Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 1. Digital Systems, FPGAs and the Design Flow 5

1.1 Digital Systems 5
1.2 Field Programmable Gate Array (FPGA) 7
1.3 FPGA Internal Organization 9
1.4 Configurable Logic Block 11
1.5 Electronic Design Automation (EDA) and the FPGA design flow 13
1.6 FPGA Devices and Platforms 14
1.7 Writing Software for Microprocessors and VHDL Code for FPGAs 16
1.8 Laboratory Assignment 17

Chapter 2. HDL Based Designs 34

2.1 Theoretical Background 34
2.2 Laboratory Assignment 36

Chapter 3. Hierarchical Design 44

3.1 Hierarchical Design in VHDL 44
3.2 Laboratory Assignment 49

Chapter 4. Multiplexer and Demultiplexer 58

4.1 Theoretical Background 58
4.2 Laboratory Assignment 61

Chapter 5. Code Converters 70

5.1 Arrays of Signals 70
5.2 Seven Segment Displays 74
5.3 Encoders and Decoders 75
5.4 Designing a Seven Segment Decoder 76
5.5 Case Study: A Simple but Fully Functional Calculator 78
5.6 Laboratory Assignment 84

Chapter 6. Sequential Circuits, Latches and Flip-Flops 89

6.1 Sequential Circuits in VHDL – The Process Statement 89
6.2 Describing a D Latch in VHDL 92
6.3 Describing a D Flip-Flop in VHDL 95
6.4 Implementing Registers with D Flip-Flops 98
6.5 Laboratory Assignment 99

Chapter 7. Synthesis of Finite State Machines 103

7.1 Finite State Machines 103
7.2 VHDL Synthesis of Finite State Machines 105
7.3 FSM Case Study: Designing a Counter 109
7.4 Laboratory Assignment 112

Chapter 8. Using Finite State Machines as Controllers 115
8.1 Designing an FSM Based Control Unit 115
8.2 Case Study: Designing a Vending Machine Controller .. 117
8.3 Laboratory Assignment 124

Chapter 9. More on Processes and Registers . 134
9.1 Implicit and Explicit Processes 134
9.2 Designing a Shift Register 137
9.3 Laboratory Assignment 140

Chapter 10. Arithmetic Circuits 143
10.1 Half-Adder, Full-Adder, Ripple-Carry Adder 143
10.2 Laboratory Assignment 151

Chapter 11. Writing synthesizable VHDL code for FPGAs 153
11.1 Synthesis and Simulation 153
11.2 VHDL Semantics for Synthesis 154
11.3 HDLGen - Automatic Generation of Synthesizable VHDL 159

Chapter 6. Sequential Circuits, Latches and Flip-Flops

The first part of this book introduces the concepts related to the design of combinational circuits in VHDL. In the second part of this book, starting from this chapter on, the design of sequential circuits in VHDL is discussed and explained. At the end of the chapter, the students should be able:

- to understand the concepts and principles of sequential circuits;
- to understand the concepts and differences between latches and flip-flops;
- to understand the principles of registers design in VHDL;
- to design and test flip-flops, latches and registers in VHDL;
- to design and implement the proposed case study using an FPGA board.

6.1 Sequential Circuits in VHDL – The Process Statement

In the **combinational circuits** discussed in previous chapters, the results presented in their outputs depend only on their current input values. The outputs of a **sequential circuit**, on the other hand, depend also on previous values internally stored. For this reason, a sequential circuit must be designed making use of some sort of memory resources. In other words, combinational circuits can be built based on Boolean equations only, while sequential circuits must employ also storage resources as latches and flip-flops. The design of latches and flip-flops in VHDL is explored later on in this chapter.

In VHDL, as introduced in previous chapters, all signal assignments are performed in parallel. This language feature is very interesting to describe the behavior of combinational circuits. The behavior of sequential circuits is described in VHDL through the *process* statement. In Figure 6.1 there is an example of process usage in VHDL. Important characteristics regarding the functionality of this statement are listed next:

- A *process* defines a sequence of commands to be performed by the circuit. This means that all the commands listed from line 16 to 19 in Figure 6.1 are performed in the defined order, and one after the other.
- A *process* is cyclic, and it never ends. However, its list of commands (e.g. lines 16 to 19 in Figure 6.1, and lines 17 to 21 in Figure 6.2) is performed only when a change is noticed in one of its activation signals (e.g. C and D on line 14 in Figure 6.1).
- The signals activation list of a process is known as *the sensitivity list*. A process is triggered by new “events” detected on its sensitivity list.
- After the last command of a process has been performed, the first command in the sequence is run again, but only after the next time a change is picked up in one or more signals of the sensitivity list (e.g. line 14 in Figure 6.1).
- Some VHDL constructions can only be used inside a process as, for instance, the IF .. THEN .. ELSE shown on lines 17 to 19 in Figure 6.1.

