Abraham McIlvaine, Benjamin Steenkamer
CPEG 324-010
Lab 3 Report
5/3/17

Lab #3: A Single Cycle Calculator in VHDL

Abstract:

The main goal of this lab was to create a single cycle calculator in VHDL using our previously designed 8 bit ISA. The second goal was to create a VHDL test bench which tested the calculator's functionality and verified the correctness of our ISA implementation. The motivation for these goals is for us to better understand how the datapath of a simple calculator is implemented at the register transfer level (RTL). The key issues being addressed in this lab are creating and combining datapaths based on an ISA, how control signals are generated, how to use VHDL to implement a schematic, and how to verify the functionality of a VHDL implementation using test benches.

Overall, we were able to successfully implement our 8 bit ISA as a datapath and then translate that datapath into VHDL code. The end result is a simple 8 bit calculator that can add, subtract, load immediate values, skip 1 or 2 instructions, and print the contents of a register. This functionality was verified with a test bench file that showed the expected outcomes of various combinations of instructions. The main method used to create the calculator was first creating the individual datapaths for each instruction. We then combined each datapath together, adding control signals, to create the complete datapath. In VHDL, we created modules that implemented only a single section of our datapath. Using hierarchal structures in VHDL, we created instances of these components and connected them together using port maps and control logic.

Division of Labor:

This lab had three main components: datapath design, VHDL implementation, and implementation verification. The design of the datapath involved creating the individual circuit schematics for each calculator instruction. Each datapath was then combined into one diagram and various control signals were added. Once the datapath schematic was complete, the circuit was then implemented in VHDL. This involved creating individual VHDL modules that were combined into a hierarchical structure. Control signals and port mappings were used to connect each component together, as decided in the datapath schematic. Finally, a VHDL test bench was created to drive the calculator's inputs with a set of tests. These tests were designed so that every combination of the instructions were verified for correctness.
Below is a breakdown of the group responsibilities:
Datapath schematic:

- Add / Subtract datapath schematic: Abraham and Benjamin
- Load Immediate datapath schematic: Abraham
- Print datapath schematic: Benjamin
- Compare datapath schematic: Abraham
- Top level calculator datapath schematic: Abraham and Benjamin

VHDL implementation:

- Add / Subtract VHDL module: Abraham and Benjamin
- Load Immediate VHDL module: Abraham and Benjamin
- Print VHDL module: Abraham and Benjamin
- Compare VHDL module: Abraham
- Top level calculator VHDL module: Abraham and Benjamin

Verification:

- Creating the test bench VHDL file that reads in instructions and drives the calculator: Benjamin
- Creating a set of instruction tests to be read in and determining the expected outputs: Benjamin
- Running and verifying the results: Abraham and Benjamin

Writing the lab report: Abraham and Benjamin

Detailed Strategy:

Adder / Subtracter Instruction Datapath:

The add / subtract portion of our datapath either adds or subtracts two registers, and then stores the result in a third register.

Add:

OP-code		Destination Register		Operand Registers			
0	1	RD	RD	RS	RS	RT	RT

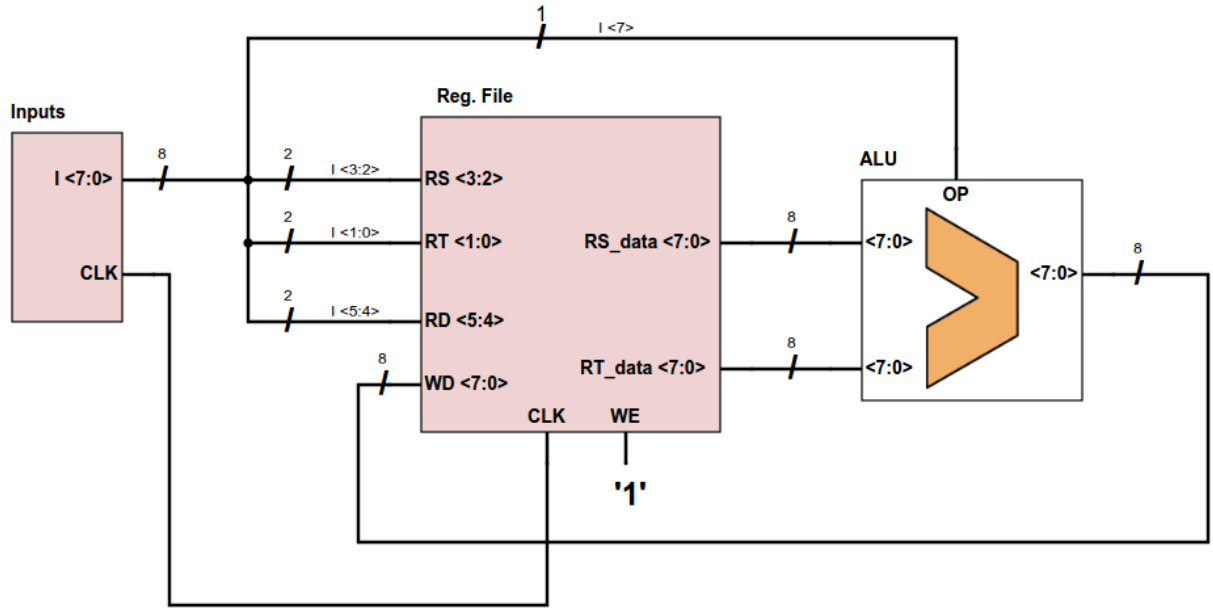
RD[data] = RS[data] + RT[data]

Subtract:

OP-code		Destination Register		Operand Registers			
1	0	RD	RD	RS	RS	RT	RT

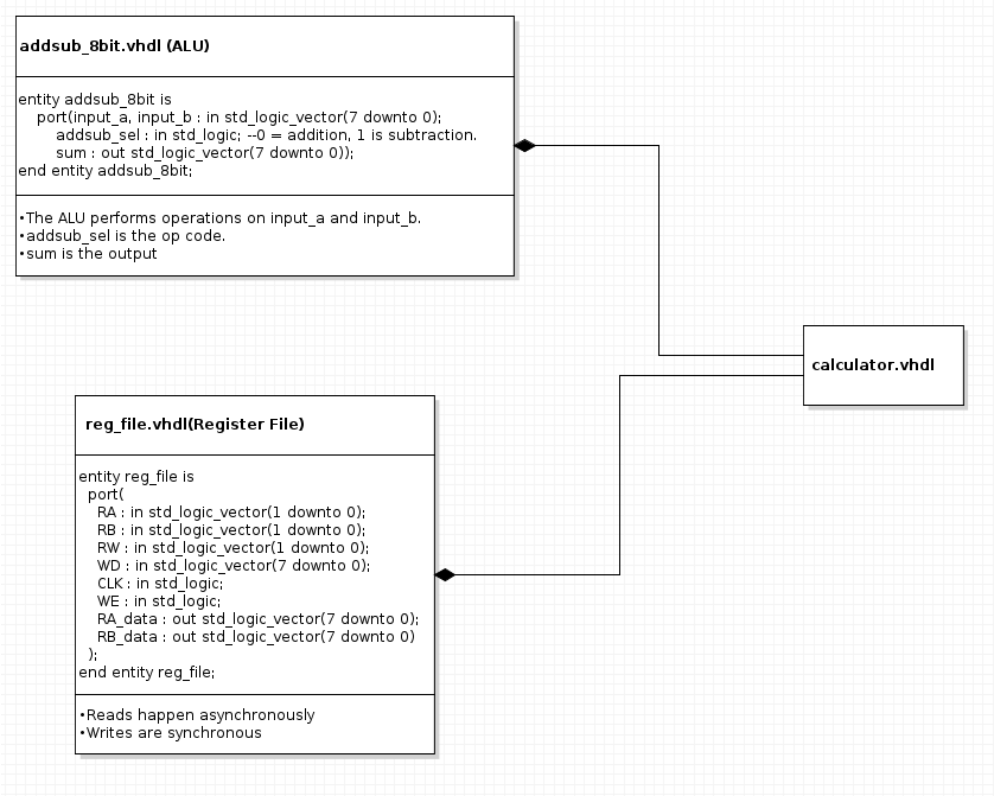
RD[data] = RS[data] – RT[data]

The isolated datapath for adding and subtracting is shown below.



Register RS comes from bits 3 and 2 of the instruction, RT comes from bits 1 and 0, and RD comes from bits 5 and 4. The ALU receives I<7> as an op-code, which determines whether the ALU will perform addition or subtraction. The output of the ALU then gets sent back to the write data input of the register file.

When implementing this in VHDL, we used “addsub_8bit.vhdl” and “reg_file.vhdl” shown in Appendix II. “addsub_8bit.vhdl” implements the functionality of the ALU. The 8 bit ALU is made up of 8 full adders, which are each made up of two half adders. “reg_file.vhdl” implements the functionality of the register file (Note: “reg_file.vhdl” is also used in the other datapaths as well). A UML block diagram of the datapath is shown below. This diagram shows how we used a hierarchical structure of components to create this datapath. Also, we made the design decision for the ALU to simply ignore underflow or overflow when it occurs. This choice was made as we did not have a way to signal that overflow or underflow occurred. The trade off is that our datapath is simpler but overflow and underflow can occur without warning.



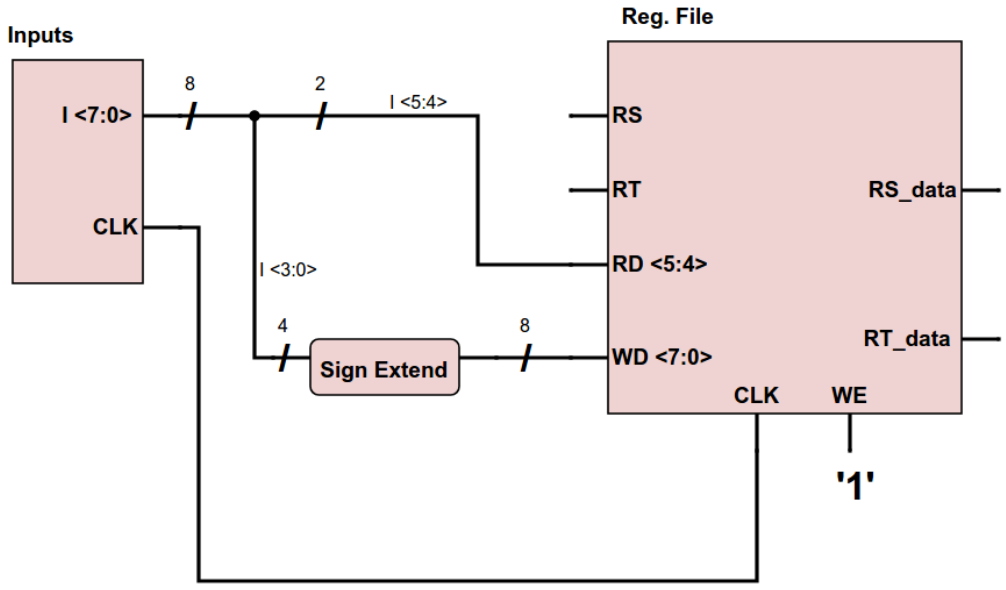
Load Immediate Instruction Datapath:

The load immediate instruction loads a 4 bit immediate signed value into a given register. The immediate value is sign-extended to 8 bits.

Op-code		Destination Register		Signed Immediate Value			
1	1	RD	RD	I_3	I_2	I_1	I_0

RD = 0b$I_3I_3I_3I_3I_2I_1I_0$

The isolated datapath for load immediate is shown below.



The RD register comes from bits 5 and 4 of the instruction, and the immediate value comes from bits 3 to 0. This value is sign extended and sent to the WD input of the register file. When implementing this in VHDL, we used the same register file component as described previously. To implement the sign extender, we used a simple with-select statement.

```
--Sign extedend the immediate value.
sign_ext_imm(3 downto 0) <= I(3 downto 0);
with I(3) select sign_ext_imm(7 downto 4) <=
  "1111" when '1',
  "0000" when others;
```

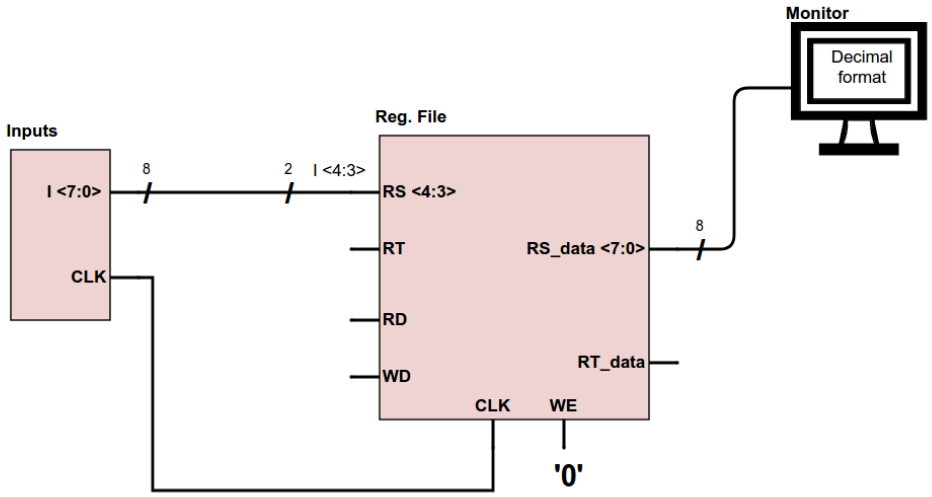
This statement represents a multiplexer at the physical level.

