CME341 HDL Class Notes

Dr. Eric Salt

Originally created over 1 year ending September 8, 2016
Latest revision: September 3, 2021

Revised on:

October 2, 2016:	added verilog 2001 synthesis directive for full case
September 5, 2017:	Revised chapter 1 up to end of demo. Updated procedures for the new version of Quartus, Quartus Prime, and the new version of modelsim-altera, Modelsim-Altera 10.5b. Also removed references made to the old DE2 boards.
September 6, 2017:	Corrected a few typos
November 7, 2017:	Extended these typed notes to include the first part of the section on Sequential Circuits. The remainder of the notes are in the hand written version of Chapter 3
December 29, 2017:	Added instructions on saving and restoring the wave window in modelsim.
September 4+5, 2020:	Initial modifications for SystemVerilog support (Brian Berscheid)
September 21, 2020:	Fixed a few minor typos in Chapter 3. (Brian Berscheid)
September 3, 2021:	Added instructions to generate timing simulation netlist in Quartus 20. (Brian Berscheid)

Contents

Chapter 1

Getting Started with an Overview

1.1 Introduction

Designing a digital circuit is almost always divided into several distinct tasks. The flow chart shown in Figure 1.1 illustrates the general design process (also referred to as design flow). The boldface text represents the step in the design process and the normal text shows the tools used for that step. One of these tasks (block 4) involves synthesizing circuits with HDL tools. This class concentrates on developing skills for synthesizing circuits with Verilog HDL[1]. It also introduces some techniques that are used in the last three blocks.

The process described in Figure 1 is simplified but captures the essence of a practical process. The chart indicates that the design and synthesis is completed before the circuit is debugged in the simulator. Clearly the synthesis must be completed before the debugging can be completed. However, one of the tasks in the debugging process is usually completed before the circuit is designed and synthesized. That task is writing the HDL that generates the signals that will be used to excite the design when it is being debugged.

[1]There have been numerous versions of the Verilog language (Verilog-95, Verilog 2001, Verilog 2005, etc.) over the last 25+ years. The most recent versions are known as SystemVerilog and have introduced some powerful new capabilities, especially for testing digital circuits. The vast majority of this class is focused on features common to Verilog and SystemVerilog, and the languages are often collectively called "Verilog". Following this convention, in these notes, the terms "Verilog" and "Verilog HDL" should be interpreted to represent the entire family of languages, including SystemVerilog. Where features unique to SystemVerilog are discussed, the term "SystemVerilog" will be used to refer exclusively to the SystemVerilog versions of the language.

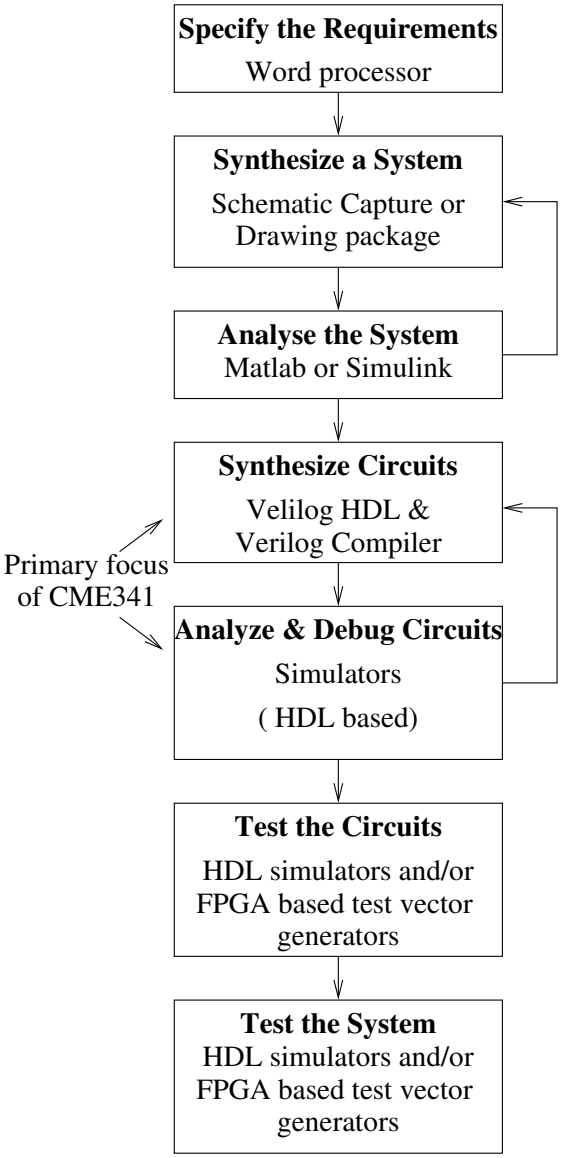


Figure 1.1: Design Process

1.2 Wiring

1.2.1 Motivation

Most engineering students have looked inside a personal computer or stereo amplifier and have seen the printed circuit boards that hold electronic components and the printed tracks that act as the wires connecting the components together. While they are not visible, there are wiring tracks inside Integrated Circuits (ICs) that connect the electronic components fabricated inside the IC.

Originally wiring plans for integrated circuits, for example first generation microprocessors, were in the form of schematic diagrams. Now the wiring plans for ICs are done with an HDL like Verilog HDL.

FPGAs have been steadily increasing in size while their cost per gate has been decreasing. The size of circuit at which they become cost competitive with ASICs (Application Specific Integrated Circuits) is steadily increasing. While FPGAs are in themselves ICs, they are special in that they are programmable. They contain thousands of a few different types of circuits (like "look-up-tables" and "flip/flops") that can be interconnected by programming the wiring connections, which by the way can be done in the field. The wiring plans for FPGAs are derived from the HDL.

The purpose of this section is to show how the Verilog HDL is used to specify the logical wiring connections. (The physical wiring plan is generated by a separate software program called a router. The router uses the output of the Verilog compiler to make the physical wiring plan.)

1.2.2 Block Diagrams in Schematic Diagram Form

Pulse Width Modulator

To explain how wiring information is embedded in schematic diagrams an example is used. The example is a pulse width modulator circuit. There are two inputs: a square clock and a 10 bit unsigned binary number. The output is a pulse train with the width of the pulse determined by the 10 bit input. (in other words the output is a periodic rectangular wave with a controlled

duty cycle). The inputs and outputs as well as a simple timing diagram are shown in Figure 1.2.

