Fourth LACCEI International Latin American and Caribbean Conference for Engineering and Technology (LACCET'2006)
"Breaking Frontiers and Barriers in Engineering: Education, Research and Practice"
21-23 June 2006, Mayagüez, Puerto Rico.

Modular HDL Designs are Efficient, and Reliable

Jaime Marcelo Montenegro, Eng. Ph.D. Candidate
Research Associate, Florida International University, Miami, Florida, USA, montenegro@ieee.org

Vivek Jayaram, Eng. Ph.D. Candidate
Research Associate, Florida International University, Miami, Florida, USA, vjaya002@fiu.edu

Dr. Subbarao Wunnava, Ph.D., P.E.
Professor of Electrical & Computer Engineering, Florida International University, Miami, Florida, USA, subbarao@fiu.edu

Abstract
Digital systems are becoming more and more complex and elaborate in terms of functionality and performance. While an 8 bit system can perform well for certain applications, the need often arises for 16 or 32 bit units for certain other applications. It is impractical and inefficient to write the linear code for the enhanced systems. Also, the test bench and evaluation of the enhanced system become extremely tedious and error prone. The alternative to this linear coding is the modular approach, where either a 4 bit or 8 bit base unit is efficiently designed with the appropriate Hardware Description Language (HDL) platforms, and then integrate several of these modules in tandem, along with a microcontroller unit, to realize any complex and enhanced digital system. In this article, the authors will discuss such a modular design methodology, and present case studies of 4 bit digital units such as Counters, and Shift-registers. The methodology can be extended easily for any other sized system. The authors will also present the simulations and evaluate the performance and reliability of such designs.

Keywords
Very-large-scale integration, VHDL.

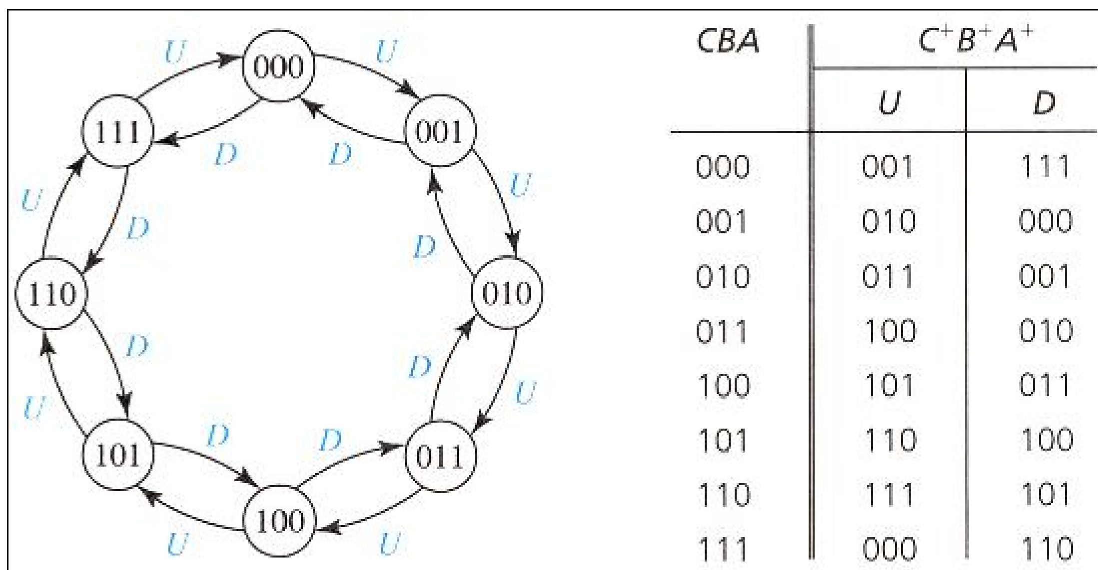
1. Introduction

Counters and register systems are basic logic systems that are used repeatedly in many applications. They provide various central functions and are fundamental to larger logic systems, such as a central processing unit (CPU). Registers are found within the CPU, in the arithmetic logic unit (ALU), and data storage devices. Registers provide the capability to store data for a desired period of time. Large arrays of registers may be implemented to supply memory to store any desired data. Counters may be used to keep track of time or to coordinate certain activities within a system. All of these logic systems are easily implemented and not complex to design, making them proficient in building larger systems.
Both, Counters and registers, can be programmed into chips through VHDL. VHDL is a programming language to specify, model, represent and simulate digital hardware. Concerns that have to be supported by VHDL are real concurrency, timings and level of signals, controls for behavior in time, interconnection and propagation delay, and existence and meaning of signal edges. VHDL allows the

