EGCP381-01 LAB 2 – TINY8 Microprocessor 2/9/20

By James Samawi

Introduction

In this lab, I programmed the data path which links the components of a Tiny8 microprocessor together for the purpose of computing the Fibonacci sequence (numbers less than 256) within the microprocessor's RAM. I coded the signals and linked components together within the port map statements; then I utilized the testbench in Xilinx Vivado and tested the RAM signal where the sequence is located. The purpose of this lab was to familiarize the way of connecting components together in VHDL and to set ourselves up for the future. This sort of project is important in the field of digital design such that a computer is comprised of different departments with their own job and require a bus to transfer data and link the system together.

Procedure

I thought this lab was fairly straightforward. I made sure every connection was right before I began the simulation. The first half of the lab I connected all the "easy" signals in their respected port maps (these signals were located in the entity list). After I ran out of options, I began creating signals while staying mindful of how many bits wide they were for the second half of the lab. I also made sure the zero-extender component outputted an extra 4 bits to create an 8 bit signal. After double checking the code, I ran the project through a simulation and drag & dropped the RAM signal from memory and saw the correct result. Getting code to work for the first time for me is rare, so I was pleased. I know the project testbench was correct because of the iconic Fibonacci that was displayed every clock cycle. Once the output in the RAM passed 256, overflow would occur and would instead output a number that is the difference of 256.

- **Why are the Fibonacci values less than 256? Do any unexpected values appear in your Fibonacci sequence? Please explain.**

  The Fibonacci values are initially less than 256 because of the nature of the sequence. It is the addition of the previous two values that form the next one, and so on. The pattern is discontinued once the processor calculates the output to be of a value greater than 256.

Overflow occurs since our 8 bit processor encountered a number larger than 8 bits wide. In the next clock cycle, the RAM signal displayed a number that is the difference of 256.

- **How does the datapath for this Tiny8 processor compare to the accumulator-based processor from Lab 1? What components are different between these two datapaths? How do these differences impact the instructions these processors can execute?**

These processors are similar with how they contain ALUs that are programmed to compute a certain instruction, and are structured like a circuit or a loop, that will continue to execute continuously. The main difference between the two is that the Tiny8 stores the output in memory every clock cycle. The Tiny8 has different components such as two 4 input muxes, a 2 input mux, two zero extenders, and six D flip-flop based registers (five more than in lab 1). These impact the effectiveness and practicality of the Tiny8 processor, as it can handle opcode instructions better and actually store the ALU's output in memory.

- **Are you familiar with architectures of any other processors (e.g. ARM, x86, MIPS, etc.)? If so, how does the datapath for this Tiny8 processor compare to the other processor(s)?**

I am mostly familiar with ARM architecture, and I'm certain this processor has all of the same ARM instructions. The Tiny8 processor has addition, add immediate, subtraction, subtract immediate, load, store, and jump. The ARM counterparts are ADD, SUB, LDR, STR, MOV, and B.

