Kaleb Frey & Justus Matteson

CPEG324-010

5/3/19

**Lab 3: A Single-Cycle Calculator in VHDL**

**Abstract:**

The goal of this lab was to build the VHDL model for the single-cycle calculator ISA that we had created in a previous lab. Alongside this overarching goal, we were also tasked with creating a fully functional testbench for the project that would verify that the design and implementation were working correctly. Through the process of completing the tasks we developed our VHDL coding skills while also gaining a better understanding of the single-cycle datapath system. Additionally, we began exploring the correct way to test the functionality of a program and obtained invaluable skills for any projects we may involve with moving forward.

Overall, we were able to successfully create the 8-bit calculator VHDL model. Our model can add or subtract two registers, load an immediate value into a register, compare two registers and skip instructions accordingly, and print out the value stored in a register. We accomplished this by first designing our overall datapath and then translating each component to VHDL before finally creating the overall process of the datapath.

**Division of Labor:**

This lab had four major areas that we needed to work on. Our first step was to create the circuit design of our single-cycle datapath that would implement all the necessary instructions. Following the design, we had to implement our model as a VHDL program. Once the VHDL program was completed we required a testbench that could verify the program worked correctly. Finally, we had to complete the lab report that summarized our process and findings.

**Single-Cycle Datapath Design:** Kaleb Frey

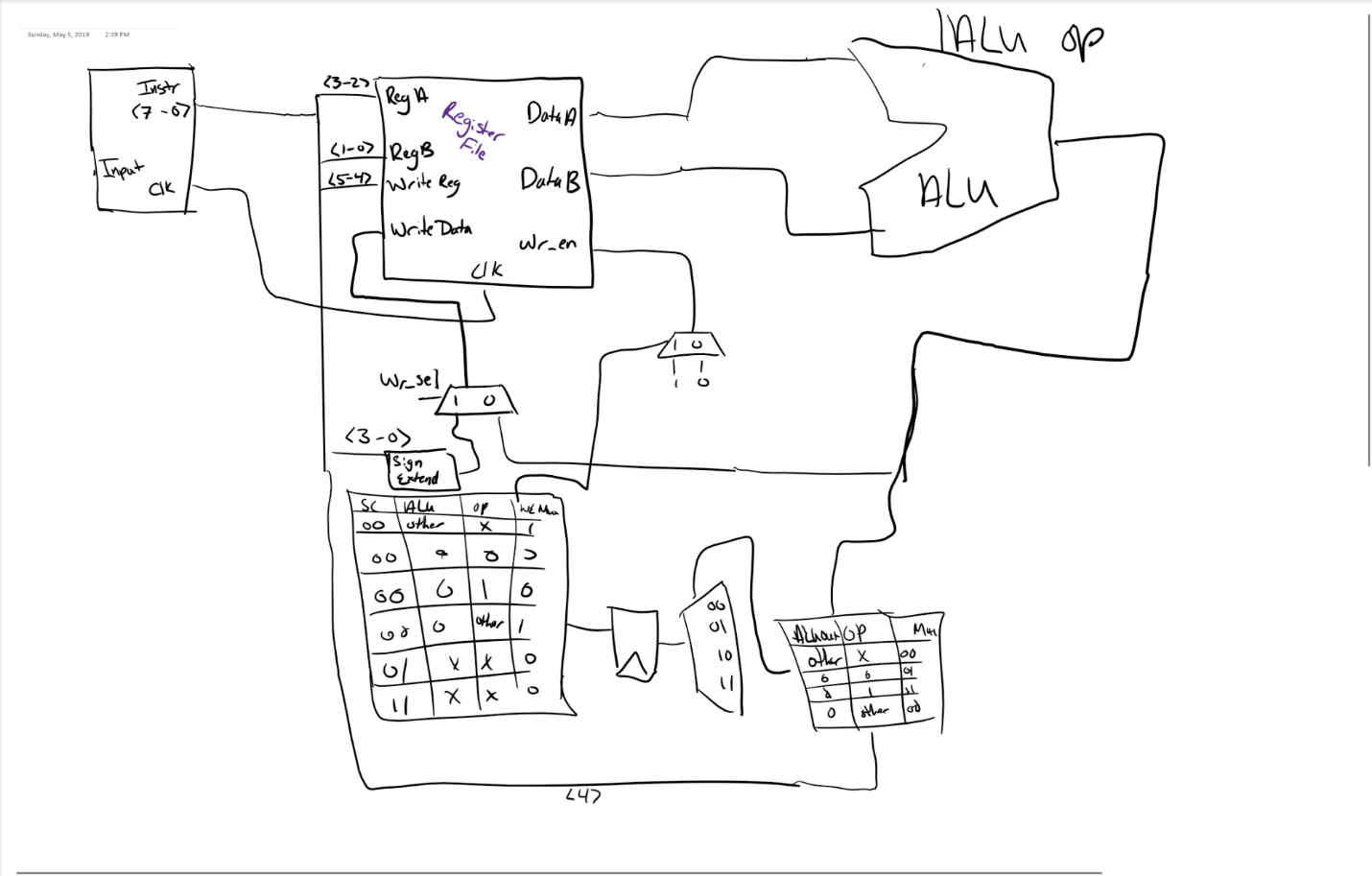
**VHDL Implementation:** Kaleb Frey and Justus Matteson