designer to model hardware: special constructs and elements in the language to model several levels of abstraction, structured design and hierarchical decomposition. The traditional design methodology of gate level bottom-up (schematic entry with GUI) is replaced by RTL (register transfer level) description, such as HDL and text-files, of desired behavior. A synthesis tool is used to generate optimized gate-level implementation.

2. Counters

Counters consist of a register, or a few flip flops, which hold data that may increment or decrement over a period of time. In the design of binary counters, they may be categorized into two types, synchronous and asynchronous counters. Asynchronous counters have several clock inputs designated for the various flip flops in the system. Synchronous counters have only one clock input which drives all the flip flops simultaneously. Binary counters may be designed with any type of flip flop, most commonly JK flip flops and D flip flops. To simplify the design of a binary counter, a state diagram and state table may be generated, as shown in Figure 1.



CBA	$C^+B^+A^+$	
	U	D
000	001	111
001	010	000
010	011	001
011	100	010
100	101	011
101	110	100
110	111	101
111	000	110

Figure 1: Counter state diagram and state table

From the state table, the circuit diagram may be implemented, which depends on the type of flip flop that is chosen. A binary counter may also include input control signals which may change the behavior of the counter. A few examples of control signals to a binary counter may be RESET, SET, UP/DOWN, and ENABLE. The RESET control signal would force all the flip flops to a LOW state. The SET control signal would force all the flip flops to a HIGH state. The UP/DOWN control signal dictates whether the binary counter will increment of decrement on the next clock pulse. The ENABLE control signal at a HIGH would allow the binary counter to increment or decrement. The binary counter would not change states if the ENABLE signal is LOW.

2.1 Counter implementation

The binary counter, for all simulations, was implemented as an up/down counter. For each simulation, the binary counter included control signals of CLOCK, RESET, ENABLE, and UP/DOWN, shown in Table 1.

System Signals	Function
CLOCK	System clock that triggers the flip flops and states
RESET	Resets the counter to lowest binary number
ENABLE	Counter can only increment/decrement at HIGH
UP/DOWN	Determines if counter will increment of decrement on the next clock pulse. HIGH = increment and LOW = decrement
COUNTER_REG	Register that hold the output binary number

