Justin Carlson

1/31/2023

Prof. Mealy

CPE 233-07

Experiment 2: Program Counter

Executive Summary:

Experiment involves the implementation of the Program Counter, which houses and controls the register keeping track of the address of the MCU's current instruction. The PC is programmed to either advance the count by a fixed amount to read the next instruction, or to branch to another operation as given by its current address.

PC_MOD:

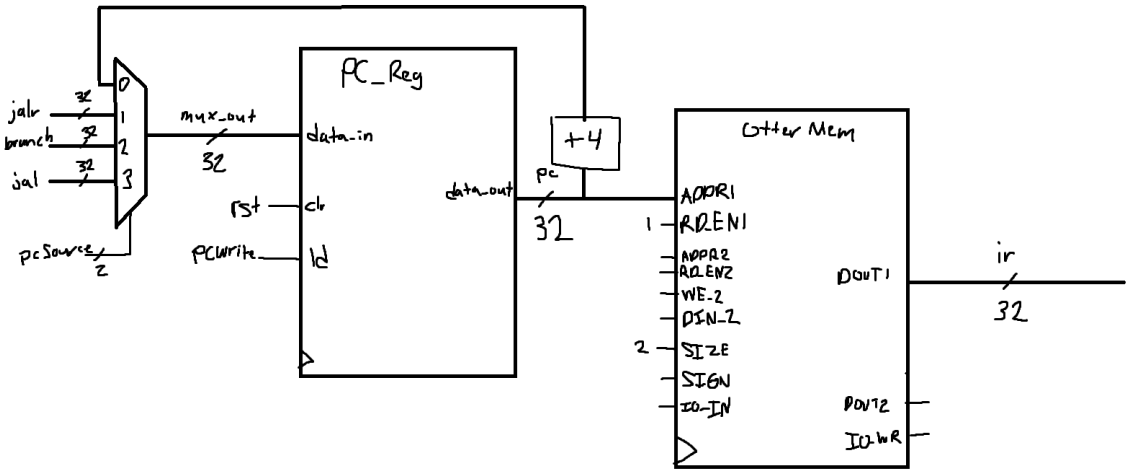
```
module PC_MOD(
   input rst,
   input PCWrite,
   input clk,
   input [1:0] pcSource,
   input [31:0] jalr, branch, jal,
   output [31:0] address
   );

   logic [31:0] mux_out, input_address;

   mux_4t1_nb  #(.n(32)) mux (
      .SEL   (pcSource), // pcSource is the selector for the mux
      .D0    (input_address), // PC Output address + 4
      .D1    (jalr),
      .D2    (branch),
      .D3    (jal),
      .D_OUT (mux_out) );

   reg_nb_sclr #(.n(32)) PC (
      .data_in (mux_out),
      .clk (clk),
      .clr (rst), // Sets address to 0
      .ld (PCWrite), // Determines if address should advance or hold
      .data_out (address)   );

   assign input_address = address + 4; // Steps to next word
```

```
endmodule
```

Program_Counter:

```
module Program_Counter(
   input rst,
   input PCWrite,
   input clk,
   input [1:0] pcSource,
   output reg [31:0] ir
   );

   wire [31:0] pc;

   PC_MOD PC_MOD (
      .rst (rst),
      .PCWrite(PCWrite),
      .clk(clk),
      .pcSource (pcSource),
      .jalr (32'h00004444),
      .branch (32'h00008888),
      .jal (32'h0000CCCC),
      .address (pc)   );

  Memory OTTER_MEMORY (
      .MEM_CLK   (clk),
      .MEM_RDEN1 (1'b1),
      .MEM_RDEN2 (1'b0),
      .MEM_WE2   (1'b0),
      .MEM_ADDR1 (pc[15:2]),   // 14-bit signal
```

```
    .MEM_ADDR2 (32'd0),
    .MEM_DIN2  (32'd0),
    .MEM_SIZE  (2'b10), // Word-addressable memory
    .MEM_SIGN  (1'b0), // Signed values
    .IO_IN     (1'b0),
    .IO_WR     (),
    .MEM_DOUT1 (ir),        // Output signal
    .MEM_DOUT2 ()   );

Endmodule
```



Test Instructions:

```
.text
init:   li   x0,12      # initialize register
        li   x1,8       #
        li   x2,4       #

main:   add  x3,x7,x8   # do something meaningful
        sub  x6,x8,x9   #
```

```
    sra     x3,x7,x8     #
    j       main         # repeat doing meaningful things
```

Memory Dump:

00c00013

00800093

00400113

008381b3

40940333

4083d1b3

ff5ff06f

Discussion Questions:

1. In your own words, describe the relationship between the programming model and the instruction set of a given computer.