Print Instruction Datapath:
The print instruction prints out the contents of a register.

Op-code			Source Register		Don't care		
0	0	0	RS	RS	X	X	X

RS[data] is printed to the console, in right justified format.

The isolated datapath is shown below.



The register to print is given by bits 4 and 3 of the instruction, and the data in the register is gotten from the register file and printed out. In VHDL, we used the "report" statement to report the value of the register to the console. The value of the register is converted from a std_logic_vector to an integer and printed using the following code: `report integer'image(to_integer(signed(RS_data)));`

Since we do not need the bottom three bits of the instruction for the operation, their values do not matter. In our original ISA design, these bottom three bits where used to decide the formatting of the printout, however the requirements of this lab stated that the output must be in decimal form. We made the design decision to have these three bits be "don't cares" as their values are ignored and only the top five bits are used.

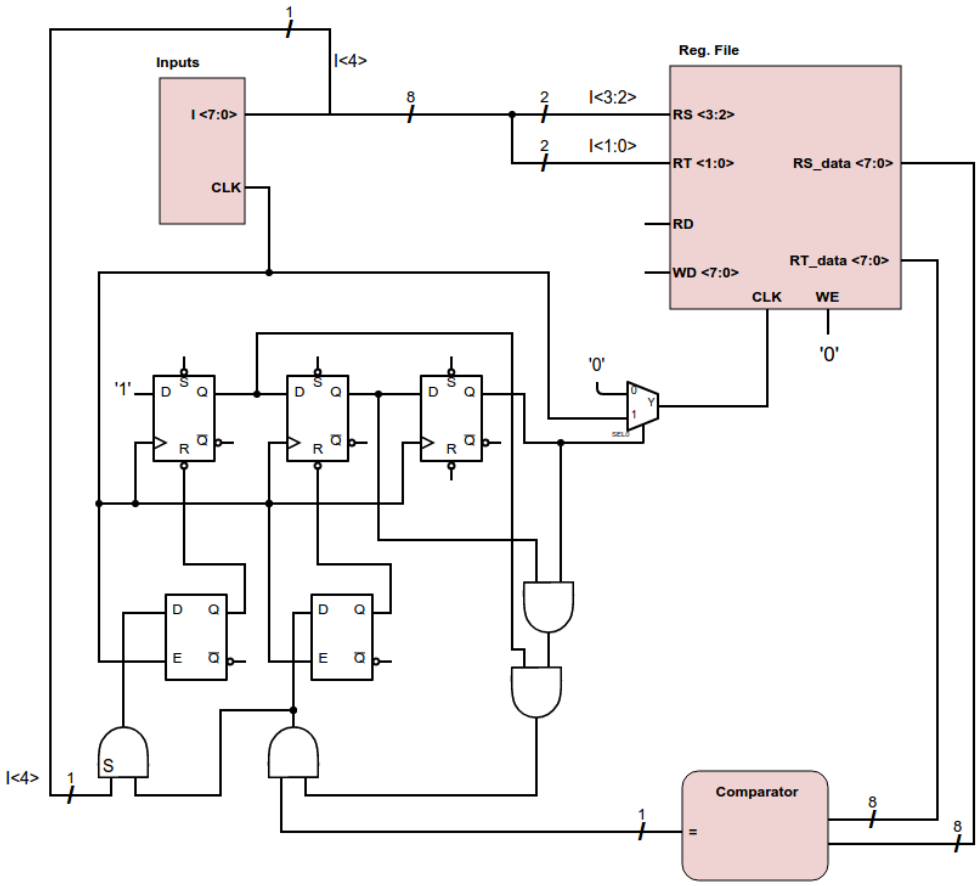
Compare Instruction Datapath:

The compare operation compares two registers. If they are equal, the next 1 or 2 instructions are skipped, according to the parameter "S" that is encoded into the instruction. If they are not equal, then the next instruction gets executed normally. The instruction format for compare registers is shown below.

Op-code			Skip 0 or 1	Compared Registers			
0	0	1	S	RS	RS	RT	RT

- **If RS != RT, then the next instruction gets executed normally.**
- **If RS = RT**
  - **Next instruction is skipped if S = 0**
  - **Next 2 instructions are skipped if S = 1**

Below is the isolated datapath for the compare instruction.



The functionality of our compare operation mainly comes from the 3 flip-flops that are shown above. These 3 flip-flops act as a 3 cycle buffer for the select line of the clock multiplexer. This multiplexer chooses either '0' or the main clock signal to send as the filtered clock signal to the register file. If we want to skip the next instruction, we can insert a '0' into the flip-flop buffer using the flip-flop's reset feature. Then, on the next clock cycle, the select line of the mux will be '0' and the register file clock will receive '0'. If S = '1' , then zeros are inserted at the 2 flip-flops on the left, causing the next 2 instructions to be skipped. Notice that the flip-flop on the far left receives a constant input of '1', which cleans up the zeros introduced in the buffer.

The other components, besides the 3 flip flops, all serve as control logic for resetting flip-flops. The comparator checks if the registers are equal. The "and" gate in the bottom left receives the "S" parameter, and controls whether or not the flip-flop on the far left is reset or not. The D-latches ensure that the flip-flops can only be reset when the main clock is high. This avoids prematurely resetting a flip-flop before they activate on the next

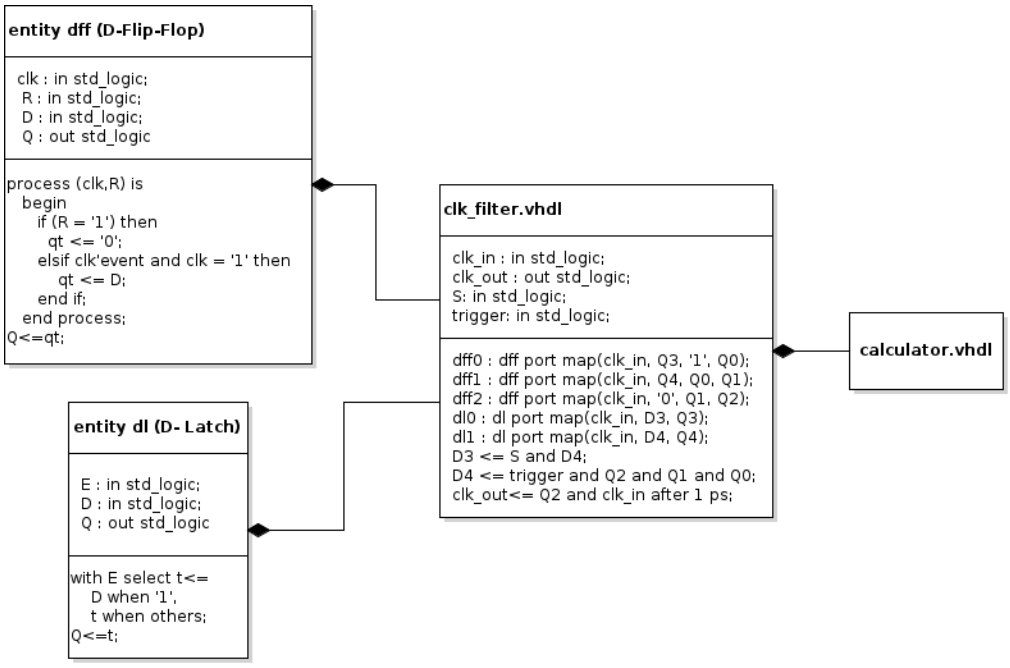
rising edge. The “and” of the three flip-flop outputs ensures that the flip-flops can only be modified if they are currently in a state of “111”. If they are not in a state of “111”, this means that the datapath is still in the process of skipping instructions, and so no other instruction should affect the datapath. This “and” of the three flip-flops also serves as a feedback loop into the reset signal. Once a flip flop is reset, its reset signal immediately goes back to ‘0’, and the flip-flop cannot be reset again until the flip-flops are back in the “111” state.