- Signals cannot be declared inside a process.
- In a process, in case of several assignments for the same signal, only the last assignment will be performed. In Figure 6.2, B will be assigned only on line 18, and C will be assigned only on line 20.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   entity ProcessExample is port (
4       C: in  std_logic;
5       D: in  std_logic;
6       Q: out std_logic
7   );
8   end ProcessExample;
9   architecture behv of ProcessExample is
10      signal A, B: std_logic;
11  begin
12      A <= D;
13      Q <= B;
14      P1: process (C, A)
15      begin
16          B <= '0';
17          if (C = '1') then
18              B <= A;
19          end if;
20      end process P1;
21  end behv;
```

Figure 6.1. Process statement in VHDL.

An important VHDL process concept is that all signal attributions performed will only be valid when the execution reaches the *end process* statement. Two versions of the example in Figure 6.2 have been implemented, and the simulation results are shown in Figure 6.3. In Version 1, the sensitivity list of the process has only the A signal, and in Version 2 it has the A and C signals. The waveforms in Figure 6.3 show the simulation results for Version 1 (top of the figure) and Version 2 (bottom of the figure). In both versions, the *data_in* input receives the values '0', '1' and '0', in this order, and when the simulation starts there is no history of previous values for any of the signals.

In Version 1, when A receives '0' from *data_in*, the process is triggered as it has signal A on its sensitivity list. Next, all the assignments are performed, sequentially. B receives '0' on line 18 in Figure 6.2, overwriting the assignment performed on line 17. C receives '1' on line 20 (*not A*, where A = *data_in* = '0'), overwriting the assignment performed on line 19. On line 21, D receives the value stored in C but, as shown in the top waveform in Figure 6.3, the simulation has just started, and the simulator has no knowledge of the value of C. For this reason, the waveform shows neither '0' nor '1'. At this point, it is important to notice that the C value assigned on line 20 in Figure 6.2 can-

not be used for the D assignment on line 21, as in a process, all assignments are performed sequentially, and considering the values the signals hold when the process started.