While the programming model designates the resources and pathways that are available to controlling the computer at a high-level, its instruction set actually provides the methods and capabilities to actually controlling it through the use of combining specific instructions.

2. Briefly explain why you should implement the PC as a simple register instead of a counter as the name implies.

The idea of the Program Counter is that the current address to be operated on is loaded into some sort of memory. We use a register to perform this, since registers are meant for loading and storing data in short-term increments, and we never operate on the PC by incrementing it.

3. If the PC were to advance every clock cycle, could the RISC-V Otter memory output ever show the data associated with the current value on the PC's output? In other words, could the output of the memory ever be associated with the current output of the PC. Briefly but fully explain your answer in terms of the operating characteristics of the MEMORY module.

No, the described operation is impossible. Since the memory reads occur on a synchronous signal, changing the address in the PC and reading the memory cannot be done on the same clock cycle. It takes a cycle to load the value from the register, then it takes an additional cycle to check the given value in the memory then output it to other parts of the computer. If the PC is always advancing every clock cycle, the memory module must be one cycle behind.

4. The PC has two types of input signals: control signals and data signals, where we consider the

clock input a type of control signal. List which inputs are data inputs, and which are control inputs. For this problem, use the PC as it appears as the PC_MOD in Figure 8(b).

The data inputs include the address signal, and the jalr, branch, and jal signals. The control inputs are the clk, PCWrite, pcSource, and rst.

5. Why does there need to be different inputs associated with the jal and jalr instructions? Examine the instruction's explanations in assembly language manual for this question.

The jal and jalr instructions work for different purposes. According to the RTL of each, they both operate on the PC, but jal uses the PC to operate on itself while jalr uses a register on the PC. The jal instruction uses a relative address to determine the next instruction, while jalr uses absolute addressing.

6. What is the capacity of the MEMORY module? List the capacity in both bytes and words. Examine the source code for the MEMORY module to answer this question.

The memory module is 16k x 32 bits, which is 16k x 4 bytes and 16k x 2 words.

7. Briefly explain why this experiment asked you to include the dump file associated with the assembly language program in the lab submission.

Possibly to demonstrate how instructions are accessed in the memory through the use of the PC and describe their relationship with each other.

8. Briefly explain whether the RISC-V Otter handle operations on both signed and unsigned data.

It works with both signed and unsigned data as there's different instructions that handle both. Generally though, the data is usually signed.

9. Briefly explain why the data output from the MEMORY module associated with the jal, jalr, and branch inputs produced junk on the outputs (unknown outputs) for this experiment.

Since there was no data in the memory specified at the test addresses for the jal, jalr, and branch instructions, the memory produced outputs instead. Once more components are hooked up to the PC, the jal, jalr, and branch instructions will have meaningful outputs.

10. Briefly explain why the PC advanced by 4 when it is executing instructions.

When the PC doesn't have to make any jumps elsewhere, to get to the next instruction the PC simply has to read the data on the next line of the subroutine. Advancing the PC by 4 leads it to the next address of the instruction to be executed.

11. Briefly explain why the two LSBs of the PC's output are not connected to the memory module.

Since all the RISC-V instructions are 4-bytes long, you can't fit another instruction into the two smallest bits in the memory module.

12. Briefly describe whether the MEMORY module in general is bit, byte, halfword, or word addressable. Fully explain your answer.

The memory module has two data inputs, ADDR1 and ADDR2, and ADDR1 is word addressable while ADDR2 is byte addressable. For the purpose of the Otter MCU, we use the memory module as program memory that requires word addressable memory for reading instructions.