We chose to implement the compare datapath in this way because we thought it was the simplest method. We also could have used a counter to count the skipped instructions, which would have been about equivalent in terms of complexity. Also, our logic might have been simplified if we used flip-flops with synchronous resets instead of being asynchronous.

When implementing this datapath in VHDL, we created D-flip-flop components and D-latch components, and then connected them together in a structural design exactly as shown in the schematic above. All of this was inside “clk_filter” component that takes in the normal clock and outputs the filtered clock signal. The comparator implementation was done by XNORing the bits of each register and then ANDing the XNOR results together. If the registers are equal, then this operation will produce a value of ‘1’. Below is a portion of VHDL code that does this operation:

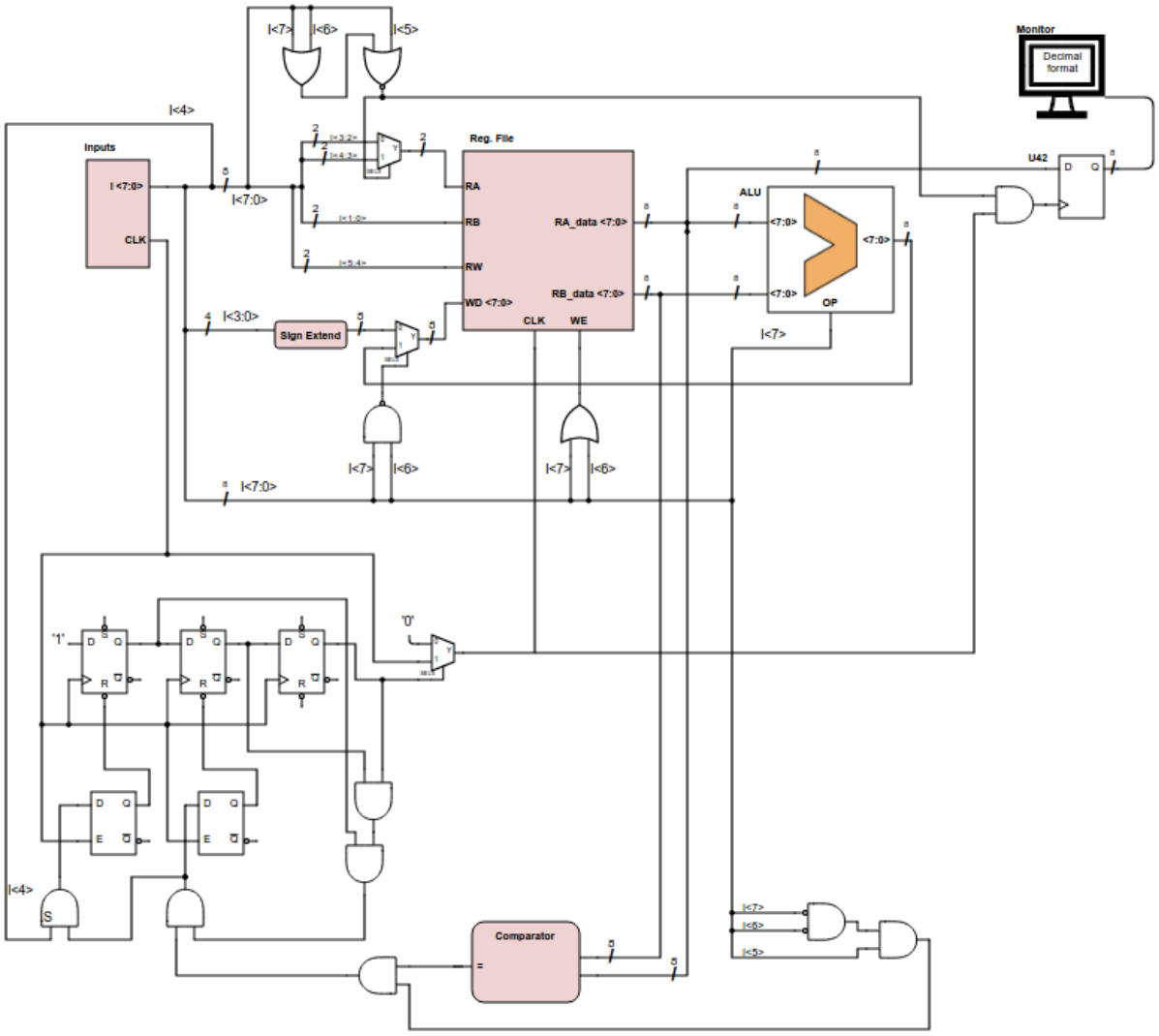
```
cmp_out <= (RA_data(7) xnor RB_data(7)) and
           (RA_data(6) xnor RB_data(6)) and …;
```

Shown below is the hierarchy used to implement the clock filter function.