```
1  library IEEE;
2  use IEEE.Std_Logic_1164.all;
3  entity TestProc is
4  port(
5       data_in  : in  std_logic;
6       data_out : out std_logic
7  );
8  end TestProc;
9  architecture circuit of TestProc is
10   signal A, B, C, D: std_logic;
11 begin
12   A <= data_in;
13   data_out <= D;
14   -- Version 1      -- Version 2
15   process (A)       -- process (A, C)
16   begin
17     B <= A;
18     B <= '0';
19     C <= A and '1';
20     C <= not A;
21     D <= C;
22   end process;
23 end circuit;
```

sequential statements

Figure 6.2. Sequence of signal attributions in a process.

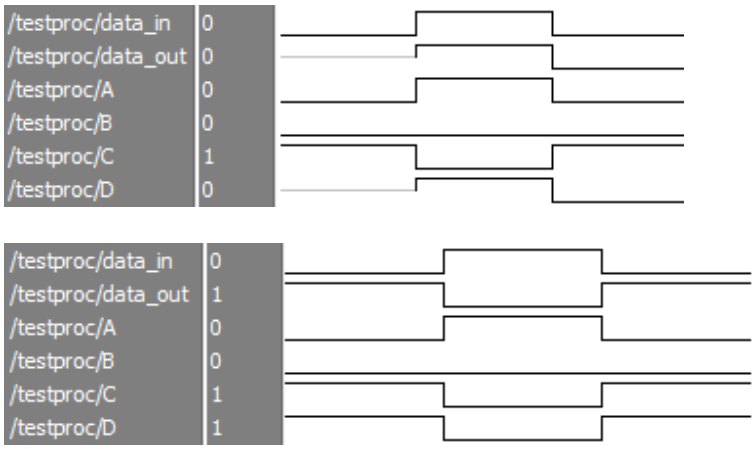


Figure 6.3. Simulation results for versions 1 and 2 of the VHDL code listed in Figure 6.2.

Proceeding with the simulation of Version 1, next A receives '1' from *data_in*, and the process is triggered again as a result of this change in A ('0' to '1'). Once more, all

the assignments are performed, sequentially. B receives '0' on line 18 in Figure 6.2. C receives '0' on line 20 (*not A*, where A = *data_in* = '1'). D receives the value stored in C on line 21, which is '1'. This is the value stored in C, when the process started, and not the value just assigned to C on line 20.

In Version 2, the only difference is the C signal added to the process sensitivity list. This tiny modification results in significant differences in the simulation results. Initially, A receives '0' from *data_in*, and the process is triggered. The assignments are performed sequentially starting by B receiving '0' on line 18 in Figure 6.2, overwriting the assignment performed on line 17. Next, C receives '1' on line 20 (*not A*, where A = *data_in* = '0'), overwriting the assignment performed on line 19. On line 21, D receives the value stored in C and, as there was a change in C, and as C is now on the process sensitivity list, then the process is triggered once more and D also receives '1', which is the new value of C. This simulation result is shown in the bottom waveform of Figure 6.3.

In Figure 6.1 the process has a label (P1 on line 14). This identification is optional, but it is very convenient when performing the simulation of a VHDL code with several processes. The sensitivity list is also optional in a process. Though, in case of processes with no sensitivity list, a *wait* statement must be included in the process body in order to hold the process from time to time.

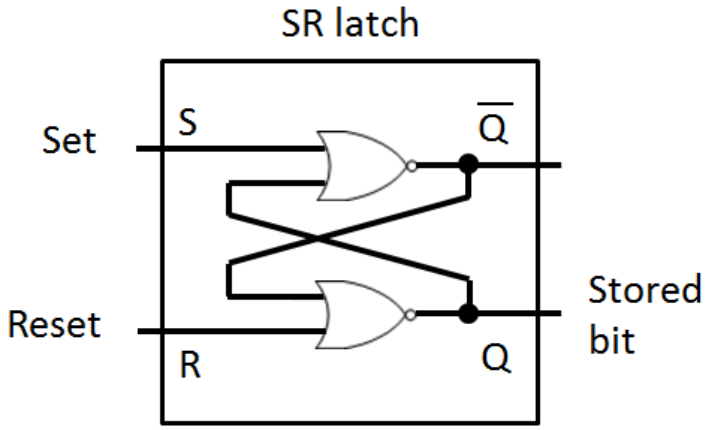
The behavioral description of sequential circuits in VHDL is an important topic, and it will be further discussed in Chapter 9 together with a more detailed discussion of processes.

6.2 Describing a D Latch in VHDL

As mentioned before, the output of a sequential circuit depends not only on its inputs, but also on data previously stored. Consequently, sequential circuits require the implementation of storage resources. Figure 6.4 shows a basic storage element, the set-reset (SR) latch. This simple circuit manages to store data (a single bit) through the cross-feedback loop in the pair of cross-coupled NOR gates shown in Figure 6.4. The stored bit is available on the Q output signal.

The truth table in Figure 6.4 describes the SR latch functionality, which is straightforward: when the S input is '0' and the R input is '1', the circuit outputs '0' ("reset" the circuit, forcing its Q output to '0'); when the S input is '1' and the R input is '0', the circuit outputs '1' ("set" the circuit, forcing its Q output to '1'); when both S and R inputs are equal to '0', the circuit holds its current state (memory!). In all situations, one of the outputs is always the complement of the other. However, when both inputs S and R are '1', the circuit can become unstable and, consequently, this is not a valid input for this circuit.

Two versions for the VHDL implementation of an SR latch are shown in Figure 6.5. The first version, on the left, uses a process, while the second version uses just combinational logics.



S	R	Q
0	0	hold last value
0	1	0
1	0	1
1	1	not allowed

Figure 6.4. A basic SR latch and its truth table.

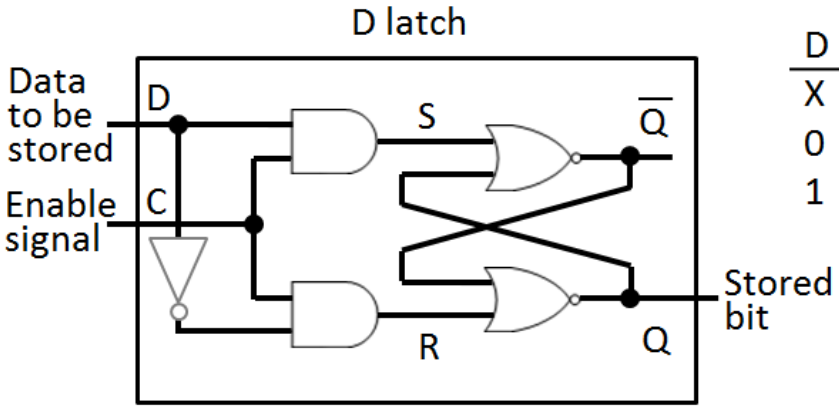
```vhdl
-- Version 1 - Using process
library ieee;
use ieee.std_logic_1164.all;
entity SR_Latch_Proc is port (
    S, R : in  std_logic;
    Q, Qn: out  std_logic
);
end SR_Latch_Proc;
architecture behv of SR_Latch_Proc is
   signal auxQ, auxQn: std_logic;
begin
   process (S, R, auxQ, auxQn)
   begin
      auxQ  <= auxQn nor R;
      auxQn <= auxQ nor S;
   end process;
   Q  <= auxQ;
   Qn <= auxQn;
end behv;
```

