COEN 313
Digital Systems Design II-Winter 2017
LAB 3: Clocked Processes and registers in VHDL

The purpose of this lab is to become acquainted with clocked processes and registers in VHDL .

Introduction

In this lab, a register file with stack capability will be designed using clocked processes in VHDL. Figure 1 gives the block diagram of the system.

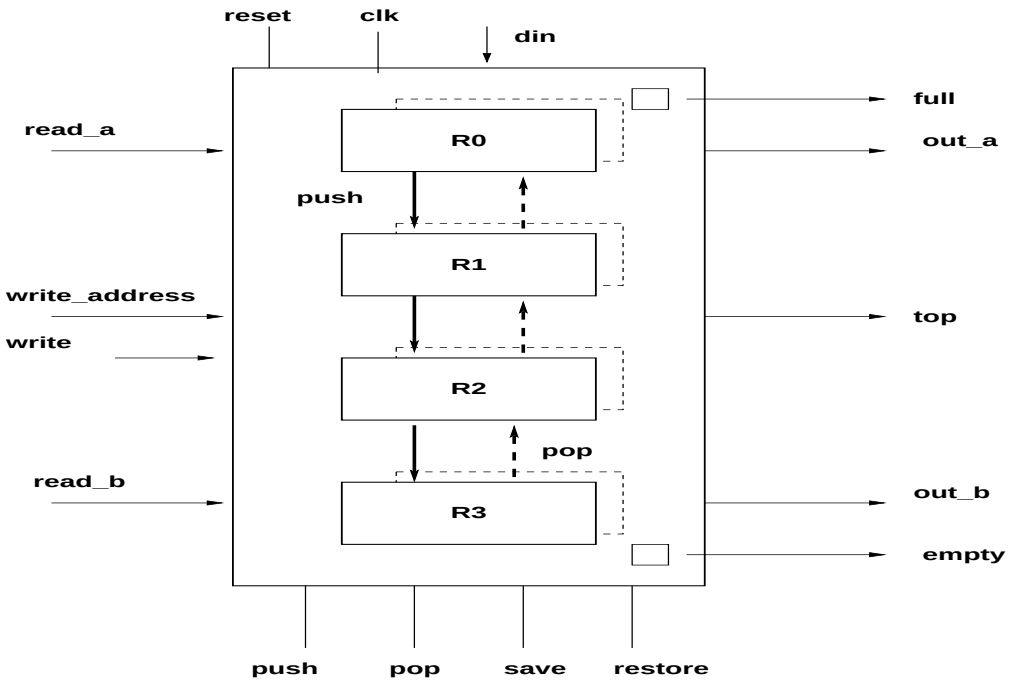

Figure 1: Block diagram of register with stack capability.

The register file consists of 4 (R0, R1, R2, R3) registers with each register consisting of 4 bits. There are two output ports: `out_a` and `out_b`. The values on the input ports `read_a` and `read_b` determine which of the 4 registers is directed to each output port. The reading is performed asynchronously. The value on the input port `write_address` determines which of the 4 registers is to be loaded with the data present on the `din` port whenever the `write` signal = '1'. The writing of a register is, of course, performed synchronously with the clock signal. In addition to these 4 registers, the register contains 4 "shadow" registers (indicated in Figure 1 by the 4 boxes in dashed lines partially hidden behind the 4 registers). These "shadow" registers will be used to save a copy of the contents of the 4 registers whenever the save input is asserted. Conversely, whenever the restore signal is asserted, the contents of the 4 shadow registers are loaded back into the registers. The saving and restoring of the registers is synchronous.

The 4 registers R0, R1, R2, and R3 also function as a PUSH/POP stack controlled by the push and pop inputs. There are also two indicators used to indicate a full stack condition (after 4 consecutive push operations have been performed for example) and an empty stack. (HINT: use a counter to keep track of the number of items in the stack - every push increments the counter and every pop decrements the counter). Once the stack has become full, any further push should have no effect on the stack. Similarly, the popping of the stack should take place only if the stack is not empty.

Table 1 summarizes the operation of this “register file with stack capability”.