Combined Datapath:

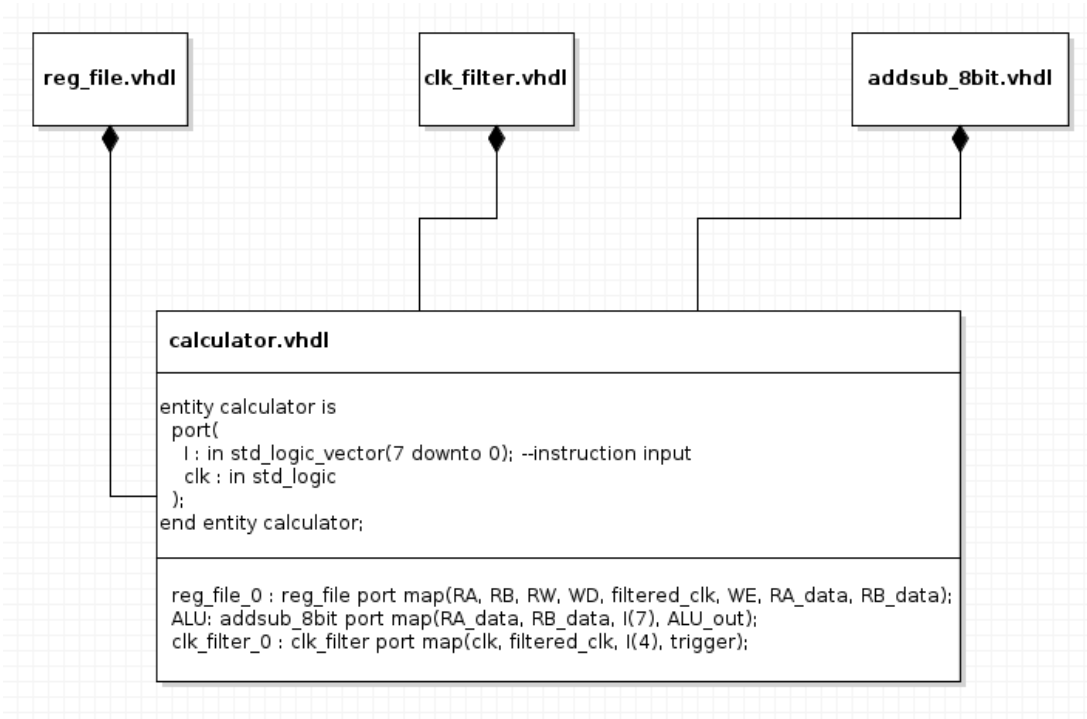
To build the combined datapath, we added each section one by one. We reused duplicate components and introduced control signals and multiplexers when necessary. The finalized datapath is shown below.



Some notes about the control signals:

- Control signals and logic (e.g.: muxs) were added wherever there was a conflict between 2 instructions and a resource. These control units also prevent false operations during any given instruction.
- The select line to the mux at the RA input of the register file is '1' only if the instruction is a print instruction. In this case, RA needs to be bits 4 and 3. Otherwise, RA is bits 3 and 2 of the instruction.
- The select line to the mux at the WD input chooses the sign extended immediate value if it is a load immediate instruction. Otherwise, it chooses the ALU output.
- The comparator is implemented using a series of XNOR gates on each bit and then ANDs each together. The output of the comparator ANDed with a compare instruction op-code serves as the triggering input to the clk_filter. The triggering input creates a skip for the next 1 or 2 instructions.
- The write enable (WE) input to the register file is 1 if either I<7> or I<6> is '1'. This covers load, add, and subtract, but excludes print and compare as it should. This prevents incorrect writes to registers.
- The added flip-flop to the display (top right) ensures that nothing will be printed if an instruction is currently being skipped. This is because the filtered clock ANDed with the display instruction op-code acts as the clock signal of the display flip-flop.

The implementation in VHDL used the following hierarchical structure:



As evident in the diagram above, we made heavy use of a structural hierarchy to combine each datapath into one unit. Our main strategy for implementation was to create a component if a resource was being used by multiple instructions. Simple operations that were only needed by one instruction (e.g.: sign extension and comparison) were implemented using CSAs. This decision was made to balance the level of abstraction while maintaining code clarity. We did not want to create so many levels of components that the logic became hard to follow.

Results (Testing and Verification):

Once our entire datapath was implemented in VHDL, we created a test bench file to drive instructions and a clock signal to the calculator. The code for the test bench can be seen in Appendix III. The test bench reads a sequence of binary instructions, which can also be found (along with their translations and expected results) in Appendix III. The methodology we used in our testing was to have at least one instruction of each type interact with another instruction. For example, we used a load immediate and print instruction right next to each other so that the print instruction would show the register was correctly loaded with the immediate value. The load immediate and print instructions were also interleaved with the add and subtract instructions to verify that immediate values could be loaded, used in an arithmetic operation, and then have the expected value printed out. Care was taken so that at least one type of every possible add or subtract operation was done. For instance: (+#) + (+#) = +#, (+#) + (-#) = +#, (+#) + (-#) = -#, …, (-#) - (-#) = (-#), et cetera.

We also tested the compare instruction in a similar manner. We interleaved compare instructions (which resulted in either 0, 1, or 2 instruction skips) with arithmetic, load immediate, print, and other compare instructions to verify that no changes were made to the registers. By doing this, we saw that the register contents were unchanged after an instruction was skipped, which verified that the control signals prevented the skipped instruction(s) from having any effect on the registers. Overall, the sequence of test instructions was constructed such that every type of instruction was shown to correctly produce an expected output by itself and that every possible interaction between instructions also produced an expected output without any conflicts. The expected outputs (shown in Appendix III) correctly matched up with the output of our calculator, which means that every instruction works correctly and properly interacts with one another. Based on these results, we concluded that our implementation of the ISA is correct.

Conclusion:

Overall, we were successful in implementing our 8 bit ISA and datapath in VHDL. Using our ISA design from lab 1, we were able to implement each instruction in the datapath. From there, we translated each circuit component into VHDL to create the finished calculator. A strength of our design is the simple ISA, as it had little variance in instruction organization. This allowed our control logic to be fairly simple as we only needed a few multiplexers and were able to reuse data lines. Another strength of the design is that it heavily relied on a structural hierarchy, which reduced the size of our code and made debugging easier. One weakness of this design is that when instructions are skipped, we simply block it from the rest of the datapath. This wastes the cycle as no work is done. Creating of the clock filtering logic was also the area that gave us the most difficulty. If given more time and resources, we could implement a PC counter and instruction memory. This would allow us to increment the PC counter in order to skip instructions, instead of simply blocking the clock signal and wasting cycles. Upon testing our calculator, we found no errors in functionality and that our design fully and correctly implemented our 8 bit ISA.