Table 1: Sequence diagram for the binary counter

The counter was designed to be flexible in its bit size. This implementation made it very easy to expand from 4-bit to 8-bit and to 16-bit if needed by only changing the size of the output parameter *counter_RegOut* and its respective internal signal. Listing 1 shows the VHDL implementation of the 4 bit counter, and Figure 2 shows the counter simulation.

```
-- Four-Bit up/down counter with Reset and Enable Signals
-- using a Finite State Machine (FSM) Design
library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE IEEE.numeric_std.all;
entity FourBit_Counter is
        port(
                Clk :                   in STD_LOGIC; -- main clock
                Reset :         in STD_LOGIC; -- activate high
                Enable :                in STD_LOGIC; -- activate high
                Up_Down :               in STD_LOGIC; -- count_up (active high);count_down (active low)
                Counter_RegOut :        out STD_LOGIC_VECTOR(3 downto 0)-- output register
          );
end FourBit_Counter;
architecture counterArch of FourBit_Counter is
  -- State definitions
        CONSTANT    IDLE    : INTEGER := 0;
  CONSTANT    START   : INTEGER := 1;
        -- State variables
  SIGNAL      counterState     : INTEGER;    -- present state
  SIGNAL      counterNext      : INTEGER;    -- next state
  SIGNAL      counterStart     : INTEGER;    -- start signal
        ------------------------------------------------------------
        SIGNAL      counter_Reg: UNSIGNED (3 downto 0); --holds counter output
BEGIN
  Counter_RegOut <= STD_LOGIC_VECTOR (counter_Reg);
  -- Acknowledge when counter should start
        StartCounter: PROCESS (Clk, Reset, Enable)
  BEGIN
          IF (Reset = '1') THEN
                  counterStart <= 0;
                ELSIF (Enable = '1')THEN -- if enable, start counter
                  counterStart <= 1;
                ELSIF (Enable = '0') THEN
                          counterStart <= 0;
                END IF;
        END PROCESS StartCounter;
```

```
        -- Acknowlege present state of the counter
        StateCounter: PROCESS (Clk, Reset,counterNext,Enable)
        BEGIN
          IF (Reset = '1') THEN
                        counterState <= IDLE;
    ELSIF (Enable = '1') THEN
                        counterState <= counterNext;
                ELSIF (Enable = '0') THEN
                  counterState <= IDLE;
                END IF;
        END PROCESS StateCounter;
        -- Acknowledge future state of the counter
        NextCounter: PROCESS (counterState, counterStart, Clk)
        BEGIN
                CASE counterState IS
                        WHEN START   =>         -- Start counter
                                counterNext <= START;
                        WHEN OTHERS =>
                                IF (counterStart = 1) THEN
                                  counterNext <= START; -- Start counter
                                ELSE
                                        counterNext <= IDLE;
                         END IF;
                END CASE;
        END PROCESS NextCounter;
        -- Counter process
        Counter: Process  (counterState, counterstart, Clk,Up_Down)
        BEGIN
          CASE counterState IS
            WHEN START   =>
              IF (rising_edge(Clk)) THEN
                IF (UP_DOWN = '0') THEN
                  counter_Reg <= counter_Reg - 1;
                ELSE
                  counter_Reg <= counter_Reg + 1;
                END IF;
              END IF;
             WHEN OTHERS   =>
              counter_Reg <= "0000";
          END CASE;
        END PROCESS Counter;
END counterArch;
```

Listing 1: VHDL 4-bit counter

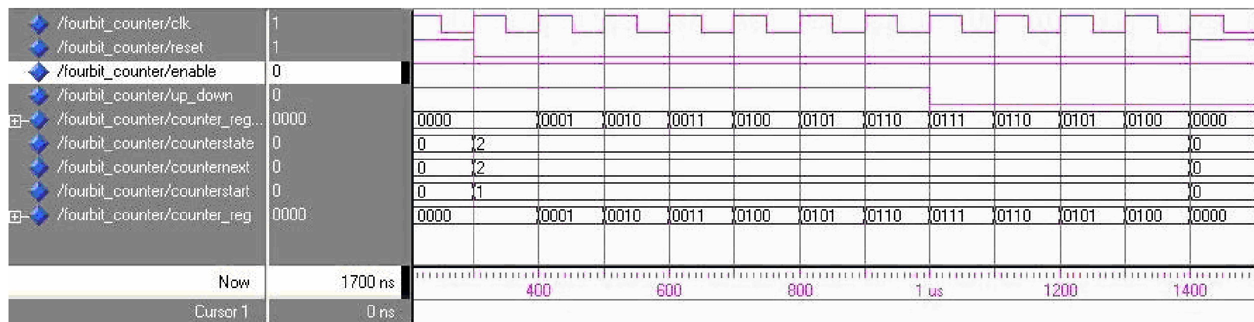


Figure 2: 4-bit counter simulation

3. Registers

Shift registers are a type of sequential logic circuit, mainly for storage of digital data. They are a group of flip-flops connected in a chain so that the output from one flip-flop becomes the input of the next flip-flop. Most of the registers possess no characteristic internal sequence of states. All the flip-flops are driven by a common clock, and all are set or reset simultaneously. Figure 3 shows an example of a 4 bit register behavior for 4 different instructions.

