Electrical and Computer Engineering

Stack Register

By Prawat Nagvajara

Synopsis

This note covers an implementation of a stack register – Last-In First-Out (LIFO) data storage. The implementation comprises copies of register cell connected as a linear array (Fig. 1).

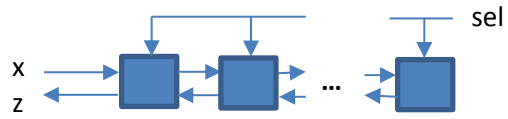

Fig. 1 Stack Register

Narrative

This implementation of LIFO storage does not use RAM (Random Access Memory) and address (pointer). With RAM, a storage uses the Stack Pointer (sp) which points to the current location available for storing new datum, after a write to the address sp, the stack pointer advances one location, i.e., sp ← sp + 1. To read a datum, the LIFO first decrements the stack pointer, i.e., sp ← sp - 1 followed by a read. The sp now points at an available location since the datum had been read.

The difference between the RAM-pointer implementation and the array-of-registers implementation is that, in the formal the pointer moves and in the later the data move. A cost analysis between the two implementations involves the energy (joules) in accessing a datum, latency (access time) and silicon area.

With RAM the energy and latency incur in datum access arise from the address decoding – multiplexing and de-multiplexing datum for read and write, respectively, and the read and write operations to the storage.

With the array-of-registers implementation, the energy spent in datum access (all registers are switching simultaneously) is higher than the RAM implementation. Since there is no address decoding the register LIFO has better access time. However, clock signal synchronization and the silicon area for large register array (stack depth and word length) poses the limitation.

Code

```
----------------------------------------------
-- Company: Drexel ECE
-- Engineer: prawat
-- Design Name: stack register
-- Description: w-bit word, n stack depth
----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity sc is
generic (w: natural);
port(
   T_in, B_in: in
   std_logic_vector(w-1 downto 0);
   ck, sel: in std_logic;
   T_out, B_out : out
   std_logic_vector(w-1 downto 0));
end sc;

architecture beh of sc is
signal temp:std_logic_vector(w-1 downto 0);
begin
process(ck)
  begin
     if ck = '1' and ck'event then
        if sel = '0' then
        temp <= T_in;
        else temp <= B_in;
        end if;
     end if;
end process;
T_out <= temp; B_out <= temp;
end beh;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity stack_reg is
generic (n : natural;
         w : natural);
port(
  x:  in
  std_logic_vector(w-1 downto 0);
  sel, ck :  in std_logic;
  z:  out
  std_logic_vector(w-1 downto 0));
end stack_reg;

architecture struc of stack_reg is
component  sc
generic (w: natural);
port(
  T_in, B_in: in
  std_logic_vector(w-1 downto 0);
  ck, sel: in std_logic;
  T_out, B_out : out
  std_logic_vector(w-1 downto 0));
end component;
type vector_array is array
      (natural range <>) of
      std_logic_vector(w-1 downto 0);
signal T, B : vector_array(n-1 downto 0);

begin
B(0) <= (others => '0');
T(n-1) <= x; --top wire into cell n-1
G1: for i in 0 to n-1 generate
 G2: if i = n-1 generate
 U: sc generic map(w)
 port map(T(i),B(i),ck,sel,T(i-1),z);
 end generate G2;
 G3: if i = 0 generate
 U: sc generic map(w)
 port map(T(i),B(i),ck,sel,open,B(i+1));
 end generate G3;
 G4: if i > 0 and i < n-1 generate
 U: sc generic map(w)
 port map(T(i),B(i),ck,sel,T(i-1),B(i+1));
 end generate G4;
end generate G1;
end struc;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity s_reg8 is
port(
 x: in std_logic_vector(0 downto 0);
 sel,ck,btn0,btn1: in  std_logic;
 z: out std_logic_vector(0 downto 0));
end s_reg8;
architecture struc of s_reg8 is
component stack_reg
generic (n : natural;
         w : natural);
port(
  x:  in
  std_logic_vector(w-1 downto 0);
  sel, ck :  in std_logic;
  z:  out
  std_logic_vector(w-1 downto 0));
end component;
signal en: std_logic;
begin
DUT: stack_reg generic map (8,1)
     port map (x, sel, en, z);
-- single step, debounce (db)
-- btn0 to enter and btn1 to reset
process(ck)
type db_state is (not_rdy, rdy, pulse);
variable db_ns: db_state;
begin
if ck='1' and ck'event then
 case db_ns is
 when not_rdy => en <= '0';
 if btn1 = '1' then db_ns:=rdy;end if;
 when rdy => en <= '0';
 if btn0 = '1' then db_ns:=pulse;end if;
 when pulse => en <= '1';
 db_ns := not_rdy;
 when others => null;
 end case;
end if;
end process;

end struc;
```

The code first describes the stack register cell (entity sc) with the top and bottom inputs and outputs and a select for multiplexing the input as new content of the register.

Next the stack register entity as with two generic parameters n the number of registers, that is, the stack depth and w the word length. The wires T and B for connecting the register cells are arrays of vectors.

Last the code describes an FPGA implementation with n = 8 and w = 1. The input x and the output y are vectors of length 1. This test case uses a clock enable signal en generated by the single-step process. User presses the button btn0 to generate a clock enable pulse then presses the button btn1 to reset.

Exercises

1. Implement a stack register with full and empty flags.
2. Implement a stack using the FPGA block RAM.