13. Briefly describe whether the "instruction memory" part of the MEMORY module is bit, byte, halfword, or word addressable. Fully explain your answer.

Since ADDR1 is purely used for the program instruction control, we use the program memory input that's word addressable. Since the PC exports a string of bytes, it needs the final two bits truncated to form the word addressable address.

Programming Assignment:

a.

```
init: li     x20, 0xC000_00BB

li x21, 0xC000_00FF

li x5, 10

mv x6, x0


main: lbu    x10,0(x20)

lbu    x11,0(x20)


loop: beq x5, 0, end

addi x6, x6, x10


admin: addi x5, x5, -1

j loop


end: sw x16, 0(x21)

j init
```

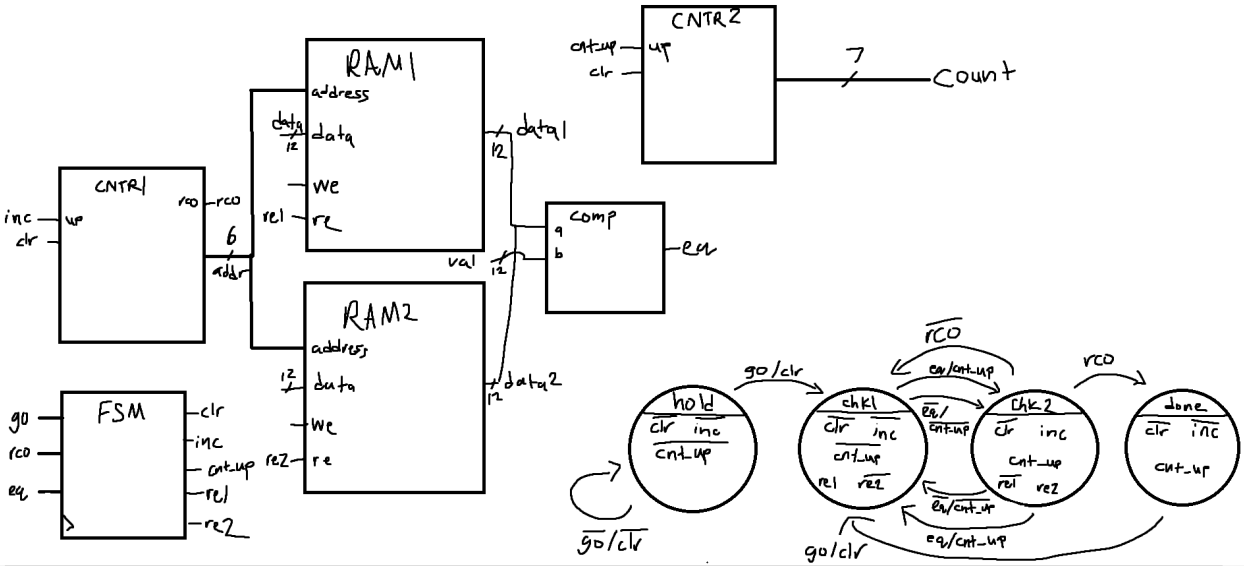
b.

```
init: li     x20, xC000_00EE

li x21, 0xC000_00FF

li x22, 0x066

mv x15, 0

mv x16, 0

mv x17, 0


main: lw    x10, 0(x20)

lw    x11, 4(x20)

neg x15, x10

neg x16, x11

add x16, x16, x15

neg x17, x22

add x16, x16, x17


end: sw x16, (x21)

j init
```

c.

In the second line of main and the second line of the second output operation, x21 is loaded from x11/12 without an offset so x20 and x21 will be identical. It should be changed to lw x21, 4(x11/12). The last line before the jump also outputs the data back in x10 rather than the memory so it should be changed to sw x20,0(x12).

Hardware Assignment:



This design implements minimum hardware since it only requires two counters, the RAMs, and a comparator. The FSM takes two clock cycles to read one address (one for each memory), so it's not necessarily the fastest solution. Since there's two RAMs with 64 addresses per, it will take 129 clock cycles to finish counting and go to the done state.