Original Value	S3	S2	S1	S0
	0	0	0	1
Serial Input "1"				
Left Shift	S2	S1	S0	Serial in "1"
←	0	0	1	1
Right Shift	Serial in "1"	S2	S1	S0
→	1	0	0	0
Left Rotation	S2	S1	S0	S3
	0	0	1	0
Right Rotation	S0	S3	S2	S2
	1	0	0	0

Figure 3: 4-bit Shift register instructions

3.1 Shift Register implementation

According to the design of the registers, it was important to know that *enable* and *rotenable* could not be 1 at the same time because the output *shiftreg* could not shift and rotate in the same clock cycle. The register was designed to be flexible in its bit size. This implementation made it very easy to expand from 4-bit to 8-bit and to 16-bit if needed by only changing the size of the ports parameter *shiftreg* and *parallel_data_in*. Listing 2 shows the VHDL implementation of the 4 bit shift register, and the subsequent figures show the shift register simulation.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

entity shiftreg is
port(
   shiftreg:            out std_logic_vector (3 downto 0);  -- output of shift register
   parallel_data_in:    in  std_logic_vector (3 downto 0);  -- parallel load
   serial_data_in:      in  std_logic;  -- serial data input
   load:                in  std_logic;  -- 1 = load, 0 = no load
   shift:               in  std_logic;  -- 1 = shift left, 0 = shift right
   rotate:              in  std_logic;  -- 1 = rotate left, 0 = rotate right
   enable:              in  std_logic;  -- 1 = Shift, 0 = no shift
   rotenable:           in  std_logic;  -- 1 = rotation, 0 = no rotation
   clock:               in  std_logic;  -- System clock
   reset:               in  std_logic); -- System reset

end shiftreg;
```

```
architecture behavioral of shiftreg is
   signal shiftregtemp :    std_logic_vector (3 downto 0);  -- temporary shift register signal

BEGIN
   operations: PROCESS (clock, reset)

   BEGIN
      IF (reset = '1') THEN

         shiftregtemp <= (OTHERS => '0'); -- resetting the temporary signal
      ELSIF ((clock'EVENT) AND (clock='0')) THEN
         IF (load = '1') THEN
         shiftregtemp <= parallel_data_in; -- loading the parallel data into the temporary signal
         ELSIF  ((enable = '1') AND (shift = '1')) THEN
         shiftregtemp <= shiftregtemp(2 downto 0) & serial_data_in; -- Shifting left
         ELSIF  ((enable = '1') AND (shift = '0')) THEN
         shiftregtemp <= serial_data_in & shiftregtemp(3 downto 1); -- Shifting right
         ELSIF  ((rotenable = '1') AND (rotate = '1')) THEN
         shiftregtemp <= shiftregtemp(2 downto 0) & shiftregtemp(3); -- rotating left
         ELSIF  ((rotenable = '1') AND (rotate = '0')) THEN
         shiftregtemp <= shiftregtemp(0) & shiftregtemp(3 downto 1); -- rotating right
         ELSE
         shiftregtemp <= shiftregtemp; -- Hold and keep same output value
      END IF;

      END IF;

   END PROCESS operations;

