ECE/CSE 4710/5710 Computer Hardware Design D. M. Hanna

Fall 2022

Lab 6 – Program Counter and Program Control

A block diagram for the *Lab6* top-level design is shown in Figure 1. It is based on the single-cycle processor of Lab 4.

**Fig. 1** Top-level design for module *Lab6*

**Program Counter, PC**

The module *Pcount* from Lab 4 has been replaced by the program counter, *PC*. This program counter behaves like *Pcount* with the added feature that it can be loaded with a value when *pload* = ‘1’. The value comes from *M*, the output of the *PROM*. Listing 1 shows the VHDL code for the *PC* program counter.

**library** IEEE;

**use** IEEE.std\_logic\_1164.**all**;

**use** IEEE.std\_logic\_unsigned.**all**;

**entity** PC **is**

**port** (

d: **in** STD\_LOGIC\_VECTOR (15 **downto** 0);

clr: **in** STD\_LOGIC;

clk: **in** STD\_LOGIC;

inc: **in** STD\_LOGIC;

pload: **in** STD\_LOGIC;

q: **out** STD\_LOGIC\_VECTOR (15 **downto** 0)

);

**end** PC;

**architecture** PC\_arch **of** PC **is**

**signal** COUNT: STD\_LOGIC\_VECTOR (15 **downto** 0);

**begin**

**process** (clk, clr)

**begin**

**if** clr = '1' **then**

COUNT <= "0000000000000000";

**elsif** clk'event **and** clk='1' **then**

**if** pload = '0' **then**

**if** inc = '1' **then**

COUNT <= COUNT + 1;

**end if**;

**else**

COUNT <= d;

**end if**;

**end if**;

**end process**;

q <= COUNT;

**end** PC\_arch;

**Listing 1** *PC.vhd*

**Opcodes**

For convenience, we can create a VHDL package for the opcode constants. Notice that in this microcontroller, the opcodes are assigned numbers and are not developed directly from their control signals (i.e. *nload, tload, msel, fcode*) like they were in Lab 4. The listing for the package is shown in Listing 2.