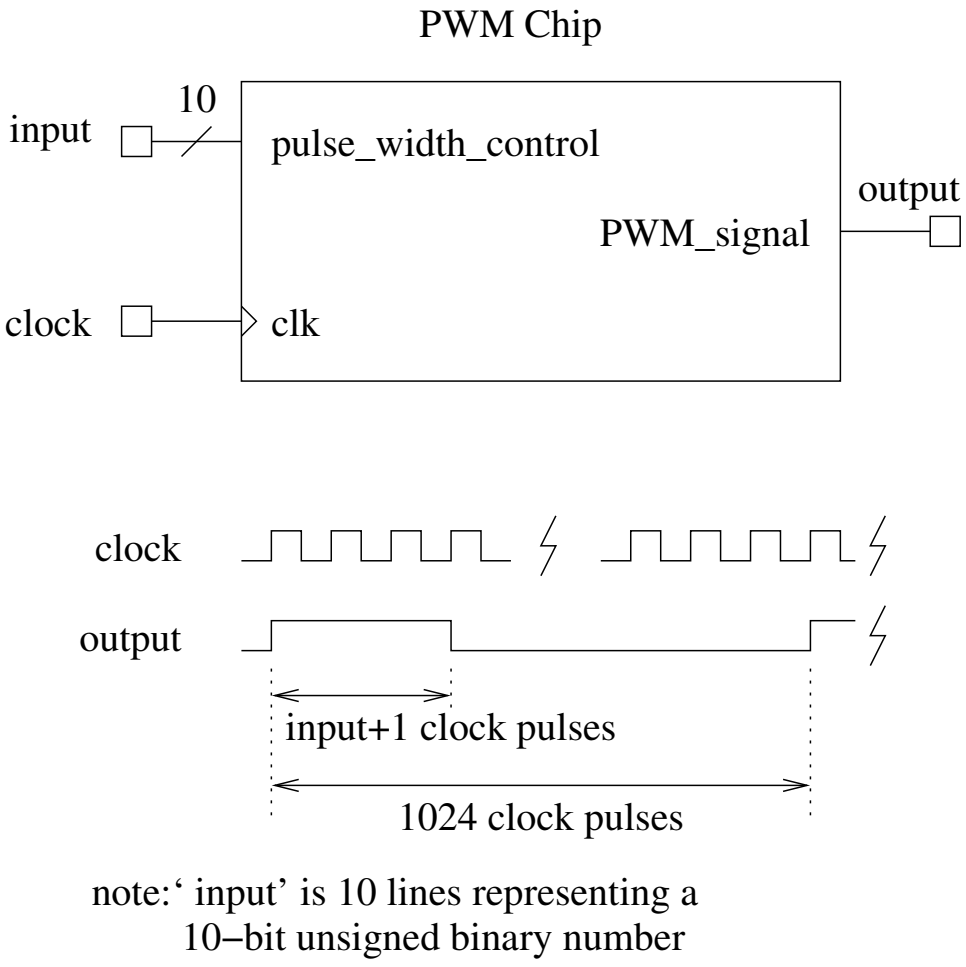


Figure 1.2: Inputs and Outputs for a pulse width modulator

Block Diagram of PULSE WIDTH MODULATOR

A Block diagram for a pulse width modulator circuit is shown in Figure 1.3. The master clock is called clk. It drives a 10-bit counter that runs continuously—counting from 000H to 3FFH then rolling over to 000H and so on.

The pulse width control word is loaded into a register (reg_1) with the same clock edge that rolls counter_1 to 000H. This is done by enabling the reg_1 when counter_1 is all ones (i.e. counter = 3FFH).

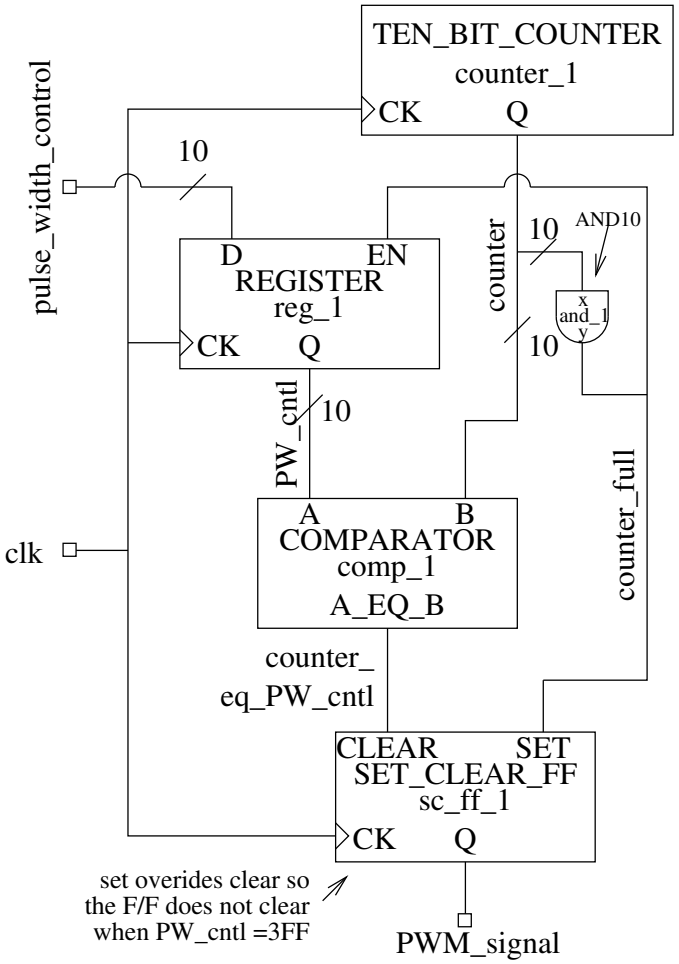


Figure 1.3: Block Diagram of Pulse Width Modulator

The output, PWM_signal, is always set (i.e. forced high) with the same clock edge that rolls counter_1 to zero. The PWM signal is cleared "PW_cntl + 1" clock pulses later. Therefore, if PW_control = 000H then PWM_signal is high for 1 clock period. If PW_cntl = 3FFH, then PWM is not cleared and is high for 1024 clock periods.

Note: PWM is not cleared when PW_cntl = 3FFH. The reason is that a set/clear flip/flop circuit (i.e. sc_ff_1) is designed to have 'set' override 'clear'.

Symbols & Conventions used in Schematic Diagrams

Edge sensitive inputs

The convention shown in Figure 1.4 is used when indicating an input is edge sensitive.

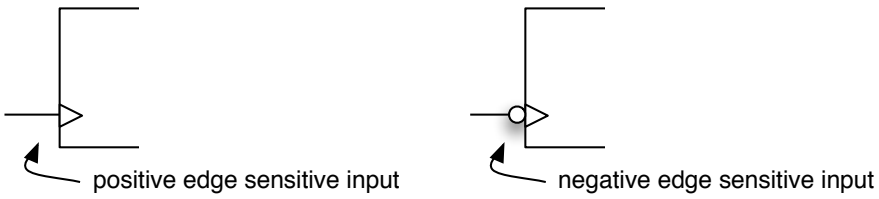


Figure 1.4: Edge Sensitive Inputs

Wire Connections

Figures 1.5 and 1.6 show examples of connected and unconnected wires in a schematic diagram.

Figure 1.5: Connected Wires

Figure 1.6: Wires with no connection

Pins (input/output pins)

Input and output pins are **always** square. Input pins must be oriented such that the wire attaches to the bottom or the right side of the square (see Figure 1.7). Output pins are indicated by attaching the wire to the top or left

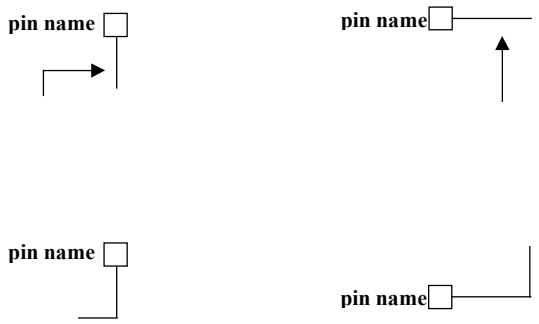


Figure 1.7: Input Pins

side of the square (see Figure 1.8).

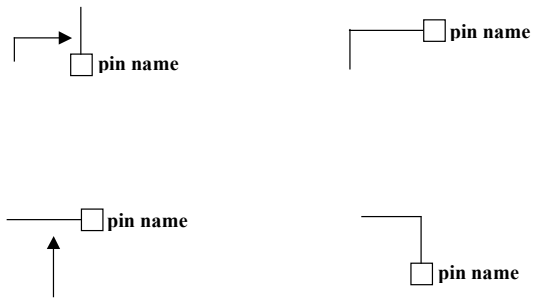


Figure 1.8: Output Pins

On-Page connections

Line connectors are used to save the clutter that arises from drawing several long lines on a page (these lines represent the wires). The wire leaving a source is connected to a triangular symbol that looks like the tip of an arrow as shown in Figure 1.9 The arrows always point in the direction of power flow. The signal source connects to the base of the line connector arrow. The signal sink connects to the tip of the line connector arrow.

Figure 1.9: On-page Line Connectors (source on the right and sinks on the left)

Note: There one source can be connected to several sinks.

Off-Page connections

Off-page connectors, as shown in Figures 1.10 and 1.11, are used to indicate the connection (i.e. the wire) extends to another page in the schematic diagram. Off-page connectors are shown in Figures 1.12 and 1.13 Again the arrow tip

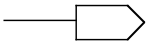


Figure 1.10: An output (source) off-page connector.

