LABORATORY MANUAL

DEPARTMENT OF
ELECTRICAL & COMPUTER ENGINEERING

UNIVERSITY OF CENTRAL FLORIDA

EEE 3342
Digital Systems

Revised
August 2012

CONTENTS

Safety Rules and Operating Procedures

1. Note the location of the Emergency Disconnect (red button near the door) to shut off power in an emergency. Note the location of the nearest telephone (map on bulletin board).

2. Students are allowed in the laboratory only when the instructor is present.

3. Open drinks and food are not allowed near the lab benches.

4. Report any broken equipment or defective parts to the lab instructor. Do not open, remove the cover, or attempt to repair any equipment.

5. When the lab exercise is over, all instruments, except computers, must be turned off. Return substitution boxes to the designated location. Your lab grade will be affected if your laboratory station is not tidy when you leave.

6. University property must not be taken from the laboratory.

7. Do not move instruments from one lab station to another lab station.

8. Do not tamper with or remove security straps, locks, or other security devices. Do not disable or attempt to defeat the security camera.

9. When touching the FPGA development boards please do not touch the solid-state parts on the board but handle the board from its edge.

10. **ANYONE VIOLATING ANY RULES OR REGULATIONS MAY BE DENIED ACCESS TO THESE FACILITIES**.

I have read and understand these rules and procedures. I agree to abide by these rules and procedures at all times while using these facilities. I understand that failure to follow these rules and procedures will result in my immediate dismissal from the laboratory and additional disciplinary action may be taken.

_____ _____

Signature Date Lab #

Laboratory Safety Information

Introduction

The danger of injury or death from electrical shock, fire, or explosion is present while conducting experiments in this laboratory. To work safely, it is important that you understand the prudent practices necessary to minimize the risks and what to do if there is an accident.

Electrical Shock

Avoid contact with conductors in energized electrical circuits. The typical can not let-go (the current in which a person can not let go) current is about 6-30 ma (OSHA). Muscle contractions can prevent the person from moving away the energized circuit. Possible death can occur as low 50 ma. For a person that is wet the body resistance can be as low as 1000 ohms. A voltage of 50 volts can result in death.

Do not touch someone who is being shocked while still in contact with the electrical conductor or you may also be electrocuted. Instead, press the Emergency Disconnect (red button located near the door to the laboratory). This shuts off all power, except the lights.

Make sure your hands are dry. The resistance of dry, unbroken skin is relatively high and thus reduces the risk of shock. Skin that is broken, wet, or damp with sweat has a low resistance.

When working with an energized circuit, work with only your right hand, keeping your left hand away from all conductive material. This reduces the likelihood of an accident that results in current passing through your heart.

Be cautious of rings, watches, and necklaces. Skin beneath a ring or watch is damp, lowering the skin resistance. Shoes covering the feet are much safer than sandals.

If the victim isn't breathing, find someone certified in CPR. Be quick! Some of the staff in the Department Office are certified in CPR. If the victim is unconscious or needs an ambulance, contact the Department Office for help or call 911. If able, the victim should go to the Student Health Services for examination and treatment.

Fire

Transistors and other components can become extremely hot and cause severe burns if touched. If resistors or other components on your proto-board catch fire, turn off the power supply and notify the instructor. If electronic instruments catch fire, press the Emergency Disconnect (red button). These small electrical fires extinguish quickly after the power is shut off. Avoid using fire extinguishers on electronic instruments.

First Aid

A first aid kit is located on the wall near the door. Proceed to Student Health Services, if needed.

Introduction

When in Doubt, Read This

Laboratory experiments supplement class lectures by providing exercises in analysis, design and realization. The objective of the laboratory is to present concepts and techniques in designing, realizing, debugging, and documenting digital circuits and systems. The laboratory begins with a review of Xilinx's ISE FPGA development environment, which will be used extensively during the laboratory. Experiment #1 introduces the student to the fundamentals of the ISE and its tool set such as the synthesizer, the test-bench user input program for the simulator, the ISE simulator, and the FPGA implementation. Xilinx's FPGA development tools support for schematic capture as well as HDL input such as VERILOG or VHDL. In Experiment #2, the basic operations found in the ISE will be used to design and simulate a simple Boolean expression that will be using the student experimenter kit using 74LSXXXX parts. Experiments #3 through #7 are experiments that deal will the design and hardware implementation of combinational logic circuits. These circuits will by designed using the ISE and implemented solely using an FPGA. Experiments #8 through #11 deal with the design and hardware implementation of sequential logic circuits and will also be designed and implemented using the FPGA and ISE development tools.