```vhdl
-- Version 2 - Combinational circuit
library ieee;
use ieee.std_logic_1164.all;
entity SR_Latch is port (
      S, R : in  std_logic;
      Q, Qn: out  std_logic
);
end SR_Latch;
architecture behv of SR_Latch is
   signal auxQ, auxQn: std_logic;
begin
   auxQ  <= auxQn nor R;
   auxQn <= auxQ nor S;
   Q  <= auxQ;
   Qn <= auxQn;
end behv;
```

Figure 6.5. VHDL implementation for an SR latch.

The SR latch is a good option to show how a basic storage circuit works. However, the “not allowed” input suggests that extra precautions should be taken during this circuit’s usage. Basically, in most applications, the circuit around the SR latch should be designed in order to never allow a ‘1’ value to be present in both S and R inputs simultaneously.

A possible solution for this problem is the D latch, which is an evolution of the SR latch. A D latch is built adding two AND gates and an inverter in front of the SR latch, as shown in Figure 6.6. With this arrangement, the “not allowed” input of the SR latch cannot happen. Now the circuit has just one data input, D. The other input, C, shown in Figure 6.6 is used to control when a new data is stored in the memory (the D latch circuit). While the C input is ‘1’, the input data D is stored. When C is ‘0’, the AND gates are “closed”, and changes on the D input cannot be stored in the circuit (the SR latch).



D	C	Q
X	0	hold last value
0	1	0
1	1	1

Figure 6.6. A D latch and its truth table.

The D latch could be implemented in VHDL following the same strategy as the SR latch shown in Figure 6.5. However, a better and more elegant way of describing a D latch is shown in Figure 6.7. This VHDL implementation is a "behavioral" description of the D latch. The *if* statement on line 13, describes the exact behavior of the D latch truth table, i.e., when C is '1', the input data provided in D is stored. Otherwise, when C is '0', the circuit holds the value provided in D, when C was '1' for the last time. The waveform in Figure 6.7 shows the D latch behavior according to C and D inputs.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_latch is port (
4      C:  in std_logic;
5      D:  in std_logic;
6      Q: out std_logic
7  );
8  end D_latch;
9  architecture behv of D_latch is
10 begin
11     process(C, D)
12     begin
13         if (C = '1') then
14             Q <= D;
15         end if;
16     end process;
17 end behv;
```

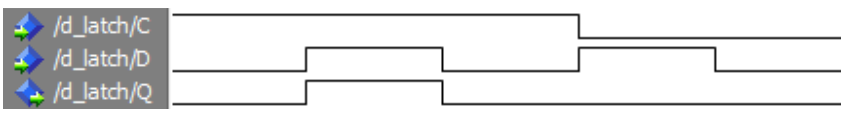
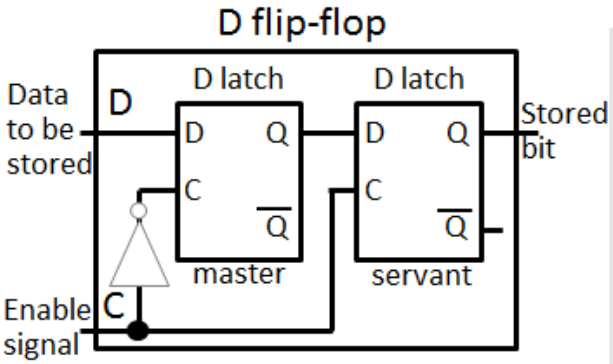


Figure 6.7. VHDL implementation for a D latch.

The D latch is level activated, as the circuit output (Q) follows the value presented at its input (D), while the C input (which is usually connected to the clock) is active, i.e. C is kept at level '1'. This behavior is called "transparent". The advantage of the D latch is its low cost on circuitry. The drawback is that it has no precise control of the input D to output Q. For instance, if a circuit has several D latches in cascade and if C = '1' for a long period of time, the data at the D input might be propagated through many latches. Otherwise, if C = '1' for a too short period of time, it may not enable a store. The terms "long period" and "short period" are dependent on the target technology.

6.3 Describing a D Flip-Flop in VHDL

The D flip-flop is conceived aiming applications where the aforementioned situation of controlling the exact moment when the input data is stored, is an important issue. In a D flip-flop, the input data is stored just when there is a transition from '0' to '1' (or from '1' to '0') on the C input. So, the D flip-flop is said to be controlled by "edge", while the D latch is controlled by "level". In a D latch, whenever the C input is '1' (active high logic level), the D value is stored in the circuit. In a D flip-flop, the D value is stored only when the C input changes from '0' to '1' (a rising edge event). In other versions of these circuits, the D latch stores the input data when C = '0' (active low logic level), and the D flip-flop stores the input data when C changes from '1' to '0' (a falling edge event). The D Flip-flop can be implemented according to different design styles. In a typical design, two D latches are connected in a master-servant topology, as shown in Figure 6.8.