**VHDL Testbench:** Kaleb Frey

**Lab Report:** Kaleb Frey

**Detailed Strategy:**

We began by designing the single-cycle gate level circuit of our calculator, which involved a register file, controller, ALU, as well as many other muxes and gates for passing specific signals to the write data and other sections of the circuit. After making the necessary changes from our original design, the final circuit design can be seen below:



We began the process of transferring our gate-level circuit design to VHDL code by first tackling the two components that would be a basis of our code. When creating the ALU we used the code that we had previously made for the adder and subtractor in Lab 2; however, we made the necessary changes to provide fully functional code. For the register file component, we first implemented all the necessary inputs and outputs that would be required, and for the write data input we implemented a series of if statements that would only write to the desired register if the write enable signal was ‘1’.

After completing these components, we moved on to implementing the overall code of the calculator ISA itself. This was by far the most difficult part of the creating the VHDL model due to figuring out how the components interacted and making sure that everything worked correctly. All the instruction fetch was done in this section of the code and was determined through the opcode of each instruction. Determining how to appropriately go about skipping the instructions was also one of the hardest tasks of this lab.

Once we had all the code for our system done, we moved on the creating the testbench for the ISA. We accomplished this by having the testbench read each individual line of a text file and each individual bit on the lines. Our testbench would then send these bits to the main program of our system and the desired output would be produced. You can see the text file that we used, which lists the instructions that we used and the result of each instruction in Appendix C. We made a few changes to the testbench we had previously made in regards to the order of each instruction, and now each instruction should be accurately read and tested without any problems.

**Results:**

After completing the VHDL implementation of the design that we had fine-tuned at the beginning of the lab, we created a VHDL testbench that would verify whether the five instructions we were tasked with implementing worked correctly. The testbench used a test text file that had numerous binary levels instructions. In Appendix C at the bottom of this report you can find the testbench code alongside the description of what each section of lines should result with. We specifically tested the load immediate instruction first to make sure that the output matched what we were looking for. After verifying the load immediate instruction was functional, we moved on to making sure the add and subtract functions worked correctly using the print instruction and the values loaded in using the load immediate instruction. Finally, we tested the compare function with all the possible outcomes including: not skipping any lines, skipping 1 line, and skipping two lines. All the tests that we created produced the output that we anticipated and desired.

**Conclusion:**

We had to make a lot of changes to our ISA and code based on issues we had in our original plans for the calculator, but we believe that we were able to fix all the mistakes that we had made previously. The most difficult part of this lab was the concept of taking a circuit level design and creating a code model for it, since this was our first time ever completing a task of this depth. We struggled to make sure that all the components interacted correctly within the code, and if we had more time for the project, we would have added more functionality such as a reset “button” and made the current code easier to follow.

**Appendix A: Notebooks**

**Kaleb Frey:**

* Datapath design (presentation and final): 3 hours
* VHDL implementation: 22 hours
* Report writing: 5 hours

**Justus Matteson:**

* VHDL implementation: 4 hours

**Appendix B: VHDL code**

**RegisterFile:**

library ieee;

use ieee.std\_logic\_1164.all;

entity reg\_file is

port(

reg\_a : in std\_logic\_vector(1 downto 0);

reg\_b : in std\_logic\_vector(1 downto 0);

reg\_write : in std\_logic\_vector(1 downto 0);

write\_data : in std\_logic\_vector(7 downto 0);

clk : in std\_logic;

write\_enable : in std\_logic;

reg\_a\_data : out std\_logic\_vector(7 downto 0);

reg\_b\_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

with reg\_a select reg\_a\_data <=

R0 when "00",

R1 when "01",

R2 when "10",

R3 when others;

with reg\_b select reg\_b <=

R0 when "00",

R1 when "01",

R2 when "10",

R3 when others;

process (clk) is

begin

if (clk'event and clk='1') then

if (write\_enable = '1') then

if (reg\_write = "00") then

R0 <= write\_data;

elsif (reg\_write = "01") then

R1 <= write\_data;

elsif (reg\_write = "10") then

R2 <= write\_data;

elsif (reg\_write = "11") then

R3 <= write\_data;

end if;

end if;

end if;

end process;

end architecture;

**ALU:**

ibrary ieee;

use ieee.std\_logic\_1164.all;

entity ALU is

port(input\_a, input\_b : in std\_logic\_vector(7 downto 0);

addsub\_sel : in std\_logic;

sum : out std\_logic\_vector(7 downto 0));

end entity ALU;

architecture structural of ALU is

component ALU\_add is

port(input\_a, input\_b : in std\_logic\_vector(7 downto 0);

sum : out std\_logic\_vector(7 downto 0));

end component ALU\_add;