Laboratory Requirements:
This laboratory requires that each student obtains a copy of this manual, a bound quad-ruled engineering notebook and have access to Xilinx's ISE version 9.2. The student can use the ISE program on the laboratory computers or the student can go to www.xilinx.com to download the ISE development tools.

The student is to prepare for each laboratory by reading the assigned topics. The laboratory notebook should contain the necessary tables, graphs, logic and pin assignment diagrams and identify the expected results from the laboratory exercise as directed by the pre-laboratory preparation assignments or by the laboratory instructor. Depending on the laboratory assignment, the pre-laboratory preparation may be due at the beginning of the laboratory period or may be completed during the assigned laboratory period. Be informed that during each laboratory period the instructor will grade your notebook preparation.

During the laboratory period you are expected to construct and complete each laboratory assignment, record data, and deviations from your expected results, equipment used, laboratory partners, and design changes in your laboratory notebook. A laboratory performance grade will be assigned by the laboratory instructor upon successful completion of the above-described tasks for each experiment.

Each student will be assigned to a computer with an FPGA board connected to it. Each student is responsible for his or her own work including design and documentation. A Laboratory Report, following the guidelines presented in this handout is due the laboratory period following the completion of the in-laboratory work or when the instructor designates. A numeric grade will be assigned using the attached laboratory-grading sheet.

Laboratory reports will be due before the start of each laboratory. A penalty of five points will be charged for those late by up to one week (0-7 days). No credit will be given for laboratory reports over-due by more than one week. However, a student must complete each assigned experiment in order to complete the laboratory. By not turning in a laboratory report, a student will receive an incomplete for that report, which results in an incomplete for the laboratory grade.

Students who miss the laboratory lecture should make arrangements to make up the laboratory at a later time. Points may be taken off the laboratory experiment and the student might not be allowed to attend the remainder of the laboratory because this will burden the laboratory instructor and the rest of the laboratory that day.

Students who are late to their laboratory section will not receive the five pre-laboratory points. A penalty of five points will also be charged for turning in a late laboratory report. If, for some reason, a student cannot attend the regularly scheduled laboratory time period, then he / she must make arrangements to make up the laboratory experiment at a later time and hand in the laboratory report and pre-laboratory early to avoid a ten point penalty.

Laboratory Point Breakdown
In-Laboratory Grade (10 points):

1. Pre-Laboratory Assignment	5 points
2. Design Completion	5 points

Laboratory Report Grade (15 points):

3. Problem or Objective Statement, Block Diagram, and Apparatus List	1 point
4. Procedure and Data or Design Steps	3 points
5. Results Statement and Logic Schematic Diagram	4 points
6. Design Specification Plan	2 points
7. Test Plan	2 points
8. Conclusion Statement	3 points
TOTAL	**25 points**

The final laboratory grade can be a percentage, an incomplete or a failing grade. If the student receives an incomplete or failing grade for the laboratory, an incomplete may be assigned for the whole course.

Guidelines for Laboratory Reports:
The laboratory report is the record of all work pertaining to the experiment, which includes any pre-laboratory assignments, schematic diagrams, and Xilinx's ISE printouts when applicable. This record should be sufficiently complete so that you or anyone else of similar technical background can duplicate the experiment by simply following your laboratory report. **Original work is required by all students (NO PHOTOCOPIES OR DUPLICATE PRINTOUTS).** Your laboratory report is an individual effort and should be unique. The laboratory notebook must be used for recording data. Do not trust your memory to fill in the details at a later time. An engineer will spend 75 percent of his/her time for documentation.

Organization in your report is important. It should be presented in chronological order with descriptive headings to separate and identify the various parts of the experiment. A neat, organized and complete record of the experiment is just as important as the experimental work. **DO NOT SECTION OFF DIAGRAMS, PROCEDURES, AND TABLES.**

The following are general guidelines for your use. Use the standard paper prescribed by your instructor. A cover page is required for each laboratory including your name, PID, name and number of the experiment, date of the experiment and the date the report is submitted. Complete the required information and attach to the front of each report. If a cover page is not included with a report, then points may be taken off.