D	C	Q
X	X	hold last value
0	0->1	0
1	0->1	1

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_FF is port (
4    C: in std_logic;
5    D: in std_logic;
6    Q: out std_logic
7  );
8  end D_FF;
9  architecture behv of D_FF is
10 begin
11   process(C, D)
12   begin
13     if (C'event and C = '1') then
14       Q <= D;
15     end if;
16   end process;
17 end behv;
```

Figure 6.8. D flip-flop. Block diagram, truth table, and VHDL implementation.

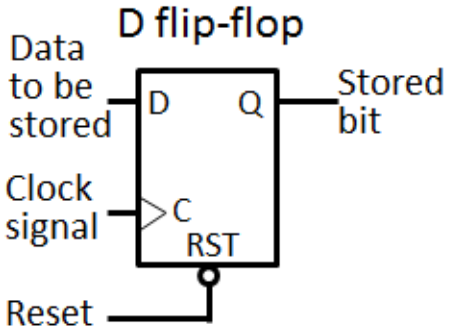
In Figure 6.8, the output (Q) of the master latch is connected to the input (D) of the servant latch. The clock signal for the master latch is inverted in relation to the clock signal for the servant latch. In this sense, the master is loaded when C = '0', and the servant

when C = '1'. When C changes from '0' to '1', the master is disabled and the servant is loaded with the value that was at D just before C has changed. Outside the active clock edge there is no transfer of the input value to the servant latch. As a tradeoff this architecture needs more logic gates to be implemented than the D latch. However, gate count is less of an issue nowadays.

The VHDL code listed in Figure 6.8 is a behavioral implementation for the described D flip-flop. The hardware detection of a rising edge event on the C input is described in VHDL on line 13. A VHDL synthesis tool understands this sort of description and a physical D flip-flop is used in the final circuit, in case this D flip-flop resource is available in the target technology.

There are several variations for the D flip-flop implementation. In Figure 6.9, a reset control signal has been added to the circuit. In this case, when the RST input is zero, the output will also be zero. Otherwise, when RST is '1', then the Q output will depend on the D and C inputs, as explained before. This D flip-flop with asynchronous reset VHDL implementation is shown on line 14. In a synchronous reset implementation, line 14 should be placed inside the *c'event* test.

Notice in Figure 6.9, on line 16, the *elsif* VHDL statement usage. This combination of *if (RST = '0')* and *elsif (C'event and C = '1')*, tells the synthesis tool to generate D flip-flop with asynchronous reset. **Very important!! Any change in this syntax, may result in the synthesis tool generating another sort of circuit, which may not be the expected D flip-flop.**



D	C	RST	Q
X	X	0	0
X	X	1	hold value
0	0->1	1	0
1	0->1	1	1

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_FF is port (
4     C   : in std_logic;
5     RST: in std_logic;
6     D   : in std_logic;
7     Q   : out std_logic
8  );
9  end D_FF;
10 architecture behv of D_FF is
11 begin
12    process(C, RST, D)
13    begin
14       if (RST = '0') then
15          Q <= '0';
16       elsif (C'event and C = '1') then
17          Q <= D;
18       end if;
19    end process;
20 end behv;
```

Figure 6.9. D flip-flop with asynchronous reset.

Figure 6.9 shows the block diagram (symbol) of a D flip-flop with asynchronous reset. In this diagram, internal details as logic gates and latches are hidden, and only the input and output signals are shown. The small triangle on the C input represents a "clock" signal in digital systems. For the sake of simplicity, this symbol and connotation has not been used in the previous flip-flop diagram shown in Figure 6.8. However, usually in flip-flop based designs, the C input is always connected to a clock signal in order to synchronize the data read/write operations with the remaining components of the circuit. Basically, when a component needs to store a bit in a flip-flop, if the component operations are coordinated by the same clock signal as the flip-flop, then it is possible to assure that the correct data is written. In a typical synchronous digital system, operations take place in the rising (or falling) edge of the global clock signal. It is not unusual for current digital systems to present several clock signals, and the implementation of such systems is a challenge for the designers.

Figure 6.10 shows a D flip-flop with asynchronous reset, and an *Enable* input. The VHDL implementation describes the behavior of this flip-flop where, basically, whenever the reset signal is high (RST = '1'), and there is a rising clock edge, if the enable signal is high (EN = '1'), then the input data is stored (Q <= D).