Conclusion

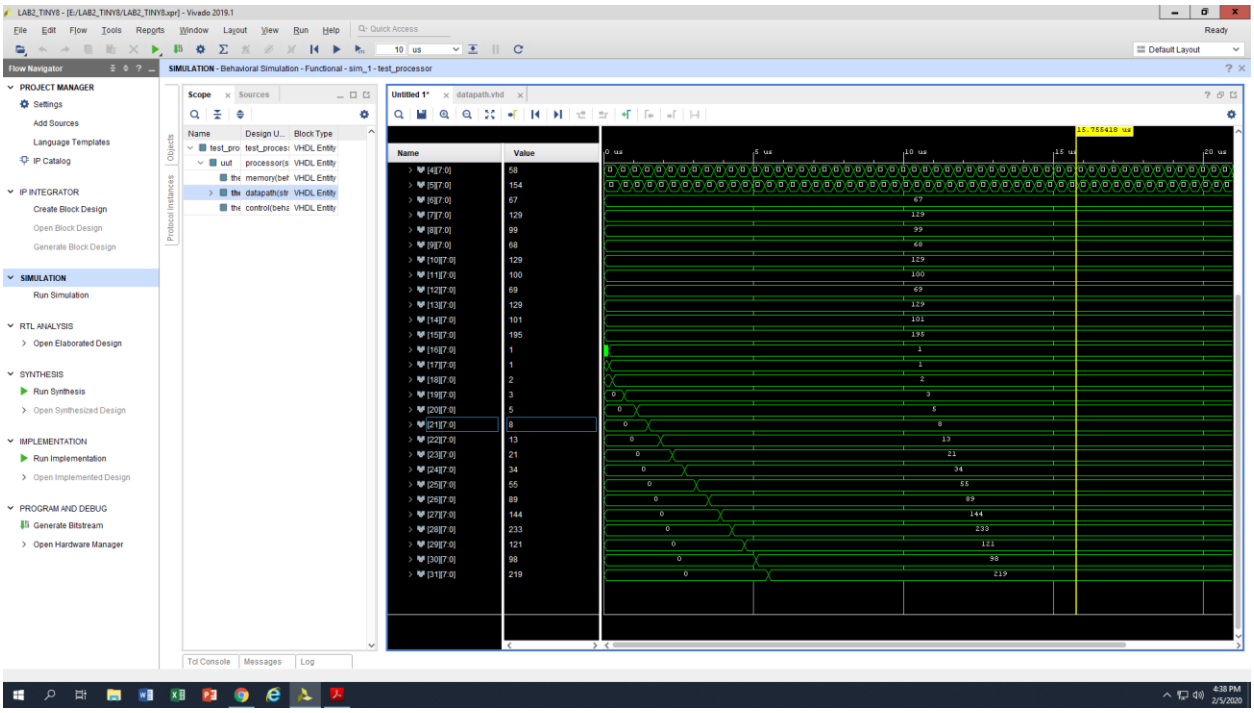
After completing this lab, I learned a lot about the architecture of this miniature computer and understand the importance of processors in larger-scaled systems. I am pleased with the result and could not find a reason to ever redo portions of this lab if given the chance. For all I know, there isn't a more efficient way to redo parts of the datapath file. I found the lab to be very insightful, and I reinforced my knowledge from the previous lab. Even though at times I was lost, I thought the lab was very fair, considering the amount of code that was given. Other than

that, I have no further comments besides the lab being a great learning experience in general and helpful to recall for future experiments.

References

I used my friends Levi and Ameya for help on debugging. Levi helped me understand the interconnections in the project, and Ameya helped me interpret my waveform.

