

Objectives

The purpose of this lab is to become acquainted with **clocked processes and registers in VHDL**. A secondary purpose is to apply combinational processes and concurrent signal assignment statements to synthesize **synchronous and asynchronous functionality**. As an example, we will be **implementing a stack with additional read/write features**.

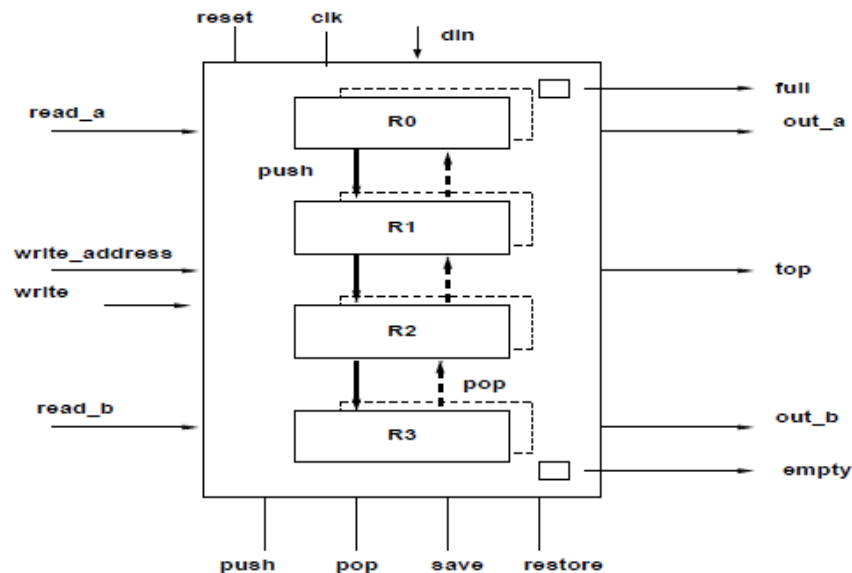


Figure 1: Block diagram of register with stack capability.

Procedure

This lab follows the procedure outlined in Parts I, II, III, and IV of the tutorial “Digital Logic Simulation and Synthesis Using Modelsim, Precision RTL, and Xilinx ISE”. The end-goal is to simulate, synthesize and download to the Xilinx FPGA board the stack with four 4-bit sized registers described in the LAB 3 manual.

Requirements

- 1) Register file: consists of 4 (R0, R1, R2, R3) registers with each register consisting of 4 bits
- 2) Two output ports: “out_a”, “out_b”
- 3) The values of “read_a” and “read_b” select registers redirected to “out_a” and “out_b” asynchronously

- 4) Value of input "write_address" selects which register is loaded with "data_in" when "write = 1" and writing is synchronous with clock
- 5) 4 "shadow" registers are used to save contents of R0-R3 when "save" input is asserted
- 6) When "restore" is asserted, the 4 shadow registers are loaded back into R0-R3, saving/restoring is synchronous with clock
- 7) R0-R3 also function as a PUSH/POP stack controlled by "push" and "pop" inputs
- 8) There's 2 indicators for "full" stack (4 consecutive pushes) and an "empty" stack

Assumptions

- 1) All signals used to control synchronous operations (write, push, pop, save, restore) are mutually exclusive – only one signal will be asserted during any clock cycle (my implementation forces a priority so this is not an issue)
- 2) Synchronous operations are tied to the rising edge of the clock
- 3) Push-buttons are inverted, pressing down produces logic '0' and when they're not pressed they produce logic '1'
- 4) My outputs are not inverted even though the board outputs are inverted
- 5) SAVE clears the stack by saving to shadow registers, to allow pushing again without popping.
- 6) WRITE simply overwrites data, does not increment stack counter (i.e. WRITE is undetectable to the stack)

Input/Output Mapping

DATA_IN:	4 Right-most DIP switches on expansion I/O board
READ_A:	2 Left-most user input switches
READ_B:	2 Right-most user input switches
RESET:	DIP switch 3 on expansion I/O board
WRITE:	ENTER
WRITE_ADDRESS:	2 Left-most DIP switches on expansion I/O board
PUSH:	UP
POP:	DOWN
SAVE:	LEFT
RESTORE:	RIGHT
OUT_A:	4 Right-most LEDs on expansion I/O board
OUT_B:	4 Left-most LEDs on expansion I/O board
FULL_OUT:	LED0
EMPTY_OUT:	LED3
TOP_OUT:	-----