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_FF is port (
4    CLK, RST,
5    EN, D     : in std_logic;
6    Q         : out std_logic
7  );
8  end D_FF;
9  architecture behv of D_FF is
10 begin
11   process(CLK, RST, D, EN)
12   begin
13     if (RST = '0') then
14       Q <= '0';
15     elsif (CLK'event and CLK = '1') then
16       if EN = '1' then
17         Q <= D;
18       end if;
19     end if;
20   end process;
21 end behv;
```

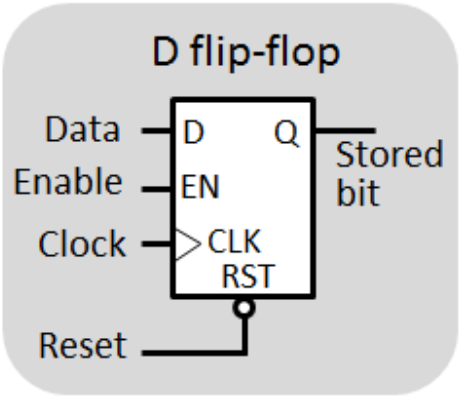


Figure 6.10. D flip-flop with enable and asynchronous reset.

The waveform in Figure 6.11 shows the simulation results for the D flip-flop with enable and asynchronous reset. The CLK input is fed with a clock signal, and the simula-

tion runs for 9 clock cycles. In the first 4 clock cycles, the reset signal (RST) is kept in active-low level ('0'), and no action takes place in the flip-flop. As the reset is asynchronous, the *elsif* statement cannot be reached while RST = '0' (see line 13 in Figure 6.10), and the changes on input signals EN and D have no effect in the circuit state. Next, in the beginning of the 5th clock cycle, the reset goes high, the enable (EN) goes low, and the D input remains high. In this case, as the enable signal is low, again the input data (D) cannot be stored in the flip-flop (see line 16 in Figure 6.10). The enable signal goes high again in the beginning of the 6th clock cycle, simultaneously with the rising clock edge. In this case, there is no change in the flip-flop memory contents, as the enable signal should have been placed in an active-high level, before the rising clock edge event. Finally, in the rising edge of 7th clock cycle, the enable signal was already in active-high level, and the D input, which is '1', is stored in the flip-flop.

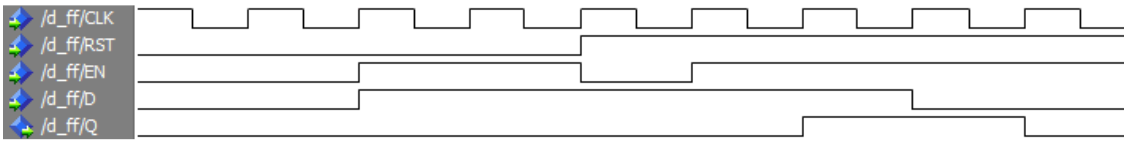


Figure 6.11. Simulation waveform for the D flip-flop with enable and asynchronous reset.

6.4 Implementing Registers with D Flip-Flops

A D flip-flop can be used to store a single bit of data. For circuits that perform operations with larger word sizes, registers should be used instead. A register can be built from an arrangement of D flip-flops, as can be seen in Figure 6.12. Registers with 32 or 64 bits are usually found in commercial microprocessors.

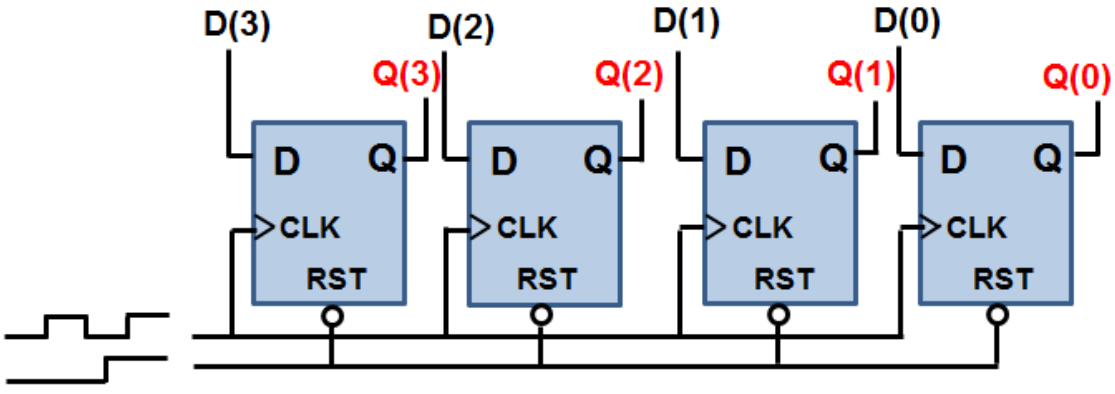


Figure 6.12. Block diagram of a 4-bits register based on D flip-flops.

In the VHDL implementation in Figure 6.13, a 4-bits word present on the D input is stored in the register whenever a rising clock edge is provided in the CLK input. In order to implement a register with enable and asynchronous reset signals, just replace line 17 of the VHDL code listed in Figure 6.13, by lines 16 to 18 of Figure 6.10. It is also necessary, obviously, to add the enable signal declaration in the register entity (*EN: in std_logic;*). Notice that on line 14 of Figure 6.10, the single bit *Q* assignment should be

replaced by *Q <="0000";*, as now Q is a 4 bits vector. In order to have a more generic code, the VHDL statement *Q <= (others => '0');* could be used instead. In this case, all bits of Q will receive '0', no matters Q length.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_4FF is port (
4     CLK:  in std_logic;
5     RST:  in std_logic;
6     D  :  in std_logic_vector(3 downto 0);
7     Q  : out std_logic_vector(3 downto 0)
8  );
9  end D_4FF;
10 architecture behv of D_4FF is
11 begin
12    process(CLK, D)
13    begin
14       if RST = '0' then
15          Q <= "0000";  -- or (others => '0');
16       elsif (CLK'event and CLK = '1') then
17          Q <= D;
18       end if;
19    end process;
20 end behv;
```

Figure 6.13. VHDL description of a 4-bits register based on D flip-flops.

