Shane Cincotta

CPEG324

LAB 3

5/7/19

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

**Abstract:**

The main goal of this lab was to implement my 8-Bit calculator ISA into a single cycle calculator in VHDL, as well as developing a testbench for the calculator. The 8-Bit ISA was previously designed in lab 1. This lab reinforces various skills such as, understanding data paths for a single cycle CPU, combining multiple data paths, developing schematic at the register transfer level, implementing the schematic in VHDL, as well as developing a testbench in VHDL to test the schematic.

Using these skills, I was able to successfully implement my 8-Bit ISA design into a single cycle CPU. The calculator was created by first creating the data paths for each individual component (such as add, subtract, load immediate, etc) then the data paths were combined into a complete data path. This data path was then translated into a schematic, which was then translated to VHDL code.

**Division of Labor:**

I do not have a partner for this class, as such I did the lab myself.

**Detailed Strategy:**

Before I could begin, I first had to remake my ISA design from lab 1, specifically the opcode portion. I had a conflict between the op code for my print and add commands. My new table of op codes is:

**ISA Design**

|  |  |
| --- | --- |
| Op-code | Instruction |
| 01 | Add |
| 10 | Sub |
| 11 | LI |
| 001 | Compare |
| 000 | Print |

Add:

7 6 5 4 3 2 1

|  |  |  |  |
| --- | --- | --- | --- |
| Op-code | rd | rs | rt |

2 2 2 2

**Format:** Add rd, rs, rt

**Purpose:** To add 2-bit integers

**Description:** rd = rs + rt

The 8-bit value in register *rs* is added to the 8-bit value in register *rt* to produce a 8-bit result.

* If the addition results in an integer overflow, the destination register is not modified, an exception occurs.
* If the addition does not result in an overflow, the 8-bit result is placed into register *rd*.

Sub:

7 6 5 4 3 2 1

|  |  |  |  |
| --- | --- | --- | --- |
| Op-code | rd | rs | rt |

2 2 2 2

**Format:** Sub rd, rs, rt

**Purpose:** To subtract 8-bit integers

**Description:** rd = rs – rt

The 8-bit value in register *rt* is subtracted from the 8-bit value in register *rs* to produce an 8-bit result.

* If the subtraction results in an integer overflow, the destination register is not modified, an exception occurs.
* If the subtraction does not result in an overflow, the 8-bit result is placed into register *rd*.

Data Path for Add/Sub Commands

A screenshot of a cell phone

Description automatically generated

Registers RS comes from bits 3/2, RT comes from bits 1/0 and RD comes from bits 5/4. The opcode is sent to the ALU, this determines which command to run (either addition or subtraction). The output of the ALU is the sent to the “Write Data” input of the reg file. The corresponding VHDL files are *“add-sub.vhdl”* (Fig 1) and *“regFile.vhdl”* (Fig 2). *Add-sub.vhdl* implements the logic of the ALU, while *regFile.vhdl* handles the logic of the register file. The ALU was made using 8 full adders (each comprised of 2 half adders).

LI:

7 6 5 4 3 0

|  |  |  |
| --- | --- | --- |
| Op-code | rd | Signed Immediate |

2 2 4

**Format:** LI rd, immediate

**Purpose:** To load an 4-bit immediate value into an 8-bit register

**Description:** rd = rd + immediate

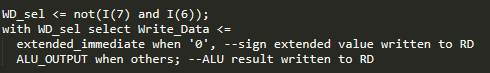
* A 4-bit *immediate* value is sign extended and placed in the 8-bit register *rd*. The sign extension is performed by using *ori* which puts a constant in the least significant bits of *rd*.
* If an overflow occurs, an exception is thrown

Data Path for LI Command

A screenshot of a cell phone

Description automatically generated

Register RD comes from bits 5/4 and the immediate value comes from bits 3-0. The immediate value is sign extended; this value is then sent to the Write Data input port in the reg file. The VHDL code for the reg file is the same as used in the add/sub commands. The sign extension was preformed by using a with/select statement.



Compare:

7 5 4 4 3 2 1

|  |  |  |  |
| --- | --- | --- | --- |
| Op-code | Skip (0 Skip 1, 1 skip 2) | rs | rt |

3 1 2 2

**Format:** compare rs, rt

**Purpose:** Compare two registers. If they are not equal, execute the next instruction. If equal, the choice exists of either skipping either the next 1 or the next 2 instructions. If RS != RT, no instructions are skipped. If RS=RT, 1 instruction is skipped if S=1, else 2 instructions are skipped.

**Description:**

Compares the result of using instruction *and* on registers *rs* and *rt*

Data Path for Compare Command

A close up of a logo

Description automatically generated

The compare operation is made possible due to the 3 flip-flops as shown in the schematic. The flip flops act as a buffer to the clock mux. If the next instruction is to be skipped, a 0 is inserted into the flip-flop. If the value of *S* is 1, then 2 zeroes are inserted to the farthest left flip-flops. This will lead to the next 2 instructions being skipped. The purpose of the D-latches is to make sure that the flip-flops are only reset when the clock signal is 1. The actual comparison was done by using the *xnor* command on the bits of the RS and RT registers

A screenshot of a computer

Description automatically generated

Print:

7 5 4 3 2 0

|  |  |  |
| --- | --- | --- |
| Op-code | rs | Don’t Care |

3 2 3 **Format:**print rs

**Purpose:** Display a registers content to the console.

**Description:**

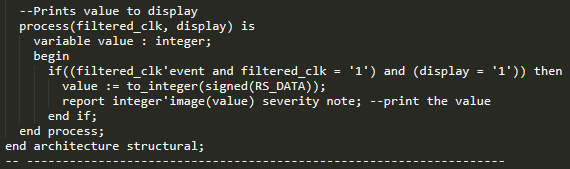
Checks the 8-bit value stored in register *rs* and prints the value to the console

Data Path for Print Command

A screenshot of a cell phone

Description automatically generated

This command prints the conents of register RS, which is given by bits 3/2. The value is converted to an integer and then printed using the “report” command within *Calc.vhdl* (Fig 4).

****

**Results**

Once the data path was implemented in VHDL, a testbench was made to test different combinations of instructions. The testbench includes a series of instructions for the calculator (in binary). Multiple instructions are tested in sequence with each other for error testing. What this means is that, for example, a LI instruction may be ran, and then following that an add command may be ran. This is repeated many times with different combinations of commands. This is to test how the commands interact with each other, and to see if there are any conflicts between the instructions. When I ran my testbench, I noticed that there was a conflict with my add and print instructions. When designing my ISA, I had created a conflict between the opcodes of my add/sub and print commands. I had to change the format of my opcodes in my ISA design so there was no longer a conflict.

**Conclusion**

Overall, I was successfully able to implement my 8-bit calculator ISA design into functional VHDL code. I began by using my original design from lab 1 (the ISA design), I continually added data paths for the various instructions. When I had all my data paths, I began translating them into VHDL code. While making my components and testing my calculator I encountered two main problems, conflicts between my opcodes and creating the skip instruction. The opcode conflict was easily fixed by recreating the ISA design from lab 1. The skip instruction was solved by using a combination of D-latches and flip-flops and using the example logic for the skip instruction as was shown in class.

**Appendix I:**

Included are the figures cited in the above report

**Fig 1: Add-sub.vhdl**

**A screenshot of a cell phone

Description automatically generated**

A screenshot of a computer

Description automatically generated

**Fig 2: regFile.vhdl**

**A screenshot of a cell phone

Description automatically generated**

**Fig 3: clock-filter.vhdl**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a cell phone

Description automatically generated**

**Fig 4: Calc.vhdl**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**