Appendix I:

Listed below is the work hour notebooks for each team member:

Abraham McIlvaine:

- Datapath schematic design: 10 hours
- VHDL implementation and verification: 7 hours
- Report writing: 6 hours
- **Total time: 23 hours**

Benjamin Steenkamer:

- Datapath schematic design: 9 hours
- VHDL implementation and verification: 7 hours
- Report writing: 8 hours
- **Total time: 24 hours**

Appendix II:

On the following pages, the source code for each VHDL module used in this project is given.

```
-- Benjamin Steenkamer and Abraham McIlvaine
-- CPEG 324-010
-- Lab 3: Single Cycle Calculator in VHDL- addsub_8bit.vhdl
-- 5/3/17

--8 Bit Adder/Subtractor--------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity addsub_8bit is
    port(input_a, input_b : in std_logic_vector(7 downto 0);
        addsub_sel : in std_logic; --0 = addition, 1 is subtraction.
        sum : out std_logic_vector(7 downto 0));
end entity addsub_8bit;

architecture structural of addsub_8bit is
component adder_8bit is
    port(input_a, input_b : in std_logic_vector(7 downto 0);
         sum : out std_logic_vector(7 downto 0));
end component adder_8bit;

signal second_term, inverted_second_term, negative_second_term : std_logic_vector(7
downto 0);
constant one : std_logic_vector(7 downto 0) := "00000001";
begin
adder_8bit_0: adder_8bit port map(input_a, second_term, sum); --Preform the addition
adder_8bit_1: adder_8bit port map(inverted_second_term, one, negative_second_term); --
Used for flipping sign of second term.

inverted_second_term <= not(input_b);

with addsub_sel select second_term <=
    input_b when '0', --Use regular term when adding
    negative_second_term when others; --Use negative second term when subtracting
becasue A - B = A + (-B)

end architecture structural;
-----------------------------------------------------------

--8 Bit Adder--------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity adder_8bit is
    port(input_a, input_b : in std_logic_vector(7 downto 0);
         sum : out std_logic_vector(7 downto 0));
end entity adder_8bit;

architecture structural of adder_8bit is
component full_adder is
    port(a, b, c_in : in std_logic;
        sum, c_out : out std_logic);
end component full_adder;

signal c0, c1, c2, c3,c4,c5,c6 : std_logic;
begin                       --(a(in), b(in), c_in(in), sum(out), c_out(out))
    fa0: full_adder port map(input_a(0), input_b(0),'0', sum(0), c0); --c_in for the
first full_adder is always 0, i.e. nothing.
    fa1: full_adder port map(input_a(1), input_b(1), c0, sum(1), c1);
    fa2: full_adder port map(input_a(2), input_b(2), c1, sum(2), c2);
    fa3: full_adder port map(input_a(3), input_b(3), c2, sum(3), c3);
    fa4: full_adder port map(input_a(4), input_b(4), c3, sum(4), c4);
    fa5: full_adder port map(input_a(5), input_b(5), c4, sum(5), c5);
    fa6: full_adder port map(input_a(6), input_b(6), c5, sum(6), c6);
```

```
    fa7: full_adder port map(input_a(7), input_b(7), c6, sum(7), open);

end architecture structural;
-------------------------------------------------------

--Full Adder-----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
    port(a, b, c_in : in std_logic;
        sum, c_out : out std_logic);
end entity full_adder;

architecture structural of full_adder is
component half_adder is
    port(a, b : in std_logic;
        sum, carry : out std_logic);
end component half_adder;

signal s1, s2, s3 : std_logic;
begin                    --(a(in), b(in), sum(out), c_out(out))
    h1: half_adder port map(a, b, s1, s3);
    h2: half_adder port map(s1, c_in, sum, s2);
    c_out <= s2 or s3;
end architecture structural;
-------------------------------------------------------

--Half Adder-----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
    port(a, b : in std_logic;
        sum, carry : out std_logic);
end entity half_adder;

architecture behavioral of half_adder is
begin
    sum <= a xor b;
    carry <= a and b;
end architecture behavioral;
-------------------------------------------------------
```

```
-- Benjamin Steenkamer and Abraham McIlvaine
-- CPEG 324-010
-- Lab 3: Single Cycle Calculator in VHDL - clk_filter.vhdl
-- 5/3/17

library ieee;
use ieee.std_logic_1164.all;

--Used to skip instructions.
entity clk_filter is
  port(
    clk_in : in std_logic;
    clk_out : out std_logic;
    S: in std_logic;
    trigger: in std_logic
  );
end entity clk_filter;

architecture structural of clk_filter is
  component dff is
     port(
      clk : in std_logic;
      R : in std_logic;
      D : in std_logic;
      Q : out std_logic
     );
  end component dff;
  component dl is
     port(
      E : in std_logic;
      D : in std_logic;
      Q : out std_logic
     );
  end component dl;

signal Q0 : std_logic := '1';
signal Q1, Q2, Q4, D3, D4 : std_logic := '1';
signal Q3 : std_logic := '1';
begin
  dff0 : dff port map(clk_in, Q3, '1', Q0);
  dff1 : dff port map(clk_in, Q4, Q0, Q1);
  dff2 : dff port map(clk_in, '0', Q1, Q2);
  dl0 : dl port map(clk_in, D3, Q3);
  dl1 : dl port map(clk_in, D4, Q4);

  D3 <= S and D4;
  D4 <= trigger and Q2 and Q1 and Q0;
  clk_out <= Q2 and clk_in after 1 ps;
end architecture structural;

library ieee;
use ieee.std_logic_1164.all;
--D Flip-Flop
entity dff is
   port(
      clk : in std_logic;
      R : in std_logic;
      D : in std_logic;
      Q : out std_logic
   );
end entity dff;
architecture behavioral of dff is
```

```
  signal qt : std_logic :='1';
begin
   process (clk,R) is
   begin
      if (R = '1') then
        qt <= '0';
      elsif clk'event and clk = '1' then
          qt <= D;
      end if;
   end process;
   Q<=qt;
end architecture behavioral;

library ieee;
use ieee.std_logic_1164.all;
--D latch
entity dl is
   port(
      E : in std_logic;
      D : in std_logic;
      Q : out std_logic
   );
end entity dl;
architecture behavioral of dl is
  signal t : std_logic := '0';
begin
   with E select t<=
    D when '1',
    t when others;
   Q<=t;
end architecture behavioral;
```

```vhdl
-- Benjamin Steenkamer and Abraham McIlvaine
-- CPEG 324-010
-- Lab 3: Single Cycle Calculator in VHDL - reg_file.vhdl
-- 5/3/17

library ieee;
use ieee.std_logic_1164.all;
--Contains 4, 8 bit registers that are initialized to 0.
entity reg_file is
  port(
    RA : in std_logic_vector(1 downto 0);
    RB : in std_logic_vector(1 downto 0);
    RW : in std_logic_vector(1 downto 0);
    WD : in std_logic_vector(7 downto 0);
    CLK : in std_logic;
    WE : in std_logic;
    RA_data : out std_logic_vector(7 downto 0);
    RB_data : out std_logic_vector(7 downto 0)
  );
end entity reg_file;

architecture behavioral of reg_file is
  signal R0 : std_logic_vector(7 downto 0) := "00000000";
  signal R1 : std_logic_vector(7 downto 0) := "00000000";
  signal R2 : std_logic_vector(7 downto 0) := "00000000";
  signal R3 : std_logic_vector(7 downto 0) := "00000000";

  begin
    --Reads are combinational.
    with RA select RA_data <=
      R0 when "00",
      R1 when "01",
      R2 when "10",
      R3 when others;
    with RB select RB_data <=
      R0 when "00",
      R1 when "01",
      R2 when "10",
      R3 when others;

    --Writes only happen on the rising edge of the clock.
    process (CLK) is
      begin
        if (CLK'event and CLK='1') then
          if (WE = '1') then
            if (RW = "00") then
              R0 <= WD;
            elsif (RW = "01") then
              R1 <= WD;
            elsif (RW = "10") then
              R2 <= WD;
            elsif (RW = "11") then
              R3 <= WD;
            end if;
          end if;
        end if;
      end process;
    end architecture;
```

```
-- Benjamin Steenkamer and Abraham McIlvaine
-- CPEG 324-010
-- Lab 3: Single Cycle Calculator in VHDL - calculator.vhdl
-- 5/3/17
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity calculator is
  port(
    I : in std_logic_vector(7 downto 0); --instruction input
    clk : in std_logic
  );
end entity calculator;


architecture structural of calculator is
  component addsub_8bit is
      port(
          input_a, input_b : in std_logic_vector(7 downto 0);
          addsub_sel : in std_logic; --0 = addition, 1 is subtraction.
          sum : out std_logic_vector(7 downto 0)
        );
  end component addsub_8bit;

  component reg_file is
    port(
      RA : in std_logic_vector(1 downto 0);
      RB : in std_logic_vector(1 downto 0);
      RW : in std_logic_vector(1 downto 0);
      WD : in std_logic_vector(7 downto 0);
      CLK : in std_logic;
      WE : in std_logic;
      RA_data : out std_logic_vector(7 downto 0);
      RB_data : out std_logic_vector(7 downto 0)
    );
  end component reg_file;

  component clk_filter is
    port(
      clk_in : in std_logic;
      clk_out : out std_logic;
      S: in std_logic;
      trigger: in std_logic
    );
  end component clk_filter;

signal filtered_clk, WE, display, WD_sel, trigger, cmp_out : std_logic;
signal RA, RB, RW : std_logic_vector(1 downto 0);
signal WD, RA_data, RB_data, sign_ext_imm, ALU_out: std_logic_vector(7 downto 0);

begin
  reg_file_0 : reg_file port map(RA, RB, RW, WD, filtered_clk, WE, RA_data, RB_data);
  ALU: addsub_8bit port map(RA_data, RB_data, I(7), ALU_out);
  clk_filter_0 : clk_filter port map(clk, filtered_clk, I(4), trigger);

  --Assign and/or create control singals route instructions.
  --See datapath schematic.
  RB <= I(1 downto 0);
  RW <= I(5 downto 4);

  --Print regesiter control signal.
```

```
  display <= not (I(7) or I(6) or I(5));

  with display select RA <=
    I(3 downto 2) when '0',
    I(4 downto 3) when others;

  --Sign extedend the immediate value.
  sign_ext_imm(3 downto 0) <= I(3 downto 0);
  with I(3) select sign_ext_imm(7 downto 4) <=
    "1111" when '1',
    "0000" when others;

  --Select whether signed extedend immediate value or ALU results to written to RW.
  WD_sel <= not(I(7) and I(6));
  with WD_sel select WD <=
    sign_ext_imm when '0',
    ALU_out when others;

  --Is WD written to RW?
  WE <= I(7) or I(6);

  --Used to trigger the skip instruction logic.
  trigger <= (not I(7)) and (not I(6)) and I(5) and cmp_out;

  --Compare the value of the two registers; cmp_out is 1 if they are equal.
  cmp_out <= (RA_data(7) xnor RB_data(7)) and
             (RA_data(6) xnor RB_data(6)) and
             (RA_data(5) xnor RB_data(5)) and
             (RA_data(4) xnor RB_data(4)) and
             (RA_data(3) xnor RB_data(3)) and
             (RA_data(2) xnor RB_data(2)) and
             (RA_data(1) xnor RB_data(1)) and
             (RA_data(0) xnor RB_data(0));

  --The display function
  process(filtered_clk,display) is
    variable int_val : integer;
    begin
      if((filtered_clk'event and filtered_clk = '1') and (display = '1')) then
        int_val := to_integer(signed(RA_data));

        --Make the ouput right aligned.
        if(int_val >= 0) then
          if(int_val < 10) then
            report "   " & integer'image(int_val) severity note;
          elsif(int_val < 100) then
            report "  " & integer'image(int_val) severity note;
          else
            report " " & integer'image(int_val) severity note;
          end if;
        else --Display value is negative
          if(int_val > -10) then
            report "  " & integer'image(int_val) severity note;
          elsif(int_val > -100) then
            report " " & integer'image(int_val) severity note;
          else
            report integer'image(int_val) severity note;
          end if;
        end if;
      end if;
  end process;
end architecture structural;
```