The report should contain the following (not segmented or necessarily in this order):

- **Heading:** The experiment number, your name, and date should be at the top right hand side of each page.
- **Objective:** A brief but complete statement of what you intend to design or verify in the experiment should be at the beginning of each experiment.
- **Block Diagram:** A block diagram of the circuit under test and the test equipment should be drawn and labeled so that the actual experiment circuitry could be easily duplicated at any time in the future.
- **Apparatus List:** List the items of equipment, including IC devices, with identification numbers using the UCF label tag, make, and model of the equipment. It may be necessary later to locate specific items of equipment for rechecks if discrepancies develop in the results. Also include the computer used and the version number of any software used.
- **Procedure and/or Design Methodology:** In general, lengthy explanations are unnecessary. Be brief. Keep in mind the fact that the experiment must be reproducible from the information given in your report. Include the steps taken in the design of the circuit: Truth Table, assumptions, conventions, definitions, Karnaugh Map(s), algebraic simplification steps, etc.

- **Design Specification Plan:** A detailed discussion on how your design approach meets the requirements of the laboratory experiment should be presented. Given a set of requirements there are many ways to design a system that meets these requirements. The Design Specification Plan describes the methodology chosen and the reason for the selection (why). The Design Specification Plan is also used to verify that all the requirements of the project have been implemented as described by the requirements.

- **Detailed Schematic Diagram:** A detailed schematic diagram should be presented. Standard symbols should be used. For logic diagrams, inputs should enter at the left side or top of the diagram and the outputs at the bottom or right side of the diagram. Data flows left to right and top to bottom. If switches and LEDs are used for logic inputs and to test logic outputs respectively, the switch numbers and LED numbers should be identified. The switches and LEDs should be organized to simplify the testing of the circuit. A location diagram should be included at the bottom of each schematic diagram. For Experiment #2 74LSXXXX parts will be used in addition to the FPGA BASYS board. The BASYS board layout with the appropriate FPGA pins layout must be included in the laboratory report. See the Sample Schematic Diagram in Appendix C for a good example of a detailed diagram and Appendix D for the pin layout of the switches, LED's and Clock signals for the BASYS development board.

- **Test Plan:** A test plan describes how to test the implemented design against the given requirement specifications. This plan gives detailed steps during the test process defining which inputs should be tested and verifying for these inputs that the correct outputs appear. The laboratory instructor will use this test plan to test your laboratory experiment.

- **Results:** The results should be presented in a form, which makes the interpretation easy. Large amounts of numerical results are generally presented in a graphical form. Tables are generally used for a small amount of results. Theoretical and experimental results should be on the same graph or arranged in the same table for easy correlation of these results. For digital data, prepare a simulation and response table and record logic levels as "1"s and "0"s. The above table is similar to a Karnaugh Map or State Transition Table. Identification of the size of a logic circuit, in terms of number of inputs, gates and packages is often required in a design-oriented experiment.

- **Conclusion:** This is your interpretation of the objectives, procedures and results of the experiment, which will be used as a statement of what you learned in performing the experiment. This is not a summary. Be brief and specific but complete. Identify the advantages and/or disadvantages of your solution in design-oriented experiments. The conclusion also includes the answers to the questions presented in each experiment.

EXPERIMENT #1

XILINX FPGA TOOLS

Goals: To introduce the modeling, simulation and implementation of digital circuits using Xilinx's FPGA ISE design tools.

References: Within the ISE there are several tutorials that are available. In particular, the ISE Quick Start describes how to create a file using either VHDL or VERILOG. The ISE also offers an extensive set of manuals under the Help menu. In addition to the electronic version of the manual and the Quick Sort Tutorial, Xilinx offers many tutorials that are available on the web site at www.xilinx.com.

Equipment: The Xilinx ISE field programmable gate arrays FPGA development tools are available in laboratory. These tools can also be downloaded from Xilinx's web site at www.xilinx.com. The ISE WEBBASE development tools that we will use for the laboratory experiments are located under the support download section at www.xilinx.com/support/download. The ISE WEBPACK software is available for windows XP, Solaris and Linux. Please take note that the file download size exceeds 1 GB and also during the installation process updates may have to be installed. It can take you up to a few hours to download and install the ISE on a user computer. The user does not need to have the BASYS development board interface to the computer to design and simulate an FPGA. Finally, a copy of EXPORT from Digilent Inc. (www.digilentinc.com) will be used to download an FPGA design to the FPGA configuration ROM located on the BASYS development board. This ROM is read at power up by the FPGA to configure the device.