Results

VHDL Code for Stack:

```
stack.vhd X

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

-- Stack implementation according to COEN 313 specs
-- Assumptions:
--             Inverted outputs
--             SAVE clears stack by saving to shadow registers,
--             so you don't have to pop to push again
--             WRITE simply overwrites data, does not increment stack counter
--             (data will be erased by push/pop operations)

entity stack is
    port (
        DATA_IN : 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);
        PUSH : in std_logic;
        POP : in std_logic;
        SAVE : in std_logic;
        RESTORE : in std_logic;

        TOP_OUT : out std_logic_vector(3 downto 0);
        FULL_OUT : out std_logic;
        EMPTY_OUT : out std_logic;
        OUT_A : out std_logic_vector(3 downto 0);
        OUT_B : out std_logic_vector(3 downto 0));
end stack;

architecture lab3 of stack is

    type mem_type is array(3 downto 0) of std_logic_vector(3 downto 0);

    signal registers : mem_type := (others => (others => '0'));
    signal shadow_registers : mem_type := (others => (others => '0'));
    signal shadow_push_counter : integer := 0;
    signal full : std_logic := '1';
    signal empty : std_logic := '0';
    signal push_counter : integer := 0;

    constant size : integer := 4; --size of stack
    constant top_ptr : integer := 0;
    constant bot_ptr : integer := 3;
```

begin

```
FULL_OUT <= full;
EMPTY_OUT <= empty;
TOP_OUT <= registers(top_ptr);
OUT_A <= registers(to_integer(unsigned(READ_A)));
OUT_B <= registers(to_integer(unsigned(READ_B)));
```

--Push/Pop/Write/Restore/Save functionality (in order of priority)

```
func : process(CLK, PUSH, POP, WRITE, RESTORE, SAVE, RESET)
```

```
    variable temp_index: integer := 0;
```

begin

-- Asynchronous reset

-- Resets EVERYTHING, stack registers & shadow registers

```
if (RESET = '0') then
```

```
    for i in top_ptr to bot_ptr loop
```

```
        registers(i) <= (others => 'Z');
```

```
        shadow_registers(i) <= (others => 'Z');
```

```
    end loop;
```

```
    full <= '1';
```

```
    empty <= '0';
```

```
    push_counter <= 0;
```

-- Synchronous with rising edge of CLK

```
elsif (rising_edge(CLK)) then
```

--PUSH

```
    if (full = '1' and PUSH = '0') then
```

--Push on top of stack if PUSH is pressed and stack is not full

-- Shift register values from top to bottom (r3 is now r2, r2 is now r1, r1 is now r0)

```
    for i in size-1 downto size - bot_ptr loop
```

```
        registers(i) <= registers(i-1);
```

```
    end loop;
```

-- Add new data to the top of stack

```
    registers(top_ptr) <= DATA_IN;
```

--Increment counter

```
    push_counter <= push_counter + 1;
```

--Set full/empty flags

```
    if (push_counter = 3) then
```

```
        full <= '0';
```

```
        empty <= '1';
```

```
    else
```

```
        full <= '1';
```

```
        empty <= '1';
```

```
    end if;
```

```

-- POP
elsif (empty = '1' and POP = '0') then
    --Pop top of stack if POP is pressed and stack is not empty

    -- Shift register values from bottom to top (r0 is now r1, r1 is now r2, r2 is now r3)
    for i in top_ptr to bot_ptr - 1 loop
        registers(i) <= registers(i+1);
    end loop;

    registers(bot_ptr) <= (others => 'Z');

    -- Decrement counter
    push_counter <= push_counter - 1;

    --Set full/empty flags
    if (push_counter = 1) then
        full <= '1';
        empty <= '0';
    else
        full <= '1';
        empty <= '1';
    end if;

-- WRITE (force)
-- No regard for stack counter, this function is "hidden" from the stack and will overwrite data
elsif (WRITE = '0') then

    temp_index := to_integer(unsigned(WRITE_ADDRESS));

    -- check if write_address corresponds to a valid entry in stack (not really necessary)
    if ((bot_ptr >= temp_index) and (temp_index >= top_ptr)) then

        registers(temp_index) <= DATA_IN;

    end if;

-- RESTORE
elsif (RESTORE = '0' and shadow_push_counter /= 0) then

    for i in top_ptr to bot_ptr loop
        registers(i) <= shadow_registers(i);
    end loop;

    push_counter <= shadow_push_counter;
    |
    if (shadow_push_counter = 0) then
        full <= '1';
        empty <= '0';
    elsif (shadow_push_counter = 4) then
        full <= '0';
        empty <= '1';
    else
        full <= '1';
        empty <= '1';
    end if;