**package** opcodes **is**

**subtype** opcode **is** std\_logic\_vector(15 **downto** 0);

-- Register instructions --WHYP WORDS

**constant** nop: opcode := X"0000"; -- NOP

**constant** dup: opcode := X"0001"; -- DUP

**constant** swap: opcode := X"0002"; -- SWAP

-- Function unit instructions

**constant** plus: opcode := X"0010"; -- +

**constant** minus: opcode := X"0011"; -- -

**constant** plus1: opcode := X"0012"; -- 1+

**constant** minus1: opcode := X"0013"; -- 1-

**constant** invert: opcode := X"0014"; -- INVERT

**constant** andd: opcode := X"0015"; -- AND

**constant** orr: opcode := X"0016"; -- OR

**constant** xorr: opcode := X"0017"; -- XOR

**constant** twotimes: opcode := X"0018"; -- 2\*

**constant** u2slash: opcode := X"0019"; -- U2/

**constant** twoslash: opcode := X"001A"; -- 2/

**constant** rshift: opcode := X"001B"; -- RSHIFT

**constant** lshift: opcode := X"001C"; -- LSHIFT

**constant** ones: opcode := X"0020"; -- TRUE

**constant** zeros: opcode := X"0021"; -- FALSE

**constant** zeroequal: opcode := X"0022"; -- 0=

**constant** zeroless: opcode := X"0023"; -- 0<

**constant** ugt: opcode := X"0024"; -- U>

**constant** ult: opcode := X"0025"; -- U<

**constant** eq: opcode := X"0026"; -- =

**constant** ugte: opcode := X"0027"; -- U>=

**constant** ulte: opcode := X"0028"; -- U<=

**constant** neq: opcode := X"0029"; -- <>

**constant** gt: opcode := X"002A"; -- >

**constant** lt: opcode := X"002B"; -- <

**constant** gte: opcode := X"002C"; -- >=

**constant** lte: opcode := X"002D"; -- <=

-- I/O instructions

**constant** sfetch: opcode := X"0037"; -- S@

**constant** digstore: opcode := X"0038"; -- DIG!

-- Transfer instructions

**constant** lit: opcode := X"0100"; -- LIT

**constant** jmp: opcode := X"0101"; -- AGAIN, ELSE

**constant** jz: opcode := X"0102"; -- IF, UNTIL

**constant** jb0HI: opcode := X"010D"; -- waitB0

**constant** jb0LO: opcode := X"0109";

**end** opcodes;

**Listing 2** Package for opcode constants

**Program Controller, Pcontrol**

Since the instructions are not microcoded, it is necessary to have a decoder that outputs the correct control signals for *tload*, *nload*, and other signals based on the input instruction. The controller for out processor contains the VHDL state machine for the processor as well as the instruction decoder. This state machine uses three processes, one synchronous process and two combinational. The synchronous process changes the current state to the next state on the rising edge of the clock. Listing 3 shows the entity for *Pcontrol* and the synchronous state machine process *synch*.

**library** IEEE;

**use** IEEE.std\_logic\_1164.**all**;

**use** IEEE.std\_logic\_arith.**all**;

**use** work.opcodes.**all**;

**entity** Pcontrol **is**

**port** (

icode: **in** STD\_LOGIC\_VECTOR (15 **downto** 0);

M: **in** STD\_LOGIC\_VECTOR (15 **downto** 0);

clr: **in** STD\_LOGIC;

clk: **in** STD\_LOGIC;

BTN0: **in** STD\_LOGIC;

fcode: **out** STD\_LOGIC\_VECTOR (5 **downto** 0);

msel: **out** STD\_LOGIC\_VECTOR (1 **downto** 0);

pinc: **out** STD\_LOGIC;

pload: **out** STD\_LOGIC;

tload: **out** STD\_LOGIC;

nload: **out** STD\_LOGIC;

digload: **out** STD\_LOGIC;

iload: **out** STD\_LOGIC

);

**end** Pcontrol;

**architecture** Pcontrol\_arch **of** Pcontrol **is**

**type** state\_type **is** (fetch, exec, exec\_fetch);

**signal** current\_state, next\_state: state\_type;

**begin**

synch: **process**(clk, clr)

**begin**

**if** clr = '1' **then**

current\_state <= fetch;

**elsif** (clk'event **and** clk = '1') **then**

current\_state <= next\_state;

**end** **if**;

**end** **process** synch;

**Listing 3** Entity and synchronous process for *Pcontrol*

Process *C1* controls the next-state signal based on the current state and the value from the *PROM*. Since the instructions with bit 8 set to ‘1’ signify a multi-clock-cycle instruction, the next state is set accordingly. Process *C2* decodes the instruction and sets the appropriate control signals for the microcontroller. This process is based on the current instruction, *icode*, and the current state. Listing 4 contains the VHDL code for processes *C1* and *C2* for the entity *Pcontrol*.

C1: **process**(current\_state, M)

**begin**

**case** current\_state **is**

**when** fetch =>

**if** M(8) = ‘1’ **then**

next\_state <= exec;

**else**

next\_state <= exec\_fetch;

**end if**;

**when** exec\_fetch =>

**if** M(8) = ‘1’ **then**

next\_state <= exec;

**else**

next\_state <= exec\_fetch;

**end if**;

**when** exec =>

next\_state <= fetch;

**end case**;

**end process** C1;

C2: **process**(icode, current\_state, BTN4)

**begin**

fcode <= "000000"; msel <= "00"; pload <= '0'; tload <= '0';

nload <= '0'; digload <= '0'; pinc <= '1'; iload <= '0';

**if** (current\_state = fetch) **or**

(current\_state = exec\_fetch) **then**

iload <= '1'; -- fetch next instruction

**end if**;

**if** (current\_state = exec) **or**

(current\_state = exec\_fetch) **then**

**case** icode **is**

**when** nop =>

null;

**when** dup =>

nload <= '1';

**when** plus =>

tload <= '1'; fcode <= icode(5 downto 0);

msel <= "01";

**when** plus1 =>

tload <= '1'; fcode <= icode(5 downto 0);

msel <= "01";

**when** invert =>

tload <= '1'; fcode <= icode(5 downto 0);

msel <= "01";

**when** twotimes =>

tload <= '1'; fcode <= icode(5 downto 0);

msel <= "01";