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment.

Discussion: Xilinx's ISE is an FPGA design, simulation and implementation tool set that allows the designer the ability to develop digital systems using either schematic capture or HDL using VERILOG or VHDL. These digital systems are then verified using simulation tools that are part of the development system. Once the simulation outputs meet the design requirements, implementation is simply assigning the inputs and outputs to the appropriate pins on the FPGA. Appendix D gives the pin configuration for the BASYS board by Digilent Inc. (www.digilentinc.com) relating the LED and switch connections to the FPGA pin assignments.

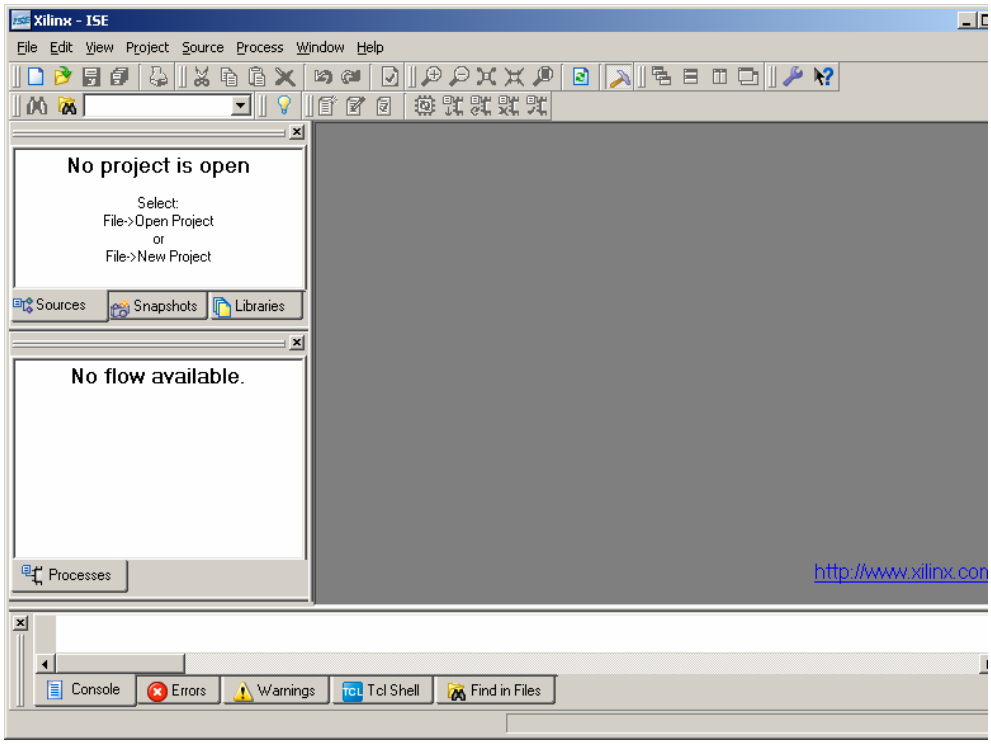
Experiment #1 is divided into four sections. Part 1 of this experiment will guide the student through the steps required to create an FPGA using the schematic

capture part of the ISE and the steps required to synthesize this design. Next, the steps required to simulate this design are given along with the steps required to implement the design in the FPGA on the BASYS board. Part 2 will expand on part 1 for additional logic gates (NAND, OR, EXOR, NOT). In part 3 of this experiment, the design implemented in Parts 1 and 2 will be implemented using the VERILOG design language. The steps required to create a VERILOG project will be given along with the steps required to simulate and implement this design. In part 4, a two-input five-output logic system will be designed and implemented.