Appendix III:

On the following pages, the source code for the VHDL test bench is given, along with the test instructions read in by the test bench.

```
-- Benjamin Steenkamer and Abraham McIlvaine
-- CPEG 324-010
-- Lab 3: Single Cycle Calculator in VHDL - calculator_tb.vhdl
-- 5/3/17

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

--The test bench has no interface.
entity calculator_tb is
end entity calculator_tb;

architecture structural of calculator_tb is

component calculator is
  port(
    I : in std_logic_vector(7 downto 0); --instruction input
    clk : in std_logic
  );
end component calculator;

signal I : std_logic_vector(7 downto 0);
signal clk : std_logic;

begin
    calculator_0 : calculator port map(I, clk);

    process
      file instruction_file : text is in "instructions.txt"; --Instructions in text
(ASCII) file.
      variable instruction_line : line;
      variable intruction_vector : bit_vector(7 downto 0);
    begin
      wait for 999 ps; --Used to offset a wait delay in the calculator so time stamps
in GTKwave look nicer.
      while (not(endfile(instruction_file))) loop --Loop to the end of the text file.
        clk <= '0';

        readline(instruction_file, instruction_line); --Read in instruction line from
file
        read(instruction_line, intruction_vector); --Pass instruction to bit vector
form
        I <= to_stdlogicvector(intruction_vector); --Convert bit vector to
std_logic_vector and pass instruction to the calculator input.

        --Create a rising edge for the clock.
        wait for 1 ns;
        clk <= '1';
        wait for 1 ns;
      end loop;
      wait;
    end process;
end architecture structural;
```

Test Bench Instructions: The instructions in the left column are entered into "instructions.txt" (found in the source code zip file) and are read in by the VHDL test bench. Next to each instruction is the translated meaning and the expected value if there is a printout.