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

add0: ALU\_add port map(input\_a, second\_term, sum);

add1: ALU\_add port map(inverted\_second\_term, one, negative\_second\_term);

inverted\_second\_term <= not(input\_b);

with addsub\_sel select second\_term <=

input\_b when '0',

negative\_second\_term when others;

end architecture structural;

library ieee;

use ieee.std\_logic\_1164.all;

entity ALU\_add is

port(input\_a, input\_b : in std\_logic\_vector(7 downto 0);

sum : out std\_logic\_vector(7 downto 0));

end entity ALU\_add;

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

fa0: full\_adder port map(input\_a(0), input\_b(0),'0', sum(0), c0);

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);

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

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;

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;

**Calculator:**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity calculator is

port(

instr : in std\_logic\_vector(7 downto 0);

clk : in std\_logic

);

end entity calculator;

architecture structural of calculator is

component ALU is

port(

input\_a, input\_b : in std\_logic\_vector(7 downto 0);

addsub\_sel : in std\_logic;

sum : out std\_logic\_vector(7 downto 0)

);

end component ALU;

component reg\_file is

port(

reg\_a : in std\_logic\_vector(1 downto 0);

reg\_b : in std\_logic\_vector(1 downto 0);

reg\_write : in std\_logic\_vector(1 downto 0);

write\_data : in std\_logic\_vector(7 downto 0);

clk : in std\_logic;

write\_enable : in std\_logic;

reg\_a\_data : out std\_logic\_vector(7 downto 0);

reg\_b\_data : out std\_logic\_vector(7 downto 0)

);

end component reg\_file;

signal write\_enable, display, write\_data\_sel, skipcount : std\_logic;

signal reg\_a, reg\_b, reg\_write : std\_logic\_vector(1 downto 0);

signal write\_data, reg\_a\_data, reg\_b\_data, sign\_ext, ALU\_out: std\_logic\_vector(7 downto 0);

begin

if((instr(7) = '0') and (instr(6) = '0') and (instr(5) = '1")) then

skipcount <= instr(4);

end if;

if(ALU\_out = "00000000")

if(skipcount = '1') then

write\_enable <= '0';

elsif(skipcount = '0') then

write\_enable <= '0';

end if;

end if;

if(skipcount = '1' and write\_enable = '0') then

skipcount <= '0';

elsif(skipcount = '0' and write\_enable = '0') then

write\_enable <= '1';

end if;

reg\_file\_0 : reg\_file port map(reg\_a, reg\_b, reg\_write, write\_data, write\_enable, reg\_a\_data, reg\_b\_data);

ALU0 : ALU port map(reg\_a\_data, reg\_b\_data, instr(7), ALU\_out);

reg\_b <= instr(1 downto 0);

reg\_write <= instr(5 downto 4);

display <= not (instr(7) or instr(6) or instr(5));

with display select reg\_a <=

instr(3 downto 2) when '0',

instr(4 downto 3) when others;

sign\_ext(3 downto 0) <= instr(3 downto 0);

with instr(3) select sign\_ext(7 downto 4) <=

"1111" when '1',

"0000" when others;

write\_data\_sel <= not(instr(7) and instr(6));

with write\_data\_sel select write\_data <=

sign\_ext when '0',

ALU\_out when others;

process(display) is

variable int\_val : integer;

begin

if((clk'event and clk = '1') and (display = '1') and (write\_enable = '1') then

int\_val := to\_integer(signed(reg\_a\_data));

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

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;

**Testbench:**

library ieee;

use ieee.std\_logic\_1164.all;

use std.textio.all;

entity calculator\_tb is

end entity calculator\_tb;

architecture structural of calculator\_tb is

component calculator is

port(

instr : in std\_logic\_vector(7 downto 0);

clk : in std\_logic

);

end component calculator;

signal instr : std\_logic\_vector(7 downto 0);

signal clk : std\_logic;

begin

calculator\_0 : calculator port map(instr, clk);

process

file instruction\_file : text is in "test.txt";

variable instruction\_line : line;

variable intruction\_vector : bit\_vector(7 downto 0);

begin

wait for 999 ps;

while (not(endfile(instruction\_file))) loop

clk <= '0';

readline(instruction\_file, instruction\_line);

read(instruction\_line, intruction\_vector);

instr <= to\_stdlogicvector(intruction\_vector);

wait for 1 ns;

clk <= '1';

wait for 1 ns;

end loop;

wait;

end process;

end architecture structural;

**Appendix C: Testing**

00000000 --prints R0 which contains 0

00001000 --prints R1 which contains 0

11010101 --loads 5 into R1

00001000 --prints R1 which contains 5

11101000 --loads 8 into R2

01111001 --adds R1 and R2 and stores in R3

00011000 --prints R3 which contains 13

10111001 --subtracts R1 from R2 and stores in R3

00011000 --prints R3 which contains 3

11010010 --loads 2 into R1

11100010 --loads 2 into R2

00101001 --compares R1 and R2 and skips 1 line

00000000 --this line is skipped

00100001 --compares R0 and R1 and does not skip

00001000 --prints R1 which contains 2

00110110 --compares R2 and R1 and skips 2 lines

00001000 --this line is skipped

00010000 --this line is skipped

00011000 --prints R3 which contains 3