shiftreg <= shiftregtemp; -- outputing value
end behavioral;
```

Listing 2: VHDL 4-bit shift register

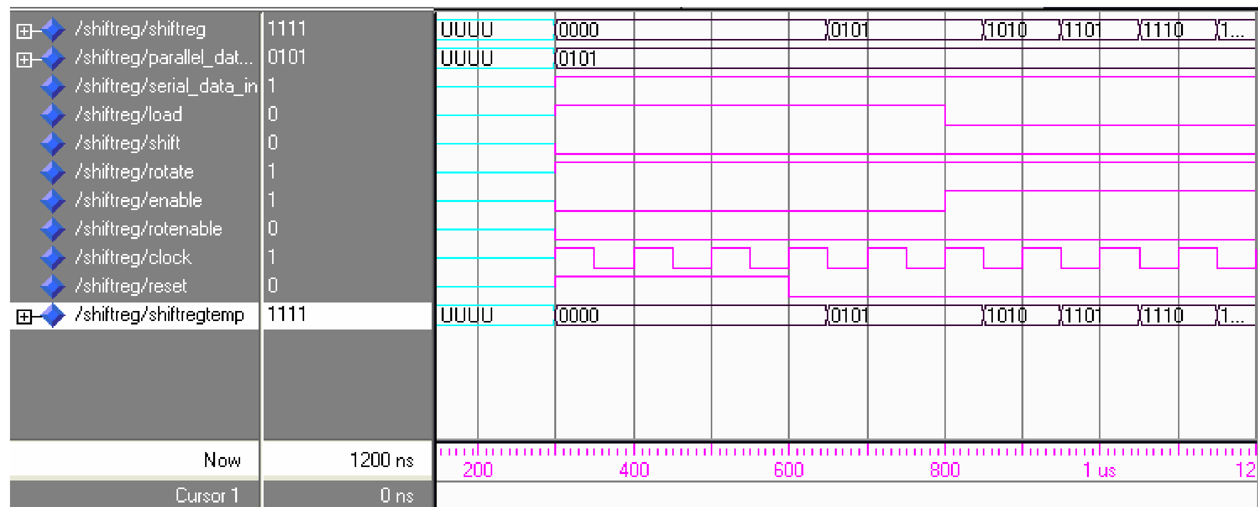


Figure 4: 4-bit Shift Register – Right Shift

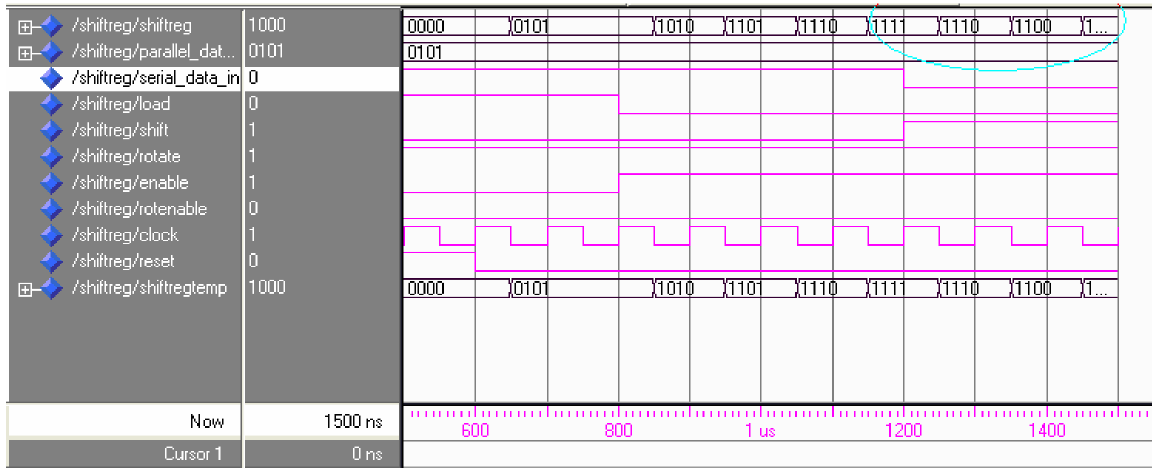


Figure 5: 4-bit Shift Register – Left Shift

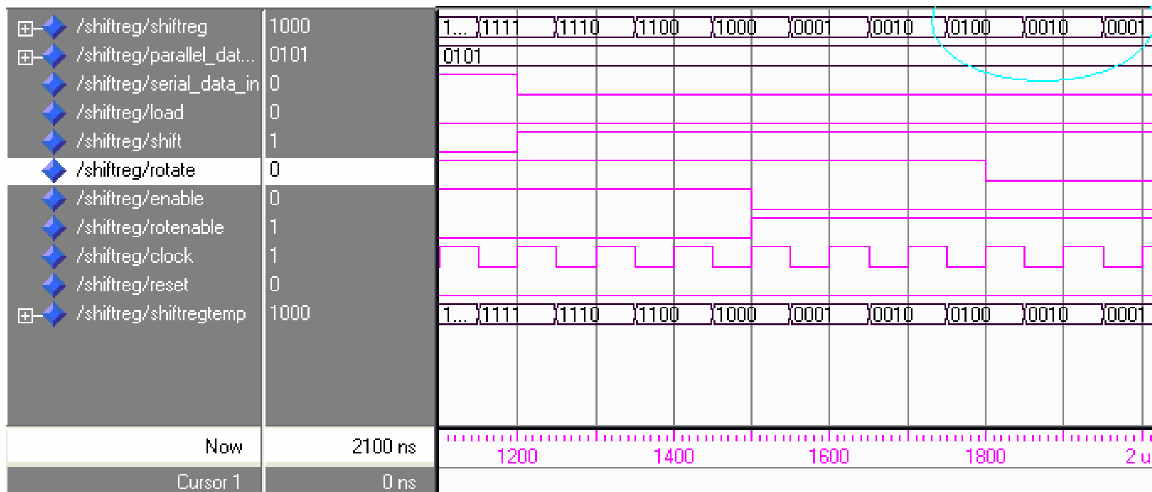


Figure 6: 4-bit Shift Register – Right Rotation

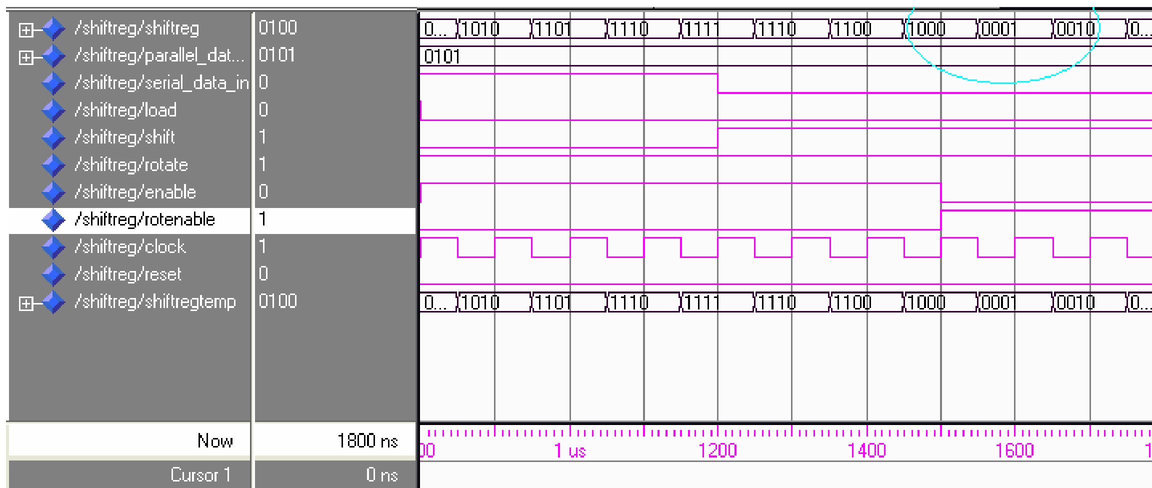


Figure 7: 4-bit Shift Register – Left Rotation

4. Conclusions

Both designed systems, the Counter and the Shift Register worked as expected. The behavior of these systems was observed under the frequency of 100 kHz. However, when a chip is going to be implemented with any of these designs, frequencies should be taken into account for efficiency purposes. The ModelSim Software worked efficiently and without problems for simulation purposes. Both of these components, the counter and register, are very important in the VLSI area since they are often used by companies to implement chips. It was demonstrated that both designs can be increased in size simply by changing the values of a few parameters in the VHDL code. This proves that these types of complex programmable logic devices (CPLD) are suitable for designs that will eventually need to be upgraded. Moreover it shows that a simple 4-bit unit can be implemented and bigger units can be derived from it.

References

[1] Spiegel, J. (1997). "Digital Design Laboratory", [online], Sep. 12, Available WWW: http://www.seas.upenn.edu/~ee201/lab/LabFullAdder/LabFullAdderF01.html
[2] Uyemura, J. (2001). "Introduction to VLSI Circuits and Systems," New York, NY: John Wiley & Sons, Inc.
[3] Skahill, K. (1996). "VHDL for Programmable Logic," Menlo Park, CA: Addison-Wesley Publishing Longman, Inc.

Authorization and Disclaimer

Authors authorize LACCEI to publish the papers in the conference proceedings. Neither LACCEI nor the editors are responsible either for the content or for the implications of what is expressed in the paper.