00000000	Print contents of R0; **Printout is :" 0"**; R0 is initialized to '0'. (Print tests.)
00001000	Print contents of R1; **Printout is: " 0"**; R1 is initialized to '0'.
00010000	Print contents of R2; **Printout is: " 0"**; R2 is initialized to '0'.
00011000	Print contents of R3; **Printout is: " 0"**; R3 is initialized to '0'.
11000001	Load immediate value of '1' into R0. (Load immediate tests.)
00000000	Print contents of R0; **Printout is: " 1"**;
11010111	Load immediate value of '7' into R1.
00001000	Print contents of R1; **Printout is: " 7"**;
11101000	Load immediate value of '-8' into R2.
00010000	Print contents of R2; **Printout is: " -8"**;
01110001	R3 = R0 + R1; (Simple add/sub tests.)
00011000	Print contents of R3; **Printout is: " 8"**; 1 + 7 = 8.
01111111	R3 = R3 + R3;
00011000	Print contents of R3; **Printout is: " 16"**; 8 + 8 =16.
01111110	R3 = R3 + R2;
00011000	Print contents of R3; **Printout is: " 8"**; 16 + -8 = 8.
01111110	R3 = R3 + R2;
01111110	R3 = R3 + R2;
00011000	Print contents of R3; **Printout is: " -8"**; 8 + -8 + -8 = -8.
10110100	R3 = R1 – R0;
00011000	Print contents of R3; **Printout is: " 6"**; 7 – 1 = 6.
11000111	Load immediate value of '7' into R0.
10111100	R3 = R3 – R0;
00011000	Print contents of R3; **Printout is: " -1"**; 6 – 7 = -1.
00011001	Print contents of R3; **Printout is: " -1"**; Bottom three instruction bits are "don't care."
00011010	Print contents of R3; **Printout is: " -1"**; Bottom three instruction bits are "don't care."
00011100	Print contents of R3; **Printout is: " -1"**; Bottom three instruction bits are "don't care."
00101100	Skip 1 instruction if R3 = R0; They are not equal, so no instruction is skipped. (Start skip instruction tests.)
00000000	Print contents of R0; **Printout is :" 7"**.
00111100	Skip 2 instructions if R3 = R0; They are not equal, so no instructions are skipped.
11000001	Load immediate value of '1' into R0.

11110001	Load immediate value of '1' into R3.
00000000	Print contents of R0; **Printout is :" 1"**.
00011000	Print contents of R3; **Printout is :" 1"**.
00101100	Skip 1 instruction if R3 = R0; They are equal, so next instruction is skipped.
00000000	Print contents of R0; Nothing will be printed since it is skipped.
00101100	Skip 1 instruction if R3 = R0; They are equal, so next instruction is skipped.
01111110	R3 = R3 + R2; This instruction will be skipped.
00011000	Print contents of R3; **Printout is :" 1"**. R3's value was not changed because of the skip.
00101100	Skip 1 instruction if R3 = R0; They are equal, so next instruction is skipped.
10111110	R3 = R3 - R2; This instruction will be skipped.
00011000	Print contents of R3; **Printout is :" 1"**. R3's value was not changed because of the skip.
00101100	Skip 1 instruction if R3 = R0; They are equal, so next instruction is skipped.
00101100	Skip 1 instruction if R3 = R0; This instruction will be skipped.
11011111	Load immediate value of '-1' into R1.
00001000	Print contents of R1; **Printout is :" -1"**. Last skip instruction was successfully skipped.
00111100	Skip 2 instructions if R3 = R0; They are equal, so next 2 instructions are skipped.
00000000	Print contents of R0; Nothing will be printed since it is skipped.
01110101	R3 = R1 + R1; This instruction will be skipped.
00011000	Print contents of R3; **Printout is :" 1"**. R3's value was not changed because of the skip.
00111100	Skip 2 instructions if R3 = R0; They are equal, so next 2 instructions are skipped.
10110111	R3 = R1 - R3; This instruction will be skipped.
00111100	Skip 2 instructions if R3 = R0; This instruction will be skipped.
00011000	Print contents of R3; **Printout is :" 1"**. R3's value was not changed because of the skip.
11000000	Load immediate value of '0' into R0. (Starting more thorough add/sub tests now.)
11010000	Load immediate value of '0' into R1.
11100000	Load immediate value of '0' into R2.
11110000	Load immediate value of '0' into R3.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :" 0"**. 0 + 0 = 0.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :" 0"**. 0 – 0 = 0.
11000001	Load immediate value of '1' into R0.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :" 1"**. 1 + 0 = 1.

11000010	Load immediate value of ‘2’ into R0.
11010101	Load immediate value of ‘5’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ 7”**. 5 + 2 = 7.
01101010	R2 = R2 + R2; (R2 = 14)
01101010	R2 = R2 + R2; (R2 = 28)
01101010	R2 = R2 + R2; (R2 = 56)
01101010	R2 = R2 + R2; (R2 = 112)
01101010	R2 = R2 + R2; (R2 = -32 [0b11100000], overflow has occurred.)
00010000	Print contents of R2; **Printout is :“ -32”**.
11001111	Load immediate value of ‘-1’ into R0.
11010010	Load immediate value of ‘2’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ 1”**. 2 + -1 = 1.
11001011	Load immediate value of ‘-5’ into R0.
11010100	Load immediate value of ‘4’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ -1”**. 4 + -5 = -1.
11001111	Load immediate value of ‘-1’ into R0.
11011001	Load immediate value of ‘-7’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ -8”**. -7 + -1 = -8.
11000011	Load immediate value of ‘3’ into R0.
11011110	Load immediate value of ‘-2’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ 1”**. -2 + 3 = 1.
11000010	Load immediate value of ‘2’ into R0.
11011010	Load immediate value of ‘-6’ into R1.
01100100	R2 = R1 + R0;
00010000	Print contents of R2; **Printout is :“ -4”**. -6 + 2 = -4.
11101000	Load immediate value of ‘-8’ into R2.
01101010	R2 = R2 + R2; (R2 = -16)
01101010	R2 = R2 + R2; (R2 = -32)
01101010	R2 = R2 + R2; (R2 = -64)

01101010	R2 = R2 + R2; (R2 = -128)
01111010	R3 = R2 + R2; (R2 = 0, underflow has occurred.)
00011000	Print contents of R3; **Printout is :“ 0”**.
11000001	Load immediate value of ‘1’ into R0.
10111000	R3 = R2 - R0 (R3 = 127 [0b01111111] underflow has occurred.)
00011000	Print contents of R3; **Printout is :“ 127”**.
11000001	Load immediate value of ‘1’ into R0.
11010010	Load immediate value of ‘2’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ 1”**.
11000011	Load immediate value of ‘3’ into R0.
11010101	Load immediate value of ‘5’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ 2”**.
11000111	Load immediate value of ‘7’ into R0.
11010110	Load immediate value of ‘6’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ -1”**.
11001110	Load immediate value of ‘-2’ into R0.
11010101	Load immediate value of ‘5’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ 7”**.
11000010	Load immediate value of ‘2’ into R0.
11011010	Load immediate value of ‘-6’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ -8”**.
11001011	Load immediate value of ‘-5’ into R0.
11011001	Load immediate value of ‘-7’ into R1.
10100100	R2 = R1 - R0;
00010000	Print contents of R2; **Printout is :“ -2”**.