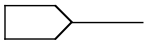


Figure 1.11: An input (sink) off-page connector.

part of the connector points in the direction of power flow.

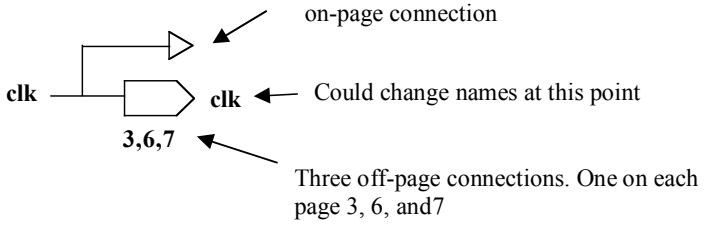


Figure 1.12: Line Connectors Example 1

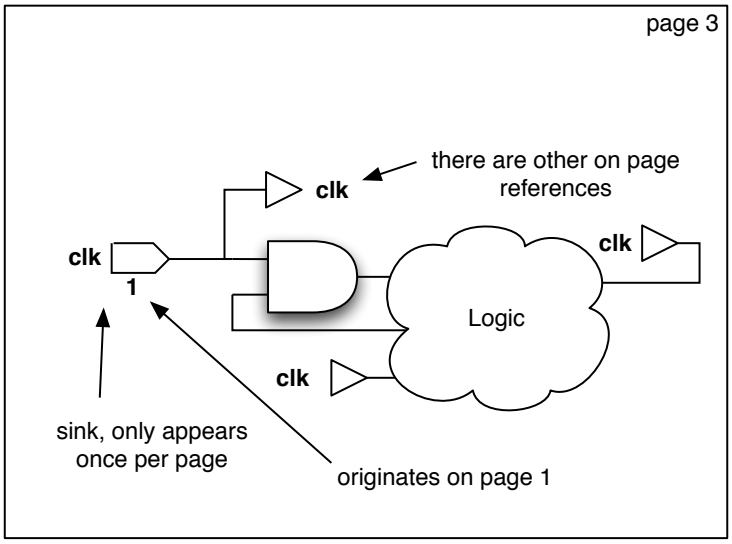


Figure 1.13: Line Connectors Example 2

1.2.3 Describing Block Diagrams in Verilog HDL

A Verilog description of the Pulse Width Modulator (shown in **Figure 1.3**) is given below

```
module PWM_chip (clk, pulse_width_control,  PWM_signal);
input clk;
input [9:0] pulse_width_control;
output PWM_signal;

wire counter_full, counter_eq_PW_cntl;
wire [9:0] PW_cntl, counter;

REGISTER reg_1(.D(pulse_width_control),
               .CK(clk),
               .EN(counter_full),
               .Q(PW_cntl));

TEN_BIT_COUNTER counter_1(.CK(clk),
                          .Q(counter));

COMPARATOR comp_1(.A(PW_cntl),
                  .B(counter),
                    .A_EQ_B(counter_eq_PW_cntl)
                    );

AND10 and_1(.x(counter),
            .y(counter_full)
             );

SET_CLEAR_FF sc_ff_1(.clear(counter_eq_PW_cntl),
           .set(counter_full),
           .CK(clk),
           .Q(PWM_signal));
```

```
endmodule
```

- **REGISTER** is a prototype definition. It defines the circuit in sufficient detail for it to be built. There will be a Verilog module called **REGISTER** in a separate file that completely defines its operation in terms of its inputs and outputs.
- **reg_1** is the name of one instance of **REGISTER**. That is to say, **reg_1** is the name of a hardware circuit that implements the functionality described in the Verilog module called **REGISTER**. The Verilog module called **REGISTER**, i.e. the definition of the prototype called **REGISTER**, can be used to make several identical circuits. When that is done each must be given a different instance name.
- The text **.Q(PW_control)** specifies that the Q of the instance **reg_1** is connected to a wire called pw_control

1.3 Block Design with schematics and Verilog prototype specification

1.3.1 Synchronous Set/Clear Flip/Flop

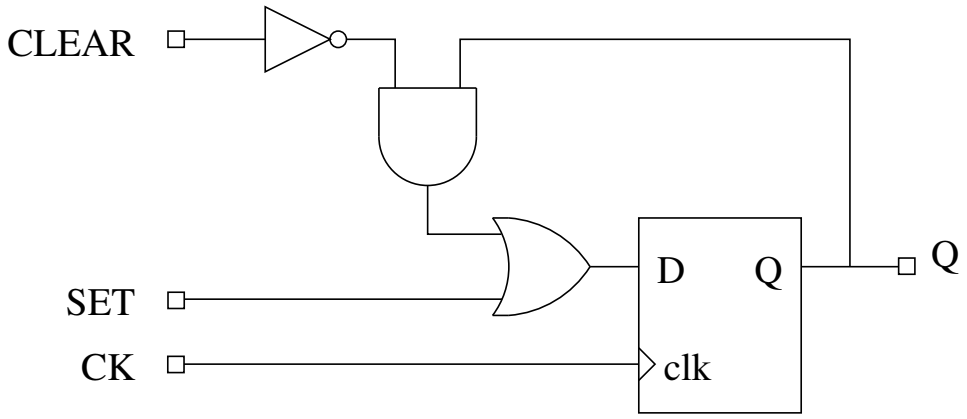


Figure 1.14: Set/Clear Flip/Flop with 'set' over riding 'clear'

In the above circuit, inputs and outputs are shown as pins on a chip. The behavior of the circuit of **Figure 1.14** is as follows:

- If set and clear are both low then D = Q and the f/f is loaded with the same value as it had before.
- If set = 1 then D = 1 and the f/f is set on the next clock edge regardless of "clear" and "Q".
- If clear = 1 and set = 0, then regardless of Q, D = 0. Therefore the f/f is cleared on the next clock edge.

A Verilog description for the synchronous set/clear flip/flop is

```
module SET_CLEAR_FF(CK,CLEAR,SET,Q);
input CK, CLEAR, SET;
output Q;
reg Q;
always @(posedge CK)
if (SET == 1'b1) Q => 1'b1;
else if (CLEAR == 1'b1) Q => 1'b0;
else Q => Q;
```

```
endmodule
```

1.3.2 Comparator

A 10-bit comparator is a simple circuit. The A_EQ_B output is asserted if and only if the 10 bits of input B are equal to the corresponding 10 bits of input A. The schematic diagram is shown in Figure 1.15.

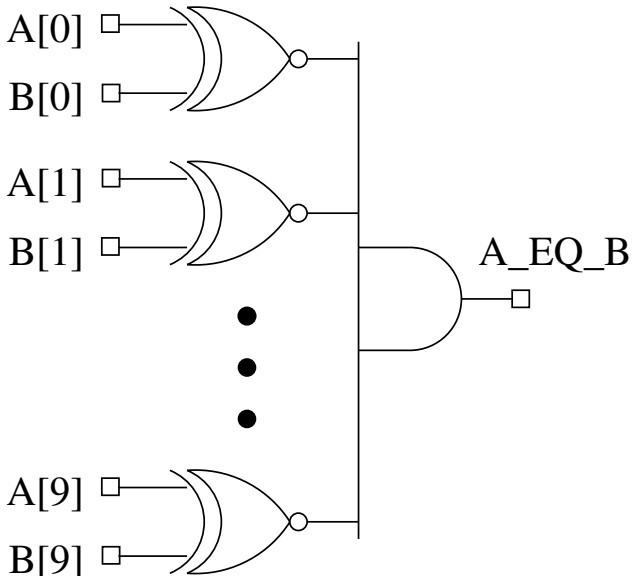


Figure 1.15: 10-Bit comparator

A Verilog description for the comparator is

```
module COMPARATOR (A, B, A_EQ_B);
input [9:0] A, B;
output A_EQ_B;

assign A_EQ_B = &(A~^B);

endmodule
```

1.4 Testing

Approaches:

- Build and test with lab equipment (only an option for FPGA or discrete circuit boards)
- Build and test using special FPGA circuits to excite the circuit under test
- Simulate the circuit under test. Also need to write an HDL that will generate the excitation signals for the circuit under test.