Table 1: Operation of register file

	Operation	Comment
out_a =	R0 when read_a = 00	the reads
	R1 when read_a = 01	are
	R2 when read_a = 10	performed
	R3 when read_a = 11	asynchronously
out_b =	R0 when read_b = 00	independent of
	R1 when read_b = 01	the
	R2 when read_b = 10	clock
	R3 when read_b = 11	signal
	R0 = din when write_address = 00 and write =1	the writes
	R1 = din when write_address = 01 and write = 1	are
	R2= din when write_address = 10 and write = 1	synchronized with the
	R3= din when write_address = 11 and write = 1	rising clock edge
	shadow registers = original registers when save =1	saving of the registers is synchronous
	original registers = shadow registers when restore =1	restoring of the registers is synhronous

Table 1: Operation of register file

	Operation	Comment
	R0 = din R1 = R0 R2 = R1 R3 = R2	pushing din into a non-full stack
	R0 = R1 R1 = R2 R2 = R3 R3 = “0000”	popping a non-empty stack. “0000” is inserted into R3 and all the other registers get “shifted up”
	top = R0	

Procedure

Employ VHDL **clocked processes** to design this register file. You may also make use of combinational processes, concurrent signal assignment statements, etc. Simulate your design with the Modelsim simulator to verify correct functioning for several typical values of the input. Synthesize your VHDL code with Precision RTL and obtain the RTL schematic diagram as produced by the synthesis tool. Program the FPGA board with the Xilinx ISE tool. Demonstrate the operation of the design by downloading your synthesized code to the FPGA demonstration board. Use the following VHDL entity specification:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity regfile  is
port( din   : in std_logic_vector(3 downto 0);
      reset : in std_logic;
      clk   : in std_logic;
      write : in std_logic;
      read_a : in std_logic_vector(1 downto 0);
      read_b : in std_logic_vector(1 downto 0);
      write_address : in std_logic_vector(1 downto 0);
      top_out    : out std_logic_vector(3 downto 0);
      push   : in std_logic;
      pop    : in std_logic;
      save   : in std_logic;
      restore: in std_logic;
      full_out   : out std_logic;
      empty_out  : out std_logic;
      out_a_out  : out std_logic_vector(3 downto 0);
      out_b_out  : out std_logic_vector(3 downto 0));
```

```
end regfile ;
```

Note: In this design, we will assume that all the signals used to control synchronous operations (`write`, `push`, `pop`, `save`, `restore`) are mutually exclusive - only one of these signals will be asserted during any given clock cycle.

Implementation Note

Due to the large number of input and outputs, it is recommended that you use the following switches and LEDs:

`din`: use 4 of the DIP switches on the expansion I/O board
`read_a`: use 2 of the DIP switches on the expansion I/O board
`write:` use 1 switch of the expansion I/O board
`reset:` use the last remaining switch on the expansion I/O board

`read_b`: use 2 leftmost of the DIP switches on the FPGA board

NET read_b(1) LOC = AF9; (SW3)
NET read_b(0) LOC = AF8; (SW2)

`write_address`: use the other 2 (rightmost) of the DIP switches on the FPGA board

NET write_address(1) LOC = AD11; (SW1)
NET write_address(0) LOC = AC11; (SW0)

`push`: use the UP pushbutton switch on the FPGA board
`pop`: use the DOWN pushbutton switch on the FPGA board
`save`: use the LEFT pushbutton switch on the FPGA board
`restore`: use the RIGHT pushbutton switch on the FPGA board

NET push LOC = AH4;
NET pop LOC = AG3;
NET save LOC = AH1;
NET restore LOC = AH2;

The pushbutton switches are configured so that they produce a logic ‘1’ when the **NOT** pressed, pushing down on the pushbutton switch produces a logic ‘0’. Write your VHDL code accordingly so that you don’t have to be always pushing down the switches.

`out_a,_out, out_b_out` : use the 8 LEDS on the expansion I/O board
`full_out`, `empty_out`: use 2 of the 4 LEDS on the FPGA board

NET empty_out LOC = AC4; (LED0)

NET full_out LOC = AA5; (LED3)

Questions

1. The more observant reader may have noticed that the output port top_out remains unconstrained in terms of its I/O placement. What do the Xilinx implementation tools do when top-level ports in an entity do not appear in a .ucf file?

T. Obuchowicz
February 17, 2017