Once more, it is important to notice that synthesis tools are not as smart as one expect them to be. This means that there is almost no flexibility to describe flip-flops and registers in VHDL. The designer who wants a D flip-flop or register properly synthesized, should follow the models shown in this book and in other available documents. Otherwise, the synthesis tool may not generate the hardware the designer wants.

6.5 Laboratory Assignment

The laboratory objectives are:

- to design 4-bits and 8-bits registers with enable and asynchronous reset;
- to add the implemented registers to the calculator developed in the last chapter.

Using as example the registers and flip-flops provided in previous sections, write the VHDL implementations for the 4-bits and 8-bit registers shown in Figure 6.14. Also, make the necessary modifications in the top_calc component as needed to add the registers to the design.

The whole design has a total of 9 files (8 components + top), as follows:

- *c1.vhd*, provided in Figure 5.17;
- *c2.vhd*, provided in Figure 5.18;
- *c3.vhd*, provided in Figure 5.19;
- *c4.vhd*, provided in Figure 5.20;
- *mux4x1.vhd*, provided in Figure 5.21;
- *reg4bits.vhd*, should be adapted from Figure 6.10 and Figure 6.13;
- *reg8bits.vhd*, should be adapted from Figure 6.10 and Figure 6.13;
- *decod7seg.vhd*, implemented in the previous chapter, according to Section 5.4 and Figure 5.12;
- *top_calc.vhd*, partially provided in Figure 5.16. The previous chapter implementation should be adapted in order to add the new registers, and also the new input signals shown in Figure 6.14.

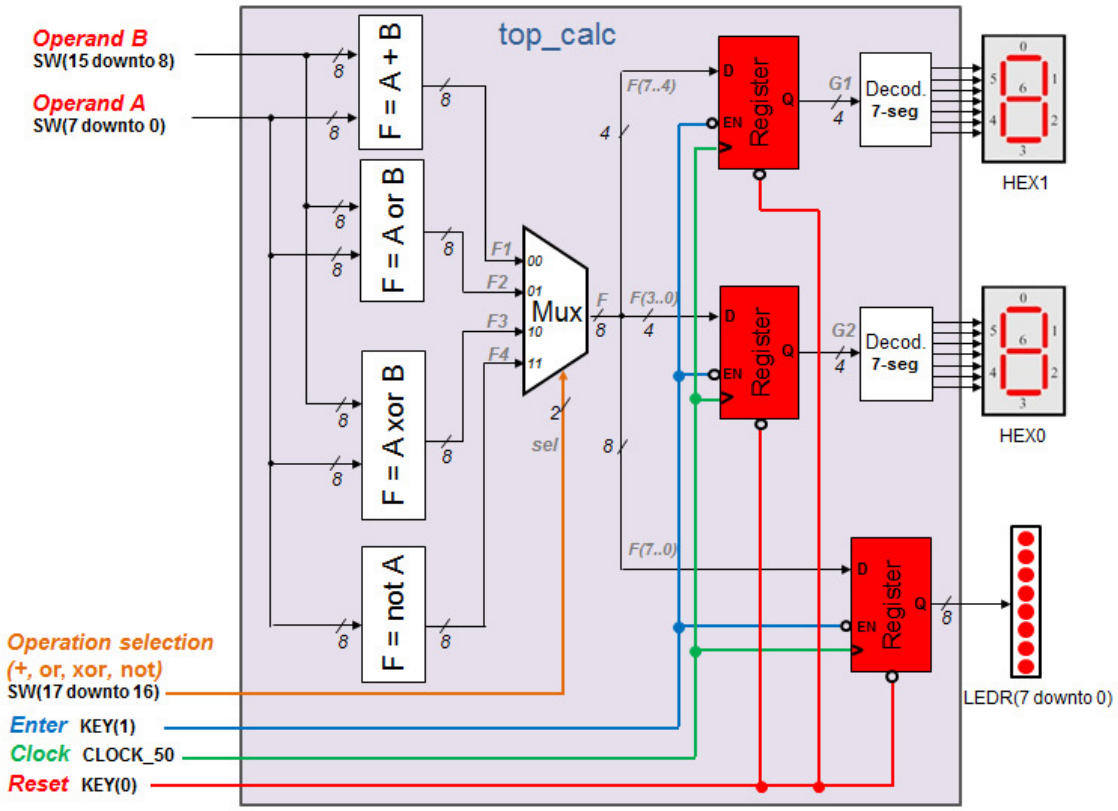


Figure 6.14. Inserting registers in the calculator to store operation results.

The tasks to be completed in this laboratory session, using Altera's Quartus II are as follows:

- Create a project with all the components shown in Figure 6.14;
- Using the VHDL design entry editor, create a 4-bits register and a 8-bits register, both with an enable signal, and asynchronous reset;

- Modify the top-level component developed in the previous chapter, in order to add the new components (the three registers shown in Figure 6.14). Notice that new signals are needed to connect the register outputs, to the decoder inputs.
- Perform the synthesis;
- Perform the simulation and fix errors, if any;
- Prototype and test the circuit in the FPGA board.

