CSCE 496: Embedded Systems Design and Implementation

Homework 1

Due: See course web-page

A stack memory is implemented using a standard RAM with a *read port* and a *write port*. The stack can store bytes of input data (incoming messages) that would be written into the stack sequentially if the stack is not full. Only one message can be read (or pop) in a clock cycle. The message read is the one that was written (or pushed) last into the stack among the messages currently stored.

The implementation of such stack typically employs one counter, pointing at the *top* of the stack as in the Figure 1. The *top* counter contains the address of the first empty location in the stack. After each pop or push operation, the *top* counter is decremented or incremented, respectively. (Hint: You can perform a pop or a push operation from/to RAM at rising edges and increment/ decrement the top counter at falling edges).

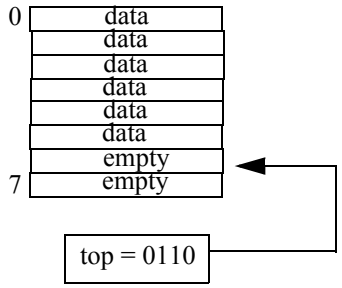

Fig. 1

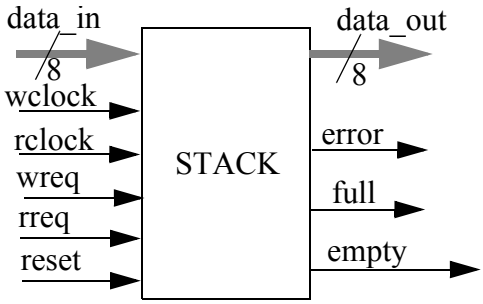

Fig. 2

In the Fig.1, there are eight entries in the stack. Thus, the counters need be only 3 bits long. However, if data is being written or pushed into the stack more often than it is being read out or pop, the *top* counter that is being incremented in the modulo 8 fashion eventually points to location 0 again (although in this case this means that the stack is full; not empty). To avoid ambiguity we can use a 4-bit counter, which will indicate that the stack is *empty* when the *top* pointer is 0000 and that it is *full* when the *top* pointer is 1000 (as seen in Fig. 1). When the stack is empty, the *empty* output line is set and a read request would result in the *error* line being set. Similarly, when the stack is full, the *full* output line is set and a write request would result in the *error* line being set.

The Figure 2 presents the possible input and output signals of the stack. Please design this stack-using a RAM with 8 bytes. This contains a *data_in* bus (8-bit wide), and a *data_out* bus (8-bit wide), and eight control signals listed below:

1. *rclock* (input): A data is read from the stack at the positive edge of *rclock* when *rreq* is asserted. .
2. *wcolck* (input): A data is written into the stack at the positive edge of *wclock* when *wreq* is asserted.
3. *wreq* (Write Request; active high; input): Enables writing into stack when asserted.
4. *rreq* (Read Request; active high; input): Enables reading from stack when asserted.

5. *full* (output): Asserted (high) only when the stack is full.
6. *empty* (output): Asserted (high) only when the stack is empty.
7. *reset* (active high; input): It will clear data in the stack and reset all the *top* counter when asserted (i.e. high).
8. *error* (output): Asserted (high) when error conditions occur.

You are asked to design this stack in VHDL with the LPM RAM component —*lpm_ram_dq* provided by Altera. The VHDL Component Declaration for *lpm_ram_dq* is as follows:

```
COMPONENT lpm_ram_dq
   GENERIC (LPM_WIDTH: POSITIVE;
      LPM_TYPE: STRING := L_RAM_DQ;
      LPM_WIDTHAD: POSITIVE;
      LPM_NUMWORDS: STRING := UNUSED;
      LPM_FILE: STRING := UNUSED;
      LPM_INDATA: STRING := REGISTERED;
      LPM_ADDRESS_CONTROL: STRING := REGISTERED;
      LPM_OUTDATA: STRING := REGISTERED);

PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
      address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
      we: IN STD_LOGIC := '1';
      inclock: IN STD_LOGIC := '1';
      outclock: IN STD_LOGIC := '1';
      q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

Note: Please use the lpm_ram-dq with following parameters:

```
lpm_ram_dq
GENERIC MAP (LPM_WIDTH => myWidth,
           LPM_WIDTHAD => myWidthAd,
           LPM_ADDRESS_CONTROL => "UNREGISTERED",
           LPM_OUTDATA => "UNREGISTERED")
PORT MAP (data => mydata_in,
           address => myaddress,
           we => mywe,
           inclock => myinclock,
           q => mydata_out); -- note: no outclock is needed in this case
--
-- The data will be written into memory at the rising edge of inclock.
-- The data will be read out from memory as soon as the address applied.
```

You also need to simulate your design to ensure correctness. Write a short report to be submitted during lab time that clearly explains your test cases. The information should include the goal of each test case, why are you choosing them, and expected output. You should have sufficient number of test cases to be sure that your design is correct.

In your report, also include the power estimation of your design when it is operated under five different clock periods (10 ns, 50 ns, 100 ns, 500 ns, and 1 us). Make sure that you set your end-time to at least 10 us. This can be done through the assignments -> settings -> simulator within Quar-

tus.

HINT: Make sure that you thoroughly test the *lpm_ram_dq* to ensure that you COMPLETELY understand the basic operation of the design (e.g. corresponding input/output signals, initialization parameters, etc.).

Submission Procedure

We will grade your design in the lab on that day. Please go to the lab early and have your design compiled and simulated on one of the machines in room 20. The report will be submitted during the grading period.

Acknowledgement

This problem was initially written by Dr. Morris Chang at Iowa State University. The initial version was written as a FIFO queue problem.