Appendix



DATAPATH DESIGN FILE

```
library IEEE;

use IEEE.std_logic_1164.all;
use work.t8_types.all;

-- datapath Entity Description
-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 2 at Pacific Lutheran University

entity datapath is
    port( CLK : in std_logic;
```

```
        RESET : in std_logic;

        -- I/O with Memory ****
        DATA_IN : out t8_data;
        DATA_OUT : in t8_data;
        ADDRESS : out t8_address;

        -- Control Signals to the ALU*******
        ALU_OP : in std_logic_vector(1 downto 0);
        B_INV : in std_logic;
        CIN : in std_logic;

        -- Control Signals from the ALU *****
        ZERO_FLAG : out std_logic;
        NEG_FLAG : out std_logic; -- last bit in alu_out

        -- ALU Multiplexer Select Signals *****
        A_SEL : in std_logic_vector(1 downto 0);
        B_SEL : in std_logic_vector(1 downto 0);
              MEM_SEL : in std_logic;

        -- Enable Signals for all registers *****
        EN_A : in std_logic;
        EN_PC : in std_logic;
        EN_OP : in std_logic;
        EN_FIELD : in std_logic;
        EN_DATA : in std_logic;
        EN_ADDR : in std_logic;

        -- OpCode sent to the Control ****
        INSTR_OP : out t8_opcode);
end datapath;

-- datapath Architecture Description
architecture structural of datapath is

  -- declare all components and their ports
    component address_reg
       port( CLK : in std_logic;
             RESET : in std_logic;
             EN : in std_logic;
             D : in t8_address;
             Q : out t8_address);
    end component;

    component data_reg
```

```
   port( CLK : in std_logic;
         RESET : in std_logic;
         EN : in std_logic;
         D : in t8_data;
         Q: out t8_data);
end component;

component op_reg
   port( CLK : in std_logic;
         RESET : in std_logic;
         EN : in std_logic;
         D : in t8_opcode;
         Q : out t8_opcode);
end component;

component alu
   port( A : in t8_data;
         B : in t8_data;
         OP : in std_logic_vector(1 downto 0);
         D : out t8_data;
         Z : out std_logic;
         CIN : in std_logic;
         B_INV : in std_logic);
end component;

component mux4
   port( IN0 : in t8_data;
         IN1 : in t8_data;
         IN2 : in t8_data;
         IN3 : in t8_data;
         SEL : in std_logic_vector(1 downto 0);
         DOUT : out t8_data);
end component;

component mux2
   port( IN0 : in t8_address;
         IN1 : in t8_address;
         SEL : in std_logic;
         DOUT : out t8_address);
end component;

component zero_extend is
   port( A : in t8_address;
         Z : out t8_data);
end component;
```

```
    signal zero_8 : t8_data := "00000000";
    signal alu_out : t8_data;
    signal a_out : t8_data;

    signal one_8 : t8_data := "00000001";
    signal BtoALU : std_logic_vector (7 downto 0);
    signal AtoALU : std_logic_vector (7 downto 0);
    signal DatatoIN2: std_logic_vector (7 downto 0);
    signal PCtoZERO: std_logic_vector (4 downto 0);
    signal PZEROtoIN3: std_logic_vector (7 downto 0);
    signal FIELDtoZERO: std_logic_vector (4 downto 0);
    signal FZEROtoIN3: std_logic_vector (7 downto 0);
    signal ADDRtoMEM: std_logic_vector (4 downto 0);



begin

    the_alu: alu port map (
       A     => AtoALU,
       B     => BtoALU,
       OP    => alu_op,
       D     => alu_out, --output
       Z     => zero_flag,
       CIN   => cin,
       B_INV => b_inv
       );

    Amux: mux4 port map (
       IN0  => zero_8,   -- 00
       IN1  => zero_8,   -- 01 nothing
       IN2  => a_out,   -- 10 A
       IN3  => PZEROtoIN3,   -- 11 PC
       SEL  => a_sel, --a select
       DOUT => AtoALU
       );

    Bmux: mux4 port map (
       IN0  => zero_8,   -- 00
       IN1  => one_8,   -- 01
       IN2  => datatoIN2,   -- 10 data
       IN3  => FZEROtoIN3,   -- 11 field
       SEL  => b_sel, --b select
       DOUT => BtoALU
       );
```

```
MEMmux: mux2 port map (
   IN0  => PCtoZERO,   -- 00
   IN1  => ADDRtoMEM,   -- 01
   SEL  => mem_sel,
   DOUT => address
   );

op: op_reg port map (
   CLK   => clk,
   RESET => reset,
   EN    => en_op,
   D     => data_out(7 downto 5),
   Q     => INSTR_OP
   );

addr: address_reg port map (
   CLK   => clk,
   RESET => reset,
   EN    => en_addr,
   D     => alu_out(4 downto 0),
   Q     => ADDRtoMEM
   );

field: address_reg port map (
   CLK   => clk,
   RESET => reset,
   EN    => en_field,
   D     => data_out(4 downto 0),
   Q     => FIELDtoZERO
   );

pc: address_reg port map (
   CLK   => clk,
   RESET => reset,
   EN    => en_pc,
   D     => alu_out(4 downto 0),
   Q     => PCtoZERO
   );

a: data_reg port map (
   CLK   => clk,
   RESET => reset,
   EN    => en_a,
   D     => alu_out(7 downto 0),
   Q     => a_out
```

```
        );

    data: data_reg port map (
        CLK   => clk,
        RESET => reset,
        EN    => en_data,
        D     => data_out(7 downto 0),
        Q     => datatoIN2
        );

    field_zero_ext: zero_extend port map ( --concatinates 4 zero bits
        A   => FIELDtoZERO,
        Z   => FZEROtoIN3
        );

    pc_zero_ext: zero_extend port map ( --concatinates 4 zero bits
        A   => PCtoZERO,
        Z   => PZEROtoIN3
        );

    DATA_IN <= a_out;

    NEG_FLAG <= alu_out(7);

end structural;
```