The new registers used to store the calculator results are as follows:

- A 4-bits register to store the Less Significant Bits (LSB) of the operation result, to be presented in HEX0;
- A 4-bits register to store the Most Significant Bits (MSB) of the operation result, to be presented in HEX1;
- An 8-bits register to store the operation result to be shown in binary on the red LEDs.

Figure 6.15 shows the user interface for the new calculator. The calculator's operation is straightforward:

- When KEY(0) is pressed (Reset), the flip-flops (registers) should be cleared. As a result, the LEDs are switched off, and the 7-segments displays (HEX0 and HEX1) should show a '0'.
- When KEY(1) is pressed (Enter), the registers are enabled for writing, allowing the operation result to be stored (memory!).

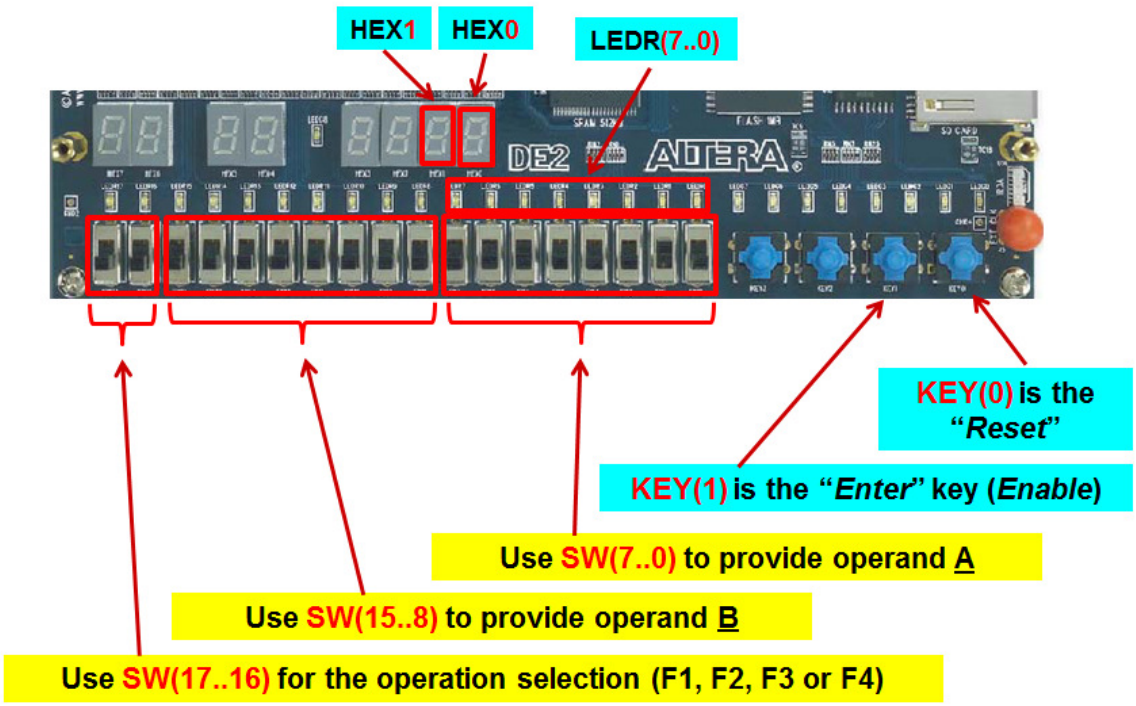


Figure 6.15. Input and output user interface for the calculator.

Additionally, make sure to connect the clock input to the CLOCK_50 signal provided by the DE2 board. This is a 50 MHz crystal available on the board. The top entity should be changed in order to include the new DE2 input signals:

- key: in std_logic_vector(1 downto 0); – – KEY(0) e KEY(1)
- clock_50: in std_logic; – – clock 50 MHz

Also, it is important to notice that the four push buttons available in DE2 board are active low. This means that when they are pressed, a zero (‘0’) is generated. In addition, there is a Schmitt Trigger based debounce circuit for the push buttons, in order to help to have just one zero pulse each time a button is pressed.

As stated before, the **project creation** process has been well discussed and explained in previous chapters. For the **synthesis** activity, it is essential to import the *DE2_pin_assignments.qsf* file, as discussed in Chapter 2, Step 5, and shown in Figure 2.7.

The **simulation** activity is performed using ModelSim. To start the simulator in Quartus II, select *Tools −> Run Simulation Tool −> RTL Simulation*. Wait for ModelSim to open and perform the steps described in previous chapters.

When the expected results are obtained in the simulation process, the next step is the **FPGA prototyping**. As described in previous chapters, to download the generated bitstream to the FPGA, using Quartus II choose menu *Tools −> Programmer*. Follow the instructions provided in Chapter 1, Step 5, observing the “hardware setup” instructions in Figure 1.32 and in Figure 1.33. Test the circuit in the FPGA board, using the switches and keys shown in Figure 6.15. The operation result is shown in binary in $LEDR_{7..0}$, and in hexadecimal on the 7-segment displays HEX0 and HEX1.