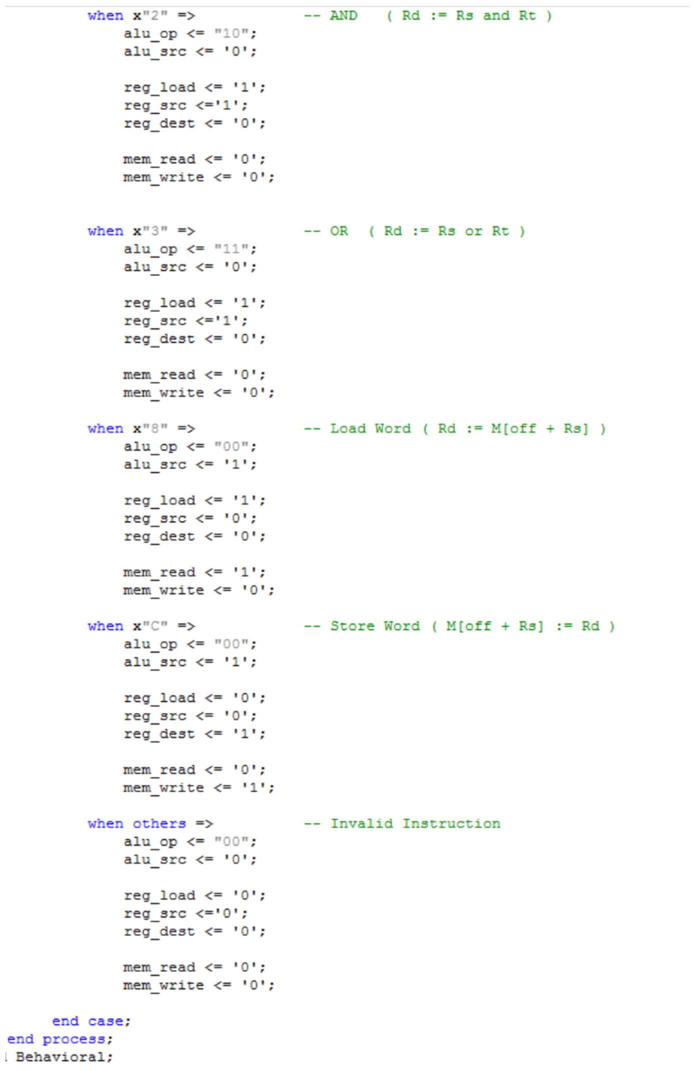
```vhdl
begin

connect_Alu: entity work.Alu_16
   port map (a => reg_bdata,
             b => alu_mux_f,
             sel => aluop,
             r => alu_result);

connect_register: entity work.reg_file
   port map (a_data => mem_mux_f,
             b_data => reg_bdata,
             c_data => reg_cdata,
             a_addr => instruction(11 downto 8),
             b_addr => instruction(7 downto 4),
             c_addr => reg_mux_f,
             load => regload,
             clear => reset,
             clk => clock);

connect_datamemory: entity work.memory
   generic map ( INPUT => "data_in.mem",
                 OUTPUT => "data_out.mem")
   port map (clk => clock,
             read_en => readmem,
             write_en => writemem,
             addr => alu_result,
             data_in => reg_cdata,
             data_out => mem_data_out,
             mem_dump => mem_dump);

connect_alu_mux: entity work.mux
   port map (w0 => reg_cdata,
             w1 => SE_output,
             s => alusrc,
             f => alu_mux_f);

connect_reg_mux: entity work.mux4
   port map (w0 => instruction(3 downto 0),
             w1 => instruction(11 downto 8),
             s => regdest,
             f => reg_mux_f);

connect_mem_mux: entity work.mux
   port map (w0 => mem_data_out,
             w1 => alu_result,
             s => reg_source,
             f => mem_mux_f);

connect_control: entity work.control
   port map (opcode => instruction(15 downto 12),
             alu_src => alusrc,
             alu_op => aluop,
             reg_src => reg_source,
             mem_read => readmem,
             mem_write => writemem,
             reg_load => regload,
             reg_dest => regdest);

connect_sign_extend: entity work.sign_extend
   port map (input => instruction(3 downto 0),
             output => SE_output);


end Behavioral;
```



```
entity control is
    port(
         opcode : in std_logic_vector(3 downto 0);
         alu_src : out std_logic;
         alu_op : out std_logic_vector(1 downto 0);
           reg_src: out std_logic;
           reg_dest:out std_logic;
           reg_load: out std_logic;
           mem_read : out std_logic;
           mem_write : out std_logic
        );
end entity control;

architecture Behavioral of control is
```



```
begin
   process (opcode) is
      begin
         case opcode is

            when x"0" =>            -- ADD ( Rd := Rs + Rt )
                alu_op <= "00";
                alu_src <= '0';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';

            when x"4" =>            -- ADD Imm ( Rd := Rs + SignExt(Imm) )
                alu_op <= "00";
                alu_src <= '1';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';

            when x"1" =>            -- SUB   ( Rd := Rs - Rt )
                alu_op <= "01";
                alu_src <= '0';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';

            when x"5" =>            -- SUB Imm  ( Rd := Rs - SignExt(Imm) )
                alu_op <= "01";
                alu_src <= '1';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';
```



```
            when x"2" =>            -- AND   ( Rd := Rs and Rt )
                alu_op <= "10";
                alu_src <= '0';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';


            when x"3" =>            -- OR  ( Rd := Rs or Rt )
                alu_op <= "11";
                alu_src <= '0';

                reg_load <= '1';
                reg_src <='1';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';

            when x"8" =>            -- Load Word ( Rd := M[off + Rs] )
                alu_op <= "00";
                alu_src <= '1';

                reg_load <= '1';
                reg_src <= '0';
                reg_dest <= '0';

                mem_read <= '1';
                mem_write <= '0';

            when x"C" =>            -- Store Word ( M[off + Rs] := Rd )
                alu_op <= "00";
                alu_src <= '1';

                reg_load <= '0';
                reg_src <= '0';
                reg_dest <= '1';

                mem_read <= '0';
                mem_write <= '1';

            when others =>          -- Invalid Instruction
                alu_op <= "00";
                alu_src <= '0';

                reg_load <= '0';
                reg_src <='0';
                reg_dest <= '0';

                mem_read <= '0';
                mem_write <= '0';

      end case;
   end process;
 Behavioral;
```

Memory Contents

```
00000101
00000000
11111111
11111111
00000010
00000000
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
```

goes on for 256 lines with "11111111"