1.5 Specification of a Prototype

In Verilog the word prototype is used to mean "definition of a circuit", which is equivalent to a schematic diagram. Prototypes are specified (i.e. described in Verilog HDL) inside keywords'module - endmodule'. Prototypes can be specified in two different styles or manners: in a structural manner or a behavioral manner.

A structural description is one that indicates exactly how functional blocks are connected to make the system work. A block diagram, which is a schematic diagram at the system level, is a structural description of a system. In large systems each block in the system block diagram could be viewed as a subsystem that could be described by a block diagram. There could be several hierarchical layers of block diagrams. However many hierarchical levels there may be, the lowest level blocks must be described in terms of basic circuit elements like 'inverters', 'and' gates and 'or' gates. These lowest level blocks are therefore described by schematic diagrams that show the basic circuit elements are how they are connected. These schematic diagrams are structural descriptions of the bottom level blocks.

In Verilog, fundamental circuit elements called primitives must be explicitly connected. Therefore, any circuit described in Verilog HDL by primitives must be a structural description. The set of primitive include: not, and, nand, or, nor, xor and xnor gates.

1.5.1 Primitives

Primitives are low level logic functions built into the Verilog language. They are instantiated like prototypes (i.e. like blocks in a block diagram), but don't have to be defined.

The port connections for primitives are made by position association. The output is always the first (i.e. on the left of) the port list. For example the Verilog statement

```
or or_gate_1(output, input1, input2);
```

places a two input or gate in the circuit. It connects the output of the or gate

to a wire named ‘output’ and connects the two inputs to wires called ‘input1’ and ‘input2’.

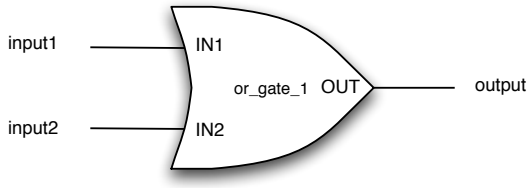


Figure 1.16: or Gate Instance

Additional primitives (more to the point these are pseudo primitives) available through Quartus and associated syntax can be found by typing “primitive” in the help index of QUARTUS (see Figure 1.17).

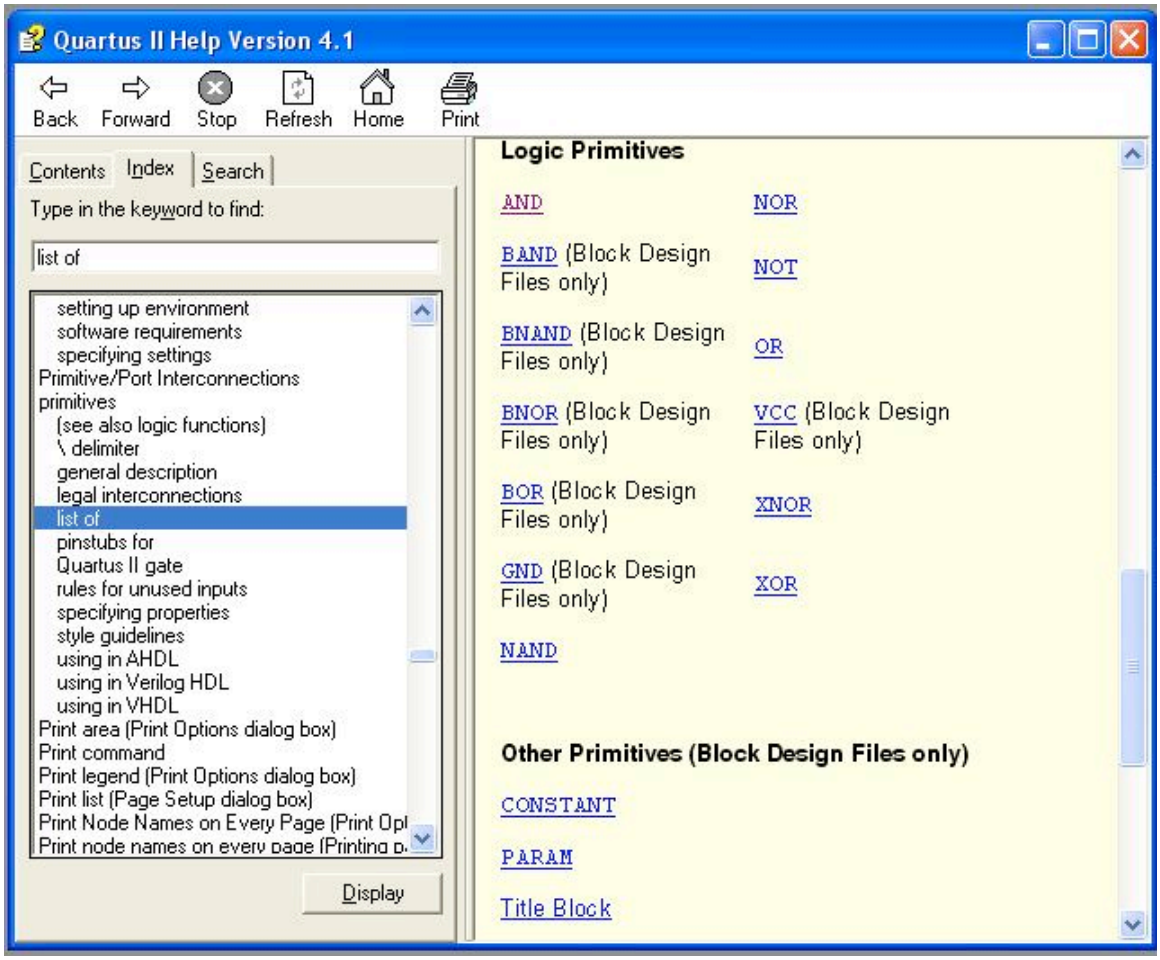


Figure 1.17: QUARTUS Primitive Help File

Example: 4-Bit Comparator

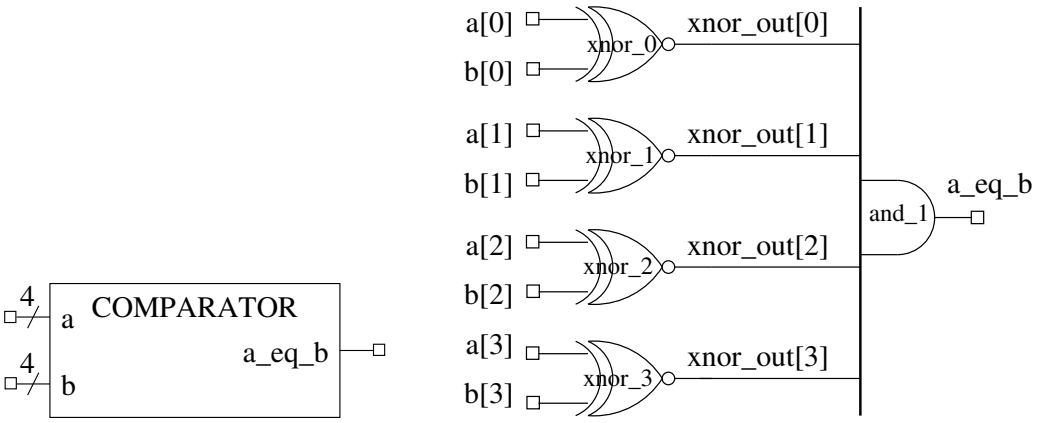


Figure 1.18: 4-Bit Comparator Pin-Out and Schematic

The schematic diagram given in Figure 1.18 is called an explicit structural description. It is called structural because the implementation is defined by specifying the elements to be used and the connections between them.