```

```

-- SAVE
-- Copies contents to shadow_registers and clears the stack
elsif (SAVE = '0') then

    for i in top_ptr to bot_ptr loop
        shadow_registers(i) <= registers(i);
    end loop;

    shadow_push_counter <= push_counter;

    for i in top_ptr to bot_ptr loop
        registers(i) <= (others => 'Z');
    end loop;

    push_counter <= 0;
    full <= '1';
    empty <= '0';

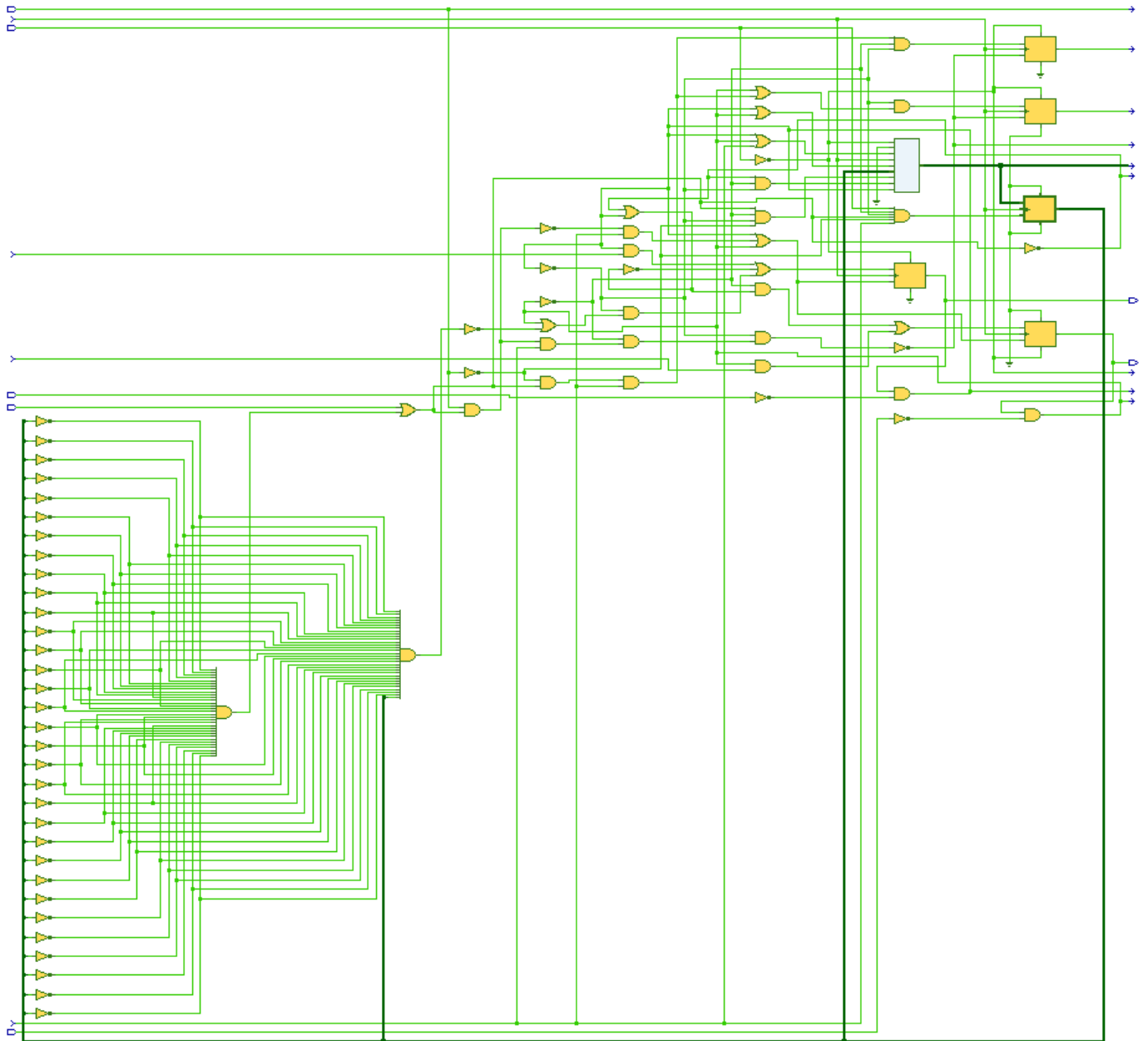
else
    -- do nothing
end if;

end process;

end lab3;