Part 1. Introduction to the XILINX ISE

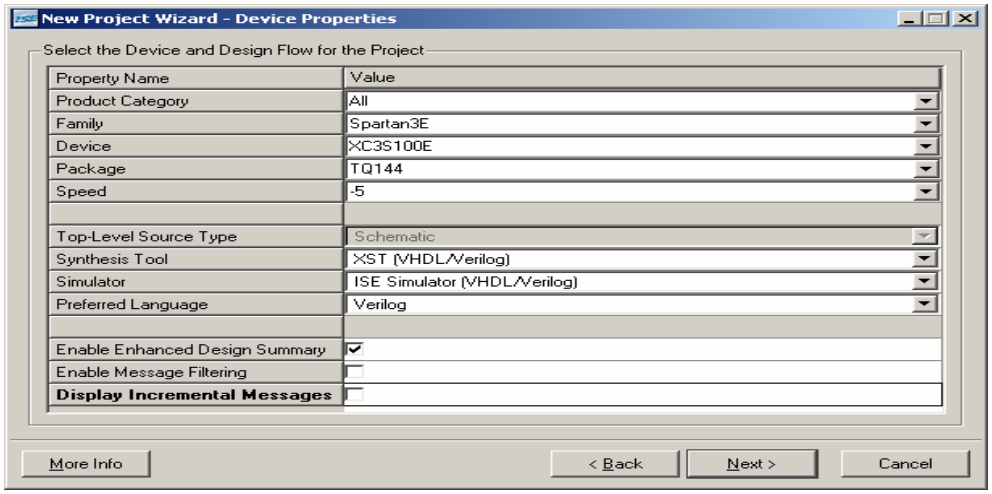
In this part, you will use Xilinx's ISE to design, simulate and implement a simple 2 input AND gate digital circuit. Once completed, this 2 input AND gate implementation will be downloaded to the BASYS board and then tested using the on board LED's and switches.

1. Double click on the ISE icon to open up the development tools as shown below. There are five general areas that will be of interest. At the top are the tool bars for file input and output, for running of the various tools, and symbol inputs for the schematic capture. To the left are **Sources window** and the **Processes window** and at the bottom is the **display window**. The user work area is located to the right.

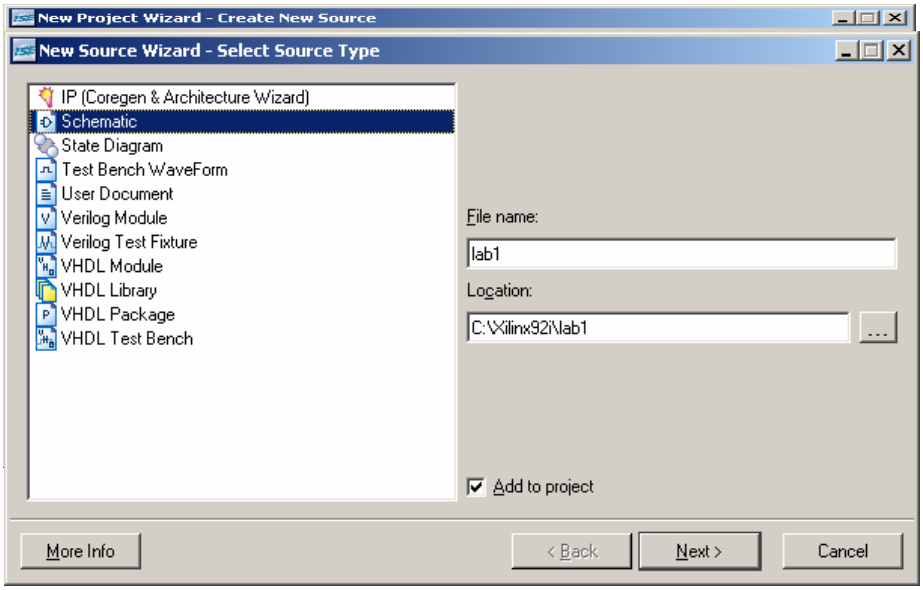


2. Now that the ISE is open, the next step is to open a project. You have to be careful about where to save your project in the computer lab. The computers