All of the gates used in the schematic of Figure 1.18 are Verilog primitives. The equivalent explicit structural description in Verilog is:

```
module comparator(a, b, a_eq_b);
input [3:0] a, b;
output a_eq_b;

wire [3:0] xnor_out;

xnor xnor_0(xnor_out[0], a[0], b[0]);
xnor xnor_1(xnor_out[1], a[1], b[1]);
xnor xnor_2(xnor_out[2], a[2], b[2]);
xnor xnor_3(xnor_out[3], a[3], b[3]);
and and_1(a_eq_b, xnor_out[0], xnor_out[1], xnor_out[2], xnor_out[3]);

endmodule
```

1.5.2 The assign statement

The assign statement is used to define an output in terms of boolean logic. The symbol used for the 'exclusive-nor' function is ~^ and the & is used for the 'and' function. When a circuit is described by a boolean equation it built from and/or/ex-or gates in the way specified by the expression. Therefore circuits expressed as boolean equations are called implicit structural descriptions.

An implicit structural Verilog HDL description for the comparator is

```
module comparator(a, b, a_eq_b);
input [3:0] a, b;
output a_eq_b;

wire [3:0] xnor_out;

assign xnor_out  =  a ~^ b;
assign a_eq_b = &xnor_out;

endmodule
```

Note: When an operator is placed between two vectors it builds multiple two-input one-output gates. One gate for each signal (each bit) in the vector. An operator so placed is called a bit-wise operator. In this case the xnor operator ~^ is a bitwise operator. **Note:** The expression '&xnor_out' is equivalent to '(xnor_out[0] & xnor_out[1] & xnor_out[2] & xnor_out[3])'. When an operator, in this case the and symbol &, is placed in front of a vector it is called a reduction operator. It builds a multiple input - single output gate where the inputs are the individual signals in the vector.
Note: The two assign statements could have been combined, in which case the xnor_out wire would not have to be defined. The Verilog description would simplify to

```
assign a_eq_b = &(a ~^ b);
```

The following logical operations can be used in assign statements

```
~     inverter (not) (reduction only)
&     and
~&    nand (reduction only)
|     or
~|    nor (reduction only)
^     xor
~^    xnor
^~    xnor
```

1.6 Building Fundamental Circuits

1.6.1 Schematic and Verilog description for a set_clear latch

The schematic diagram for a set_clear latch is given in Figure 1.19.

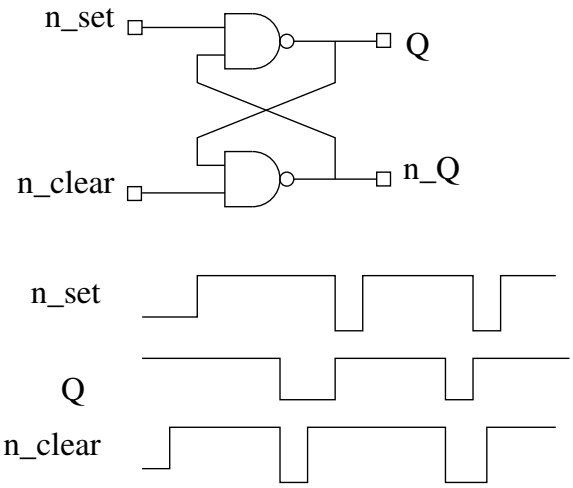


Figure 1.19: Set/Clear Latch

The Verilog description for a set_clear latch (or set_clear flip/flop) follows.

```
module set_clear_latch(n_set, n_clear, Q, n_Q);
input n_set, n_clear;
output Q, n_Q;

nand nand_1(Q, n_set, n_Q);      //instantiate nand_1
nand nand_2(n_Q, Q, n_clear);    //instantiate nand_2

endmodule  // note endmodule is one word and
           // line does not end in a semicolon
```

This is an explicit structural description using the primitive 'nand'. The connection is determined by position (for primitives the output is always the left most in the list).

1.6.2 Schematic diagram and Verilog description for a transparent latch

The schematic diagram for a transparent latch is given in Figure 1.20

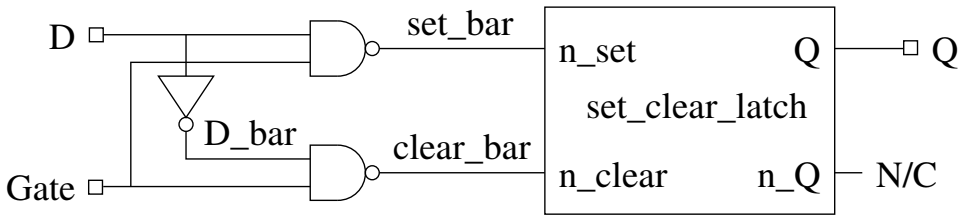


Figure 1.20: Transparent Latch

The Verilog description of a transparent latch uses a 'set_clear' latch. The prototype for the 'set_clear_latch' can be and usually is in another file. This file must be included into the 'transparent_latch' project. This is done by selecting:
Project → Add/Remove Files in Project... and adding the path of the file you wish to add.

A Verilog description for the transparent latch follows.

```
module transparent_latch(D, Gate, Q);
input D, Gate;
output Q;

wire D_bar, set_bar, clear_bar;

assign set_bar = ~(D & Gate);
assign clear_bar = ~((~D) & Gate);

set_clear_latch latch_1(.n_set(set_bar),
                        .n_clear(clear_bar),
                        .Q(Q)); // n_Q output not used

endmodule
```

1.7 Compiling a Verilog HDL Description Using Quartus

The operation of the Quartus compiler will be demonstrated using a set_clear latch built from two nand gates. The instructions parallel that of assignment 2 where the students are asked to build, compile and simulate a set_clear latch built from two nor gates. The difference is the inputs are active low for the 'nand gate' implementation and active high for the 'nor gate' implementation.

1. Open Quartus. If you are using a university computer, this may be done by selecting **start** → **all programs** →**electrical engineering** → **Quartus** (perhaps Quartus Prime). On a personal computer, the process will be similar, but the exact folder structure within the start menu may be slightly different.

   There will likely be two items named **Quartus** one will be a folder and the other executable. Select the executable. It will have a blue icon.

2. A "Getting Started" window should appear. In that window select "New Project Wizard". Alternatively, pull down the file menu and select "New Project Wizard".

3. Read the introduction page for some general information and then click Next to move to page 1 of the New Project Wizard. Choose/create an appropriate directory for this project. The project must be in a folder on one of your system drives, preferably the H-drive if you are working at the university. To set up a folder click on the ellipses on the right hand side of the entry line for the project name. After clicking on the ellipses select the appropriate drive (H: is assumed in the rest of these instructions).

   Name the project folder "set_clear_latch". Give the project and the top-level entity the same name, which is "set_clear_latch". Click Next.

4. The "Project Type" page should be displayed. Select "Empty project", then click next.

5. The "Add Files" page should be displayed. Since we are starting from scratch, there are no files to add. Just click next.

6. The "Family, Device and Board Settings" page should appear.

   (a) Pull down the selection list for the 'family' box (located inside an area referred to as "Device family". Select Cylcone IV E, which is the FPGA family used on the DE2-115 development board. (ASIDE: the DE-115 board is used in the labs in this class and also in the DSP technology stream labs.)

   (b) In the "Target Device" area select the radio button to assign a specific device.

   (c) In the "available devices" area scroll down and select device **EP4CE115F29C7**, which is the part number of the FPGAs on the DE2-115 printed circuit board

   Now click next.

7. The "EDA Tool Setting" page should be displayed. On the line where the tool type is simulation, select "modelsim-altera" for tool name and "Verilog HDL" for format.

   Then click next.

8. The "Summary Page" should be displayed. This page summarizes the settings for the project. Read through it carefully to make sure there were no typos in entering the settings.

   Then click finish.