```

Synthesized circuit:



Do files to simulate my circuit:

```
clk.do X
force clk 1
run 2
force clk 0
run 2
```

```
stack.do X

#inputs
add wave DATA_IN
add wave RESET
add wave CLK
add wave WRITE
add wave READ_A
add wave READ_B
add wave WRITE_ADDRESS
add wave PUSH
add wave POP
add wave SAVE
add wave RESTORE
add wave registers

#outputs
add wave TOP_OUT
add wave FULL_OUT
add wave EMPTY_OUT
add wave OUT_A
add wave OUT_B

#set the clk signal to a value of 1 at a time equal to 0 time units
#after the current simulation time and this will be repeated at a time commencing at 2 time units
#after the current simulation time
#force CLK 0 0 -r 2

#set the clk signal to a value of 1 at a time equal to 1 units
#after the current simulation time and this will be
#repeated starting at 2 time units after the current simulation time.
#force CLK 1 1 -r 2

force PUSH 1
force POP 1
do clk.do

#pop while empty
force PUSH 1
force POP 0
do clk.do
force POP 1

#push 0001
force DATA_IN 0001
force PUSH 0
force POP 1
do clk.do
force PUSH 1

#push 0002
force DATA_IN 0010
force PUSH 0
force POP 1
do clk.do
force PUSH 1
```



```

#pop twice, should empty stack
force POP 0
force PUSH 1
do clk.do
force POP 1

force POP 0
force PUSH 1
do clk.do
force POP 1

#pop while empty
force POP 0
force PUSH 1
do clk.do
force POP 1

#push 0110
force PUSH 0
force POP 1
force DATA_IN 0110
do clk.do
force PUSH 1

#push 1111 3 times
force PUSH 0
force POP 1
force DATA_IN 1111
do clk.do
do clk.do
do clk.do
force PUSH 1

#read bottom of stack (0110) at index 11 (R3) on both out_a and out_b
force READ_A 11
force READ_B 11
do clk.do

#try to push 1010 on a full stack
force PUSH 0
force DATA_IN 1010
do clk.do
force PUSH 1

#try to push 0000 on a full stack
force PUSH 0
force DATA_IN 0000
do clk.do
force PUSH 1

#try save, will clear stack and copy it to shadow registers
force SAVE 0
do clk.do
force SAVE 1

#push 0000 in stack 4 times
force PUSH 0
force POP 1
force DATA_IN 0000
do clk.do
do clk.do
do clk.do
do clk.do
force PUSH 1

```

```
#try restore, old values should re-appear
force RESTORE 0
do clk.do
force RESTORE 1
```

```
#try write at R0
force DATA_IN 0000
force WRITE 0
force WRITE_ADDRESS 00
do clk.do
force WRITE 1
```

```
#try write at R1
force DATA_IN 0001
force WRITE 0
force WRITE_ADDRESS 01
do clk.do
force WRITE 1
```