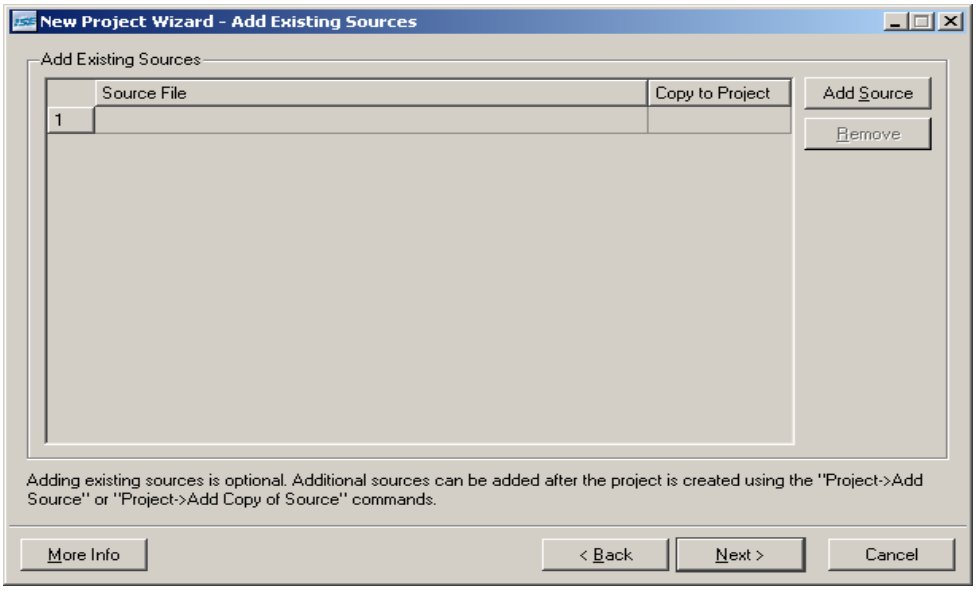
in the lab run a hard disk protection program that could interfere with Xilinx. So if you save your project in any folder, Xilinx might have problem with running the simulation. You have two choices: (1) either save the project directly on your USB flash disk. This option is good since your USB disk have normal read/write access so Xilinx will run correctly. However, this option can be slow for USB flash disks. The option (2) is to save the project in a folder that's in the desktop. Start by creating a folder on the desktop called 'temp'. Create this folder in Windows, not from Xilinx. Then, in Xilinx, create a new project inside temp. If your project is called 'Lab1', it will be in the folder \Desktop\temp\Lab1. When you finish your lab, you can copy your project on your flash disk. Next select the **Top-Level Source Type** as **schematic**. Later in this experiment the VERILOG language type will be selected. When all of the inputs have been entered the user should click the next button. This will lead to the configuration menu as shown below. The important items on this menu are the FPGA type and the Preferred Language type. Select the **Preferred language type to VERILOG** and the **FPGA family to Spartan3e**, **device to XC3S100E**, and **Speed to either -4 or -5**. For the **package type,** if you are using a Basys board, select **TQ144;** if you are using a Basys 2 board, select **CP132.** So, you might want to go select a board from the cabinet before you proceed so you use the correct settings. The FPGA on the BASYS board is the Spartan3E XC3S100E in a TQ144 (144 pins) package. Select the next button when finished.



3. In the process of creating a new project, the user must also create **a new source file**. There are two ways of doing this. The first is to create a new source file within the next menu or from the Project Menu, New Source item. Selecting the New Source button brings up another dialog box. In this box, select **Schematic** as the top-level type and select **a file name** (eg. Lab1). Finally click the next button.

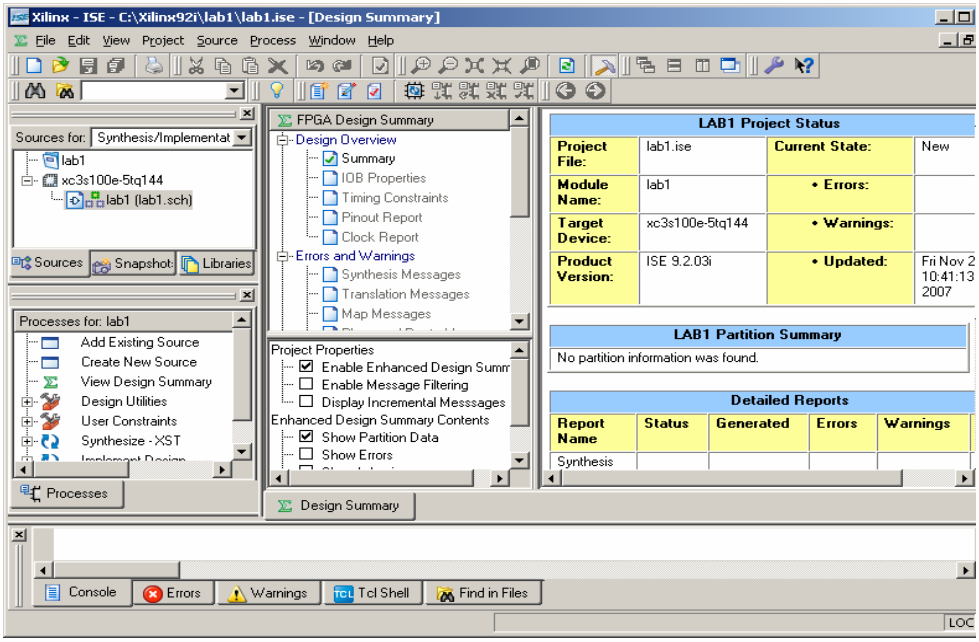


4. Click the finish button. A dialog box will come up and state that the schematic file does not exist and should one be created. Click the Yes button. The last dialog box will indicate that the file *.sch has been created and is of type Schematic. Click the Next button to continue. The Add Source dialog box will now appear. This dialog box allows the user to add an existing source to the project. For Experiment #1, click the Next button.