9. From the Quartus window select "Assignments" from the top menu, then "Settings". A window will pop up. Under the "EDA Tool Settings" item in that window, click on "Simulation". The right pane of the window will now contain a button which says "More EDA Netlist Writer Settings...". Click on that button and another window will pop up. Make sure the "Generate functional simulation netlist" setting in that window is set of "Off". (Some recent versions of Quartus have changed the default to "On", which may cause problems when simulating according to the directions in the following sections.) If the current setting is "On", change it to "Off" to force the creation of a timing simulation netlist.

10. From the Quartus window select "file" from the top menu, then "new". This will open a window. In that window under "design files", select "SystemVerilog HDL file". Click OK. This will open an editor window.

11. In the editor window, write an explicit structural description for the 'nand gate' set/clear latch shown earlier in the notes and repeated below.

```
module set_clear_latch(n_set, n_clear, Q, n_Q);
input n_set, n_clear;
output Q, n_Q;

nand nand_1(Q, n_set, n_Q);      //instantiate nand_1
nand nand_2(n_Q, Q, n_clear);    //instantiate nand_2

endmodule  // note endmodule is one word and
           // line does not end in a semicolon
```

    (Note: The module must have the same name that was listed in the " the top level entity" box in the new project wizard i.e. it must be named "set_clear_latch"). Save the file as "set_clear_latch.sv". The file name is always the same as the top level module name (The name and appropriate folder should come up automatically and the .sv extension will be added automatically.).[2]

    It is pointed out that the editor does not turn on the highlighting of the Verilog commands until the file is saved.

12. Compile the project by selecting "Processing" from the top menu. Click on "start compilation". This may take a few minutes. The percent progress is given in two places. In the "task" window and on the bottom right of the screen.

    If there are any errors (red print), alter your Verilog HDL description appropriately.

[2]Note that the .sv extension informs Quartus to use the SystemVerilog compiler when analyzing this file. Files with the .v extension will be processed using a compiler specific to the older versions of Verilog. Given that CME 341 will generally be using the older features of the language, a .v extension may also work, but it is recommended to use .sv for consistency.

13. While the compiler is running it will print information (green print), warnings (blue print) and errors (red print). Most of the of the time a warning means something is wrong. In this case there will be a few warnings that will be explained at a later time. One of the warnings indicates that the circuit that was built has feedback and appears be a latch. There is always the potential for circuits with feedback to oscillate so when this is detected by the compiler it prints a warning.

14. Upon successful completion of a compilation, Quartus creates a couple of new folders. the first folder/directory it creates is called "simulation" and it is located inside the project folder. It creates a second folder/directory called "modelsim" inside the simulation folder. Several files are placed in this folder including an output file named "set_clear_latch.vo". The .vo file contains worst case propagation delay information so the simulator can simulate the circuit that was constructed by the compiler. Other .vo files are also created, the number of which depend on the FPGA family selected. The file that will be imported into the simulator is "set_clear_latch.vo".

1.8 Simulation: Test Bench and Software Tool

1.8.1 test bench

The Quartus compiler produces a Verilog HDL file that has the same name as the top-entity input file, but with the extension .vo. The .vo file differs from the .sv file in that it describes the circuit implemented by the compiler and routed on the FPGA by the router in complete detail, which means it includes all the propagation delays. The .vo file can be used in a Verilog simulator to exactly model the circuit that was built by the compiler.

This .vo file can not be compiled in Quartus because it will contain Verilog HDL instructions that can not be synthesized, e.g. the propagation delay instructions.

A Verilog HDL simulator is of great help in debugging circuits (i.e. finding and correcting typos and also flaws in the design). The circuit being debugged is sometimes called the "device under test" or "DUT" .

A simulator has three functions:

1. It generates the input signals (i.e. the stimuli) for the circuit being debugged. That is, it acts like a signal generator.
2. It models the DUT i.e. the circuit that is being debugged. That is, it simulates the circuit that was built by the Quartus compiler.

   The circuit can be modelled with or without all the propagation delays included.

   (a) The simplest and the one that runs the fastest is one that does not include the propagation delays. This is called a functional model. The DUT is modelled without consideration for propagation delay.

   A functional model does not need to know anything about the target FPGA. The functional model does not include propagation delays and so can not be used to check timing. The functional model is described by the .sv file used in synthesis. Therefore, simulator uses the .sv file for functional simulation. (i.e. The same file that is used

by Quartus.)

(b) The model that includes propagation delays is called a timing model. It models the circuit that is to be debugged exactly as it is implemented in the target technology, complete with all propagation delays. This model is used to check if the circuit meets timing.

The timing model will be used for all the assignments in this class. The circuits built in this class are so small the computation time is not significant so there is no need to run a functional simulation.

The .vo file produced by a Quartus compilation is a timing model of the circuit. In fact, Quartus produces several .vo files. One takes on the name of the top entity. This models a circuit using typical propagation delays. The others take on the name of the top entity with a `_fast` or `_slow` appended to it. They model the extremes in the chip to chip variations. The typical model will be used as the input to the simulator for all assignments in this class.

3. It displays the simulated output waveforms (also displays the input waveforms (stimuli)).

The simulator is an executable program with a GUI interface. The simulator is controlled through the GUI.

The top entity (i.e. top module), which is the Verilog HDL input to the compiler (for the simulator), is referred to as a test bench so the name of the top module often includes the word "testbench" or "test_bench".

While a simulator and Quartus both compile Verilog HDLs, the compiler in the simulator is different than the compiler in Quartus. The compiler in Quartus produces two things:

1. A file that can be downloaded to the FPGA to configure it (i.e. programs it) to implement the circuit described by the Verilog HDL.

2. Another Verilog HDL with a .vo extension that describes the circuit that was built in complete detail, which includes all the propagation delays.

The operation of the modelsim-altera simulator will be demonstrated with a

test bench for the set_clear_latch circuit. The simulator will use set_clear_latch.vo, which was generated by Quartus when the project set_clear_latch was compiled.

The Verilog HDL for the test bench is given below.

```
‘timescale 1 ns / 1 ps // compiler directive that must be included
                       // the first time listed is used for the unit
                       // of time on waveform plots
                       // the second time listed indicates precision.
                       // all delays are rounded to the ‘‘precision time’
       // allowable units of time are ms, us, ns, and ps
       // allowable numbers are 1, 10, 100
module set_clear_latch_testbench(); // no inputs or outputs
reg n_set, n_clear;
wire Q, n_Q;

initial
#150 $stop;  // stop the simulation at 150 ns

initial
begin
#10 n_set = 1’b0;   // schedule n_set to be 0 at t=10 ns
#10 n_set = 1’b1;   // schedule n_set to be 1 at t=10+10=20 ns
#55 n_set = 1’b1;   // schedule n_set to be 1 at t=20+55=75 ns
end
initial
#65 n_set = 1’b0;   // schedule n_set to be 0 at t=65 ns


initial
begin
n_clear = 1’b1;       // schedule n_clear to be 1 at t=0
#50 n_clear = 1’b0;   // schedule n_clear to be 0 at t=50 ns
#5 n_clear = 1’b1;   // schedule n_clear to be 1 at t=50+5=55 ns
```

```
#10 n_clear = 1'b0;   // schedule n_clear to be 0 at t=55+10=65 ns
#15 n_clear = 1'b1;   // schedule n_clear to be 0 at t=65+15=80 ns
end

// instantiate the prototype for the set clear latch
set_clear_latch latch_1(.n_set(n_set),
                        .n_clear(n_clear),
                        .Q(Q)
);

endmodule
```

**It is pointed out the text that follows a double slash, i.e. //, is a comment.

The testbench module declaration must be preceded by a compiler directive that indicates the unit size for time and the round off precision that should be used in the calculations. The line preceding the module declaration above contains the compiler directive `‘timescale 1 ns / 1 ps`. Two times are listed. The first is 1 ns. The second is 1 ps. The first indicates the units for time and delay in the testbench module. The second indicates the precision and all times and delays are rounded to this precision. For the testbench above, all numbers that indicate time are multiplied by 1 ns and after the multiplication the result is rounded to 1 ps. There are restrictions on the numbers and units of time that can be used in the `‘timescale` directive. The number must be either 1, 10, or 100. The units of time can be ms, us, ns and ps.