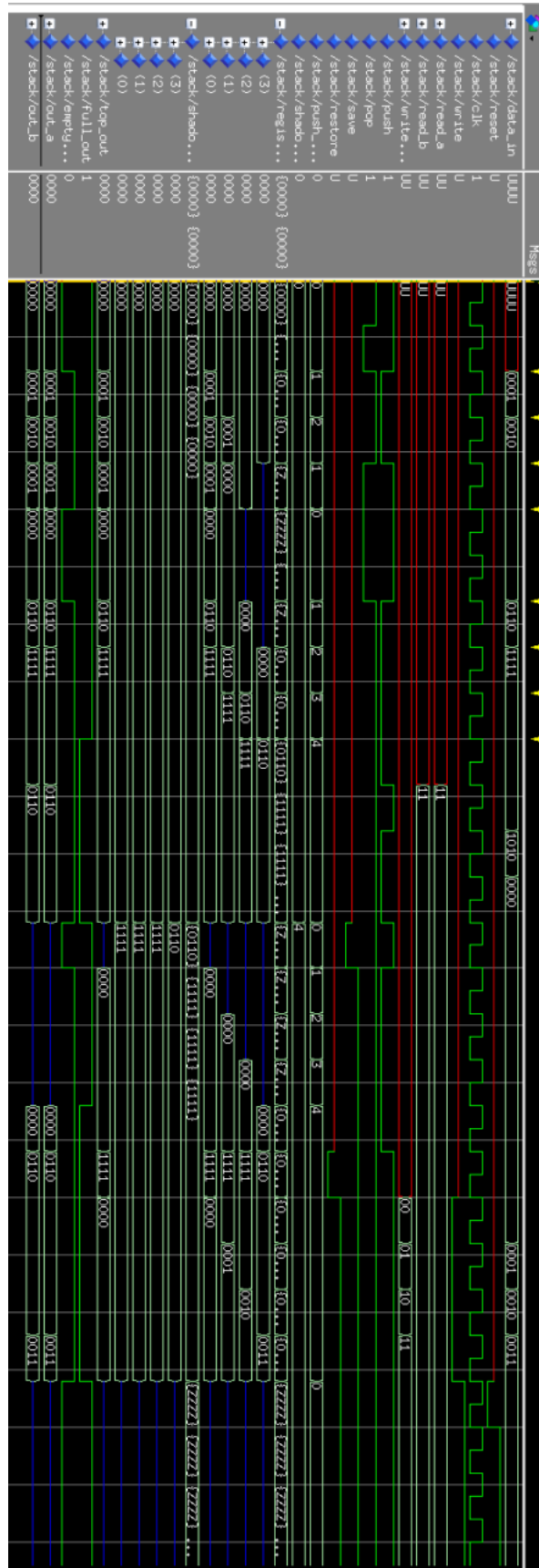
```
#try write at R2
force DATA_IN 0010
force WRITE 0
force WRITE_ADDRESS 10
do clk.do
force WRITE 1
```

```
#try write at R3
force DATA_IN 0011
force WRITE 0
force WRITE_ADDRESS 11
do clk.do
force WRITE 1
```

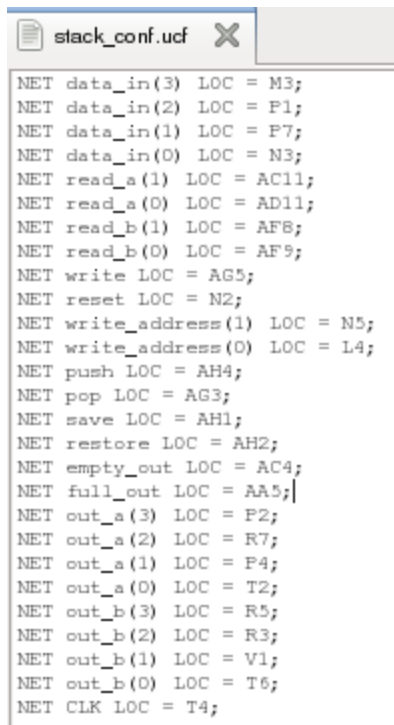
```
#try reset|
force RESET 0
do clk.do
force RESET 1
```

```
#try CLK for no reason
do clk.do
do clk.do
do clk.do
```

Modelsim Simulation:



Configuration file (I/O mapping):



```
stack_conf.ucf
NET data_in(3) LOC = M3;
NET data_in(2) LOC = F1;
NET data_in(1) LOC = F7;
NET data_in(0) LOC = N3;
NET read_a(1) LOC = AC11;
NET read_a(0) LOC = AD11;
NET read_b(1) LOC = AF8;
NET read_b(0) LOC = AF9;
NET write LOC = AG5;
NET reset LOC = N2;
NET write_address(1) LOC = N5;
NET write_address(0) LOC = L4;
NET push LOC = AH4;
NET pop LOC = AG3;
NET save LOC = AH1;
NET restore LOC = AH2;
NET empty_out LOC = AC4;
NET full_out LOC = AA5;
NET out_a(3) LOC = P2;
NET out_a(2) LOC = R7;
NET out_a(1) LOC = P4;
NET out_a(0) LOC = T2;
NET out_b(3) LOC = R5;
NET out_b(2) LOC = R3;
NET out_b(1) LOC = V1;
NET out_b(0) LOC = T6;
NET CLK LOC = T4;
```

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?

Unused resources/ports/pins are trimmed by the Xilinx technology mapper (NGDBuild phase) if it's completely unused and unconstrained.

Source: Xilinx Constraints Guide. <https://www.xilinx.com/itp/xilinx10/books/docs/cgd/cgd.pdf>

Conclusion

This lab experiment was successful in introducing clocked processes and registers in VHDL. We also observed the advantages of synchronous and asynchronous events. My stack implementation successfully met all the specifications outlined in the lab procedure: pushing, popping, saving, restoring, reading, writing and even resetting and those features were thoroughly simulated using a comprehensive DO file and tested on the board itself using the push-clock.