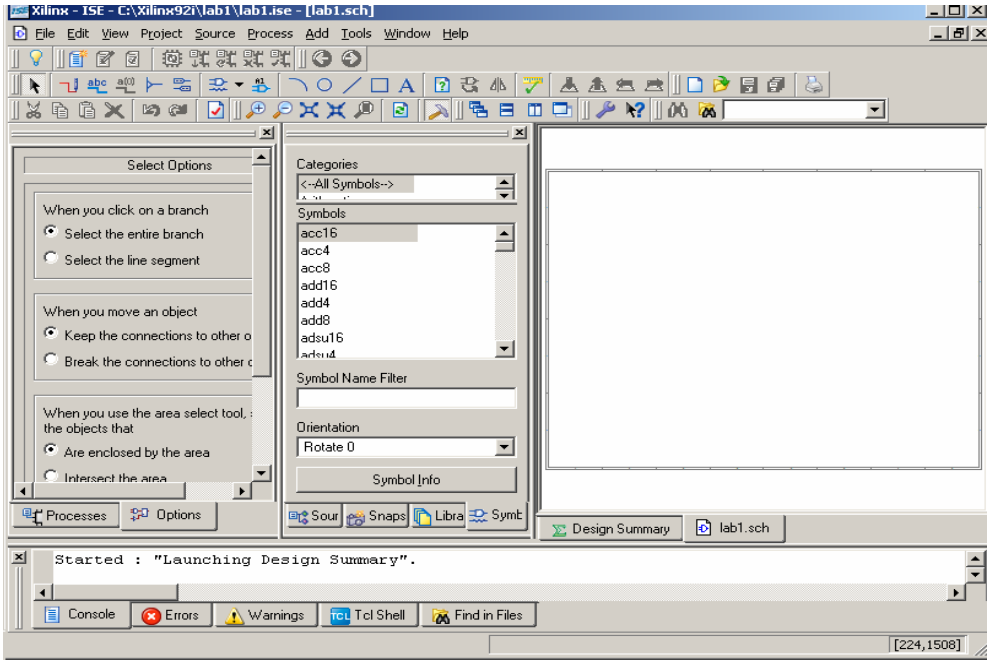


5. A project summary dialog box appears, click the Finish button. The ISE will now create the project as shown below. The **Sources window** shows the name of the project and the name of the top-level source as *.sch with a 3 box-icon to the left of the filename. The **Design Summary window** gives

a summary of the project and the **Processes window** lists all of the processes that are available to this project. The Processes window is an important window. It is this window that will allow the user to simulate and implement an FPGA project.

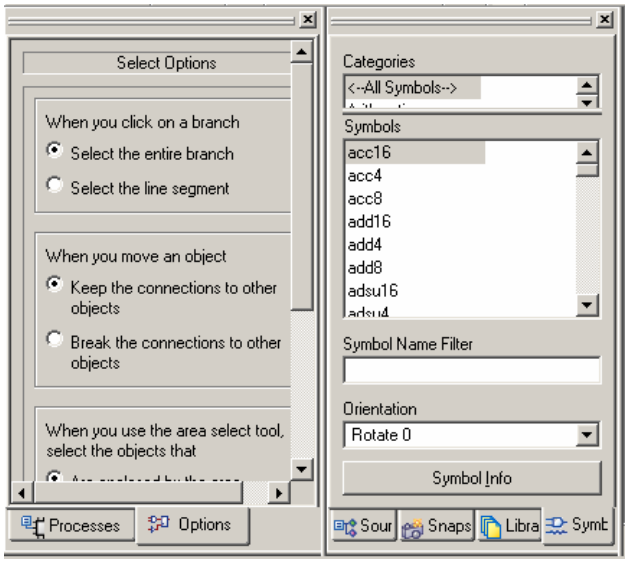


6. Double click on the **lab1 (lab1.sch)** item in the Sources window. The Lab 1 schematic will appear on the right hand side. At any time any of the windows can be floated as separate windows by selecting the Window menu and selecting float menu item.