**when** sfetch =>

tload <= '1'; msel <= "10";

**when** digstore =>

digload <= '1';

**when** jmp =>

pload <= '1'; pinc <= '0';

**when** jb0LO =>

pload <= not BTN0; pinc <= BTN0;

**when** jb0HI =>

pload <= BTN0; pinc <= not BTN0;

**when** others => **null**;

**end case**;

**end if**;

**end process** C2;

**Listing 4**  Processes *C1* and *C2* for *Pcontrol*

**DIGreg Register**

In this lab, we are also adding a register between *Treg* and the 7-segment display component. Using this register, we can control what is visible on the 7-segment display. Rather than permanently displaying the value in *Treg*, we will create an instruction, *digstore*, that will set *digload* high and load the value in *Treg* into *DIGreg* which will, in turn, display on the 7-segment display.

**Program ROM, Prom7**

To test the microcontroller, we will use a program that multiplies the switch setting by 10 and displays the results on the 7-segment display. This program is shown in the program ROM, *Prom7*, in Listing 5. You will need to complete the *Prom7* architecture using the ROM in Lab 4 as a model.

**library** IEEE;

**use** IEEE.std\_logic\_1164.**all**;

**use** IEEE.std\_logic\_unsigned.**all**;

**use** work.opcodes.**all**;

**entity** Prom6 **is**

**port** (

addr: **in** STD\_LOGIC\_VECTOR (15 **downto** 0);

M: **out** STD\_LOGIC\_VECTOR (15 **downto** 0)

);

**end** Prom6;

**architecture** Prom6\_arch **of** Prom6 **is**

**subtype** tword **is** std\_logic\_vector(15 **downto** 0);

**type** rom\_array **is** array (NATURAL range <>)

**of** tword;

**constant** rom: rom\_array := (

JB0HI, X"0000", -- X"00" wait for BTN0 up

JB0LO, X"0002", -- X"02" wait for BTN0

SFETCH, -- X"04" push switches

digstore, -- X"05" display

JB0HI, X"0006", -- X"06" wait for BTN0 up

JB0LO, X"0008", -- X"08" wait for BTN0

twotimes, -- X"0A" 2\*

DUP, -- X"0B" DUP

twotimes, -- X"0C" 2\*

twotimes, -- X"0D" 2\*

plus, -- X"0E" +

digstore, -- X"0F" display

JMP, X"0000", -- X"10" GOTO 0

X"0000" -- X"12"

);

**Listing 5** *Prom6.vhd*

**Top-level Design, Lab 6**

1. Write a top-level VHDL program called *Lab6* based on the block diagram shown in Figure 1. Use the *Funit1* module from Lab 4. As in Lab 4 use the process *clkdiv* to create *cclk* from the 100 MHz input clock, *mclk*. Make *BTN(0)* an input to *Lab6* and connect it to the input *BTN0* of *Pcontrol* through the *debounce* module described in lecture. Use *clkdiv(1)* to create a 25 MHz clock signal *clk*. The other input is *SW(7:0)* and the outputs are *AtoG(6:0),* *AN(3:0),* and *LD(7:0)*. Put all component declarations in a separate package called *Lab6\_components.vhd*.
2. Simulate the design using the simulator. Write a testbench to stimulate the signal *clr* to 1 at 0 ns and 0 at 5 ns. Apply a 100 MHz clock stimulator to the *clk* signal. Apply a value of 0x3B to the signal *SW[7:0]*. Stimulate the signal *BTN0* to ‘0’ at 0 ns, ‘1’ at 200 ns, ‘0’ at 300 ns, ‘1’ at 500 ns, and ‘0’ at 600 ns. Run the simulator for 1000ns. Hand in a screen shot of the waveform and listings of the files *lab6.vhd*, *Pcontrol.vhd* and *Prom6.vhd*. On a listing of the ROM indicate the values of *T* and *N* after each instruction is executed. Upload these files to Moodle as your submission.
3. Synthesize and implement the *Lab6* project and download the .*bit* file to the Nexys board. Test the program by setting the switches to X”3B”, and press *BTN(0)* twice. The hex answer, X”24E”, should be displayed on the 7-segment displays. Demonstrate the operation of *Lab6* to your lab instructor.