Next notice that the port list in the testbench module declaration does not list any inputs or outputs.

The first two lines inside the module declare signals to be a certain type. The signals “n_clear” and “n_set” must be declared type “reg” because they are created with a “initial” procedure. The signals “q” and “n_q” must be declared type “wire” because they are created inside an instantiation. Signal types will be explained more fully later.

The next line consists of the key word "initial", which signifies a procedure. The instruction that follows an initial is executed once. In this case the `#150 $stop` instruction is executed once. The instruction has two parts. The first part, which is `#150`, creates a delay of 150 ns. The second part is `$stop`, which stops the simulation. The entire instruction reads wait 150 ns and then stop the simulation.

The second initial procedure is followed by several instruction inside a begin-end wrapper. All instructions inside the begin-end wrapper are executed once. This sequences of instructions changes the signal "n_set" at different times. The instructions make "n_set" low at time $t = 10$ ns, high 10 ns later at $t = 20$ ns, high again 55 ns later at time $t = 75$ ns.

The third initial procedure also modifies "n_set" as well. It makes "n_set" low at $t = 65$ ns. notice "n_set" is not defined before a time of 10 ns.

The fourth initial procedure defines signal "n_clear". It makes "n_clear" high at $t = 0$, low at $t = 50$ ns, high at $t = 55$ ns, low at $t = 65$ ns and finally high at $t = 80$ ns.

After the third initial procedure the Verilog HDL model for the set clear latch is instantiated. The signals "n_clear" and "n_set" generated in the test bench are connected to "n_clear" and "n_set" inputs of the set_clear_latch. The outputs of the set_clear_latch are connected to the wires q and n_q in the test bench.

Simulation with Modelsim

1. Open Modelsim by selecting **start → all programs → electrical engineering → Modelsim Intel FPGA Starter Edition**.
2. Modelsim opens and brings up a pop-up window with license information. Inside this window there is a rectangular button labelled "jumpstart". Click on the "jumpstart" button to enter into Modelsim.
3. Upon entry a window will pop-up asking if you wish to create a new project or open an existing project. Select the new project option. This will bring up a "new project" window.

Note: A new project can also be created by selecting **file** → **new** → **project**.

4. The "new project" window has four fields.
   (a) In the "project name" field enter set_clear_latch_testbench.
   (b) In the "project location" field browse to find the directory where Quartus placed the .vo file "set_clear_latch.vo". That directory will be a sub directory of the Quartus project directory. It should be ··· `set_clear_latch/simulation/modelsim`.
   (c) Do not change the "default library name" field. The default library name must be "work".
   (d) Do not change the "copy settings from" field.

   After filling in the "project name" and "project location" fields, click O.K.

5. A window will appear with four options. Choose "add an existing file". Then, in the window that pops up, browse to select the file set_clear_latch.vo. Then click O.K. Then close the window.

6. Create the test bench module using the text editor in Modelsim. Start the editor by selecting
   file → **new** → **source** → **systemverilog**.
   Type in the set_clear_latch_testbench module given above. Save the file as `set_clear_latch_testbench.sv`. In some cases, the .sv extension may not be added automatically and must be typed in.

7. Add file `set_clear_latch_testbench.sv` to the project by selecting **project** → **add to project** → **existing file** and completing the box.

8. Compile both files by selecting **compile** → **compile all**.

9. Load the simulation by selecting
   simulate → **start simulation**.
   This will bring up a window with several tabs. If the "design" tab is not selected then select it. With the design tab selected the window will show a list of libraries.

(a) Expand the "work" library, which should be the one at the top of the list. Expanding the work library should list the two source files associated with the project (perhaps one or more others created by modelsim-altera) beneath the work library.

(b) Click on the top entity, which is `set_clear_latch_testbench`. (The top entity is the highest level module, i.e. the module that no other module instantiates.) The selection should get written into the dialog box near the bottom of the window that is titled "design unit(s)".

(c) **Important:** The first time you load your design into the simulator you must select a library that contains support data for the target FPGA. Select the "libraries" tab, near the center of the top of the window.

i. This should change the contents of the window to show two panes: The top pane is titled "search libraries" the bottom pane is called "search libraries first". Click the "add" button associated with the "search libraries" pane. This will bring up a yet another window.

ii. In that window click on the down arrow (**do not click on browse**). This will allow you to select one of the libraries that was listed in the original window.

You will need to select two or three libraries in the order given below. (You will need to click "add" again to add another library).

Scan down the list and select `cycloneive_ver` and then scan down the list again and select `altera_mf_ver` and `altera_ver` (You probably don't need the last one for this design).

After the libraries are selected click O.K. This will list the libraries in the pane titled "search libraries".

iii. Click O.K at the bottom of the window to load the design.

(d) Once the design is loaded a main window will be partitioned into three windows. The window on the top left is called the "objects" window. It will contain a list of the signals. Select the signals (hold

down cntl and click on them one at a time) then right click and select: **add to** → **wave** → **selected signals**. This will open a new tab called "wave" in the window on the right and it will transfer the selected signals to that tab.

(e) To run the simulation select
simulate → **run** → **run -all**.
This will run the simulation until it reaches a stop command. Once the simulation is completed the "set_clear_latch_testbench.sv" file will be displayed in the left window with a big blue arrow pointing to the command that stopped the simulation.

(f) Select the "wave" tab on the window. Right click in this window (on the black part) and select "zoom full". This will display the waveforms for the length of the entire simulation, which is from 0 to 150 ns.

Navigating in the wave window

On the menu bar at the top of the Modelsim wave window there is a symbol that looks like a set of traffic lights. To the right of that set of traffic lights is a set of 8 symbols used to control the cursors. Clicking on the yellow symbol annotated with a + will add a cursors. Clicking on the yellow symbol annotated with a − will remove the active cursor. The other 6 symbols are used to move the active cursor. These symbols only act when one or more of the waveforms are highlighted (i.e. selected). The cursor can be moved to coincide with one the edges of the selected signals. The symbols in order from left to right move the active cursor to: nearest edge to the left, nearest edge to the right, nearest falling edge to the left, nearest falling edge to the right, nearest rising edge to the left and nearest rising edge to the right.

The units on the time axis in the wave window can be changed by right clicking just below the axis and selecting **grid and timeline properties ...**. This brings up a window that has a box labeled "time units". Select the units you want and click O.K.

Debugging a circuit involves cycling through the process of modifying the Verilog HDL source code files, compiling them in Quartus, loading the simulator

and then running the simulator. Every time the simulator is loaded (i.e. the start simulation command is executed) the configuration of the wave window is lost. This means the signals of interest must be added to the wave window again and any changes made to the radix must be respecified, etcetera. The wave window can be reconfigured very quickly by saving the configuration of the wave window in a .do file and then running that file right after the "start simulation" command is executed.

Save the configuration of the wave window by first activating the wave window (i.e. click on the wave tab) and then selecting **file** → **save format**. This will bring up a window that asks for the directory path and name of the file in which you wish to save the configuration information. The default path is the Modelsim project directory and the default name is `wave.do`. There is no need to change these default settings. Just click OK.

To reload the wave window configuration file after the simulation has been restarted type `do wave.do` in the transcript window. The transcript window is the window at the bottom with the prompt `vsim (some number)>`.

Saving and Retrieving Wave Window Data

The data in a wave window is saved by selecting **File** ⟶ **data_set** ⟶ **save as**. It must be saved under a name with extension .wlf. Usually, for reasons that will be explained, the formating file, i.e. a wave.do file, is also saved at the same time. It is usually given the same name as the .wlf file, but with a .do extension.

Some versions of Modelsim allows more than one wave window to be open at once, but the version we have only allows one wave window to be open. Therefore, to open a .wlf file, the wave window, if it is open, must be closed prior to opening a saved .wlf file. The .wlf file can be opened either using a menu selection or by a single line command. To open the file from the menu select **File** ⟶ **data_set** ⟶ **open**. To open it using the command line type:

```
vsim -view gold=saved_wave.wlf
```

where `save_wave` is the name that was assigned to the saved data set file. The name "`gold`" is assigned to the wave window. It can be any name, but is unimportant in this class as only one wave window can be open.

Once the .wlf file is opened, signals must be selected from object window and moved to the wave window before they will be visible in the wave window. Unlike after running a simulation, the wave forms appear as soon as the object is moved. Once the data is displayed it can be manipulated with the display tools, which of course includes the zoom.

Running a simulation after a .wlf file is opened will over write the wave window.

If the format of the wave window was saved at the time the data set was saved, then the waveforms can be restored in the wave window by issuing the command:

```
do saved_wave.do
```

where `save_wave` is the name that was assigned to the saved format file. Again, once the data is displayed it can be zoomed etc. with the display tools.

1.9 Configuring the DE2 board

Quartus can not generate a configuration (programming) file until the signals in the top level port list have been assigned to pins on the FPGA. Of course, the wires on the DE2-115 printed circuit board connects the FPGA pins to chips, leds, switches and such that are mounted on the DE2-115. For purposes of assigning pins to the
set_clear_latch project the latch output should be connected to two leds and the latch inputs should be connected to two keys (momentary push button switches that are active low).

The first step in assigning pins is to create a comma separated file (a text file) by selecting **File** → **New** → **Other Files** → **text file**. Enter one of the following text stings into the file and then save the file in the project directory

as `set_clear_latch.csv`. Make sure to check mark the box that asks if the file is to be added to the project. For the Cyclone IV E on the DE2-115 board enter:

```
To,        Assignment Name, Value
n_set,     Location,  PIN_M23
n_clear,   Location,  PIN_M21
Q,         Location,  PIN_E21
n_Q,       Location,  PIN_E24
```

Once the .csv file is saved, the information in the file has to be conveyed to Quartus. This is done in a window brought up by selecting **Assignments** → **Import Assignments**. Select `set_clear_latch.csv`. If `set_clear_latch.csv` does not appear in the list change the filter so .csv files are displayed. There is a check box option that allows for the previously assigned pins to be copied into the project. This box can be checked.

In this project only 4 FPGA pins are used. However, there are many other FPGA pins that are physically connected with tracks on the PCB to other chips/devices mounted on the DE2-115 board. Surely, some of the FPGA pins will be connected to outputs from the other chips/devices. There will be a "burn out" problem if those FPGA pins are programmed to be output pins.

To make sure no FPGA pins fight with other devices, all the unused FPGA pins, i.e. the pins not assigned in the current project, should be programmed as "inputs tri-stated with weak pull-up resistors". This is the default setting for the FPGA so nothing needs to be done.

However, should the need arise to set the unused pins manually it is done by selecting **Assignments** → **Device** to bring up the "Device" window. Click on Device and Pin Options to pop up the "device and pin options" window. Select **Unused Pins** in the category list on the left and then pull down the list associated with the "Reserve all unused pins:" box and select **As input tri-stated with weak pull-up**. Then click O.K. to take down the "device and pin options" window. Click O.K. again to take down the "Device" window.

The FPGA is programmed by transferring the configuration file `set_clear_latch.sof` to the FPGA. This is done as follows:

1. Connect the power to the DE2-115 board.
2. Connect the USB cable to DE2-115 board through the connector closes to the power supply.
3. Connect other end of the USB cable to the PC.
4. Press the red power on switch. If the board is properly powered lights will start flashing.
5. Select **Tools** $\rightarrow$ **programmer**. This will bring up a window. If the "Start" button is made active the FPGA is programmed by clicking "start". Otherwise Click on the "Hardware Setup" button. This will bring up another window. Pull down the options for the "currently selected hardware" box and select **USB-blaster**. Close the window. The start button in the programmer window should now be activated. Click "Start" to program the FPGA.

1.10 Exercise on Using Quartus and Modelsim

1. Design a transparent latch in Verilog HDL and compile it in Quartus.
2. Write a test bench in Verilog HDL for the transparent latch.
3. Verify the operation of the transparent latch via simulation with Modelsim.

1.11 D Flip/Flop

1.11.1 Schematic

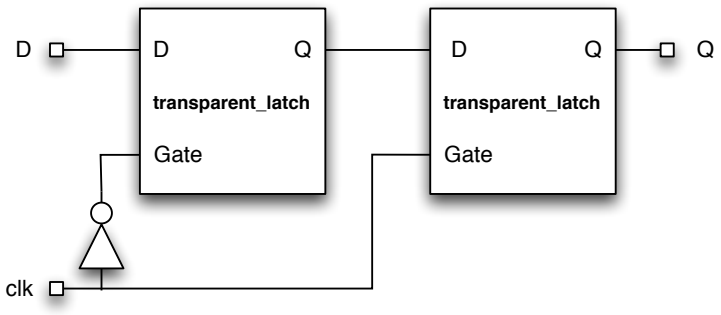


Figure 1.21: D Flip/Flop

There are many ways to make a D Flip/Flop. This one uses two transparent latches in a master/slave configuration.

1.11.2 Exercise

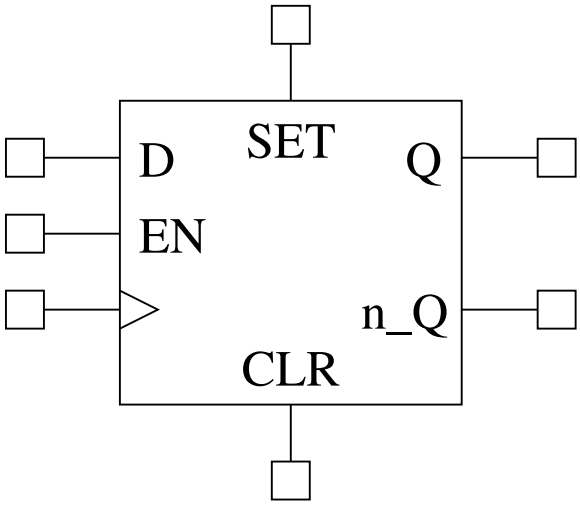
Below is a partially complete Verilog HDL of the D flip-flop shown in Figure 1.21. It is a structural description of a D flip flop. There are other structures that also describe a D flip-flop, but they will not be discussed here.

Please complete the Verilog HDL below. The structural description in Figure 1.21 is not fully annotated. Two wires need to have names assigned for the purposes of completing the Verilog HDL given below.

```
module D_FF ( ...
input ...
output ...
wire ...
not inverter_1 ( ...
transparent_latch latch_1 ( ...
transparent_latch latch_2 ( ...
endmodule
```

1.11.3 D Flip-Flop with enable

A D flip-flop can be constructed with an extra input that, when high, enables the clock input. It is referred to as the clock enable or simply the enable. While the clock enable input is high the D flip-flop works as usual. While the clock enable is low, the clock input is ignored and the output remains unchanged.



Converting an ordinary D-flip-flop to one with a clock enable is the subject of a question in assignment 1. For that reason the circuity needed for the conversion is not given in these notes. However, it will be discussed in class after assignment 1 is due.

1.12 Vectors

Vectors are simply a group of wires or registers. The Statement:

```
wire [7:0] x;
```

makes 8 wires called x[7] ... x[0].

Suppose y1 and y2 are defined by:

```
wire [7:0] y1;
wire [0:7] y2;
```

Then the statements:

```
assign y1 = x;
assign y2 = x;
```

connects the 8 wires in `x` to the 8 wires in `y1` as follows:
y1[0] to x[0], y1[1] to x[1], ..., y1[7] to x[7],
and it connects the 8 wires in `x` to the 8 wires in `y2` as follows:
y2[7] to x[0], y2[6] to x[1], ..., y2[0] to x[7],