7. The following figure shows the menu items that are of importance when creating a schematic. Going from left to right, the arrow is the **select** tool, the next set of tools are the **add wire** tool, **add a net name** tool, **rename selected bus** tool, **add a bus tap** tool, **add an I/O marker** tool, **add a symbol** tool and an **Instance Name** tool.

8. Click the **ADD a Symbol** tool. The Sources window will change to the Symbol window and the Processes window will change to the option window. There are two parts to this window one for the category and the other the symbols for that category. Select the **logic category** and select **and2 symbol**. A "b" in the symbol name indicates an inverter is located on an input pin. For example and4b1 is a 4 inputs and gate with one of the inputs connected to an inverter. And2 is a symbol for a two-input and gate. At anytime selecting a symbol and clicking the Symbol Info button an HTML file will appear describing the symbol selected. Now move the mouse over to the schematic window. The symbol will appear next to the mouse as the mouse is moved across the schematic. Select the desired location and click the mouse. This places the symbol in the schematic. Hit the ESC key on the keyboard to cancel any further symbol additions to the schematic.



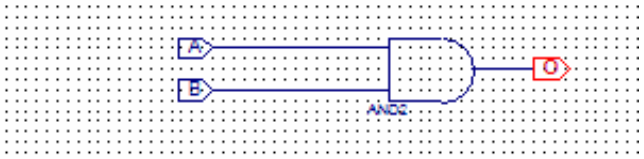
9. In the toolbars there are two magnifying glasses one for zoom in (+) and one for zoom out (-). Use the **zoom in eyeglass** to zoom into the schematic so the symbol is easily visible.

10. Adding wires: select the **Add Wires** tool and click one of the inputs (four boxes appear when the mouse is on the input to the AND gate) to the AND

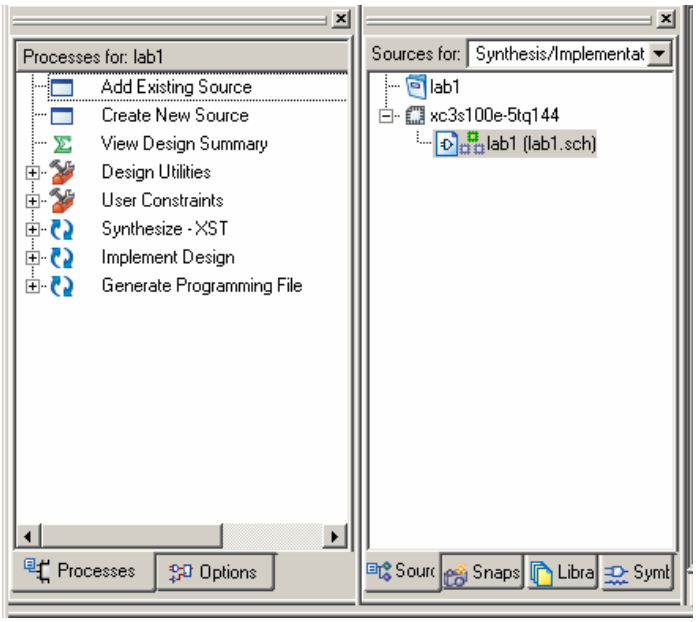
gate and drag the wire to the left and then double click the mouse. One click allows the user to change direction for the wire by applying a tie point on the schematic. To end a wire input double click of the mouse is required. Add a wire to both inputs and the output of the AND gate.

11. Next, I/O markers must be added to the schematic. I/O markers enable inputs and outputs to be tied to physical pins on the FPGA. Select the **Add I/O marker** tool. Move the mouse to the wire ends (selected when four boxes appear) and click the mouse. This will apply an I/O marker for that input/output. Repeat this for both the inputs and the output of the AND gate.

12. The ISE gives default names to the I/O markers. By highlighting the I/O marker and right clicking the mouse and selecting **object properties** allows the user to change the name. Rename the two inputs to "**A**" and "**B**" and the output to "**O**". Do not use "Out" as an I/O marker name as it is a reserve word. The figure below shows the final schematic. Under the **File menu** select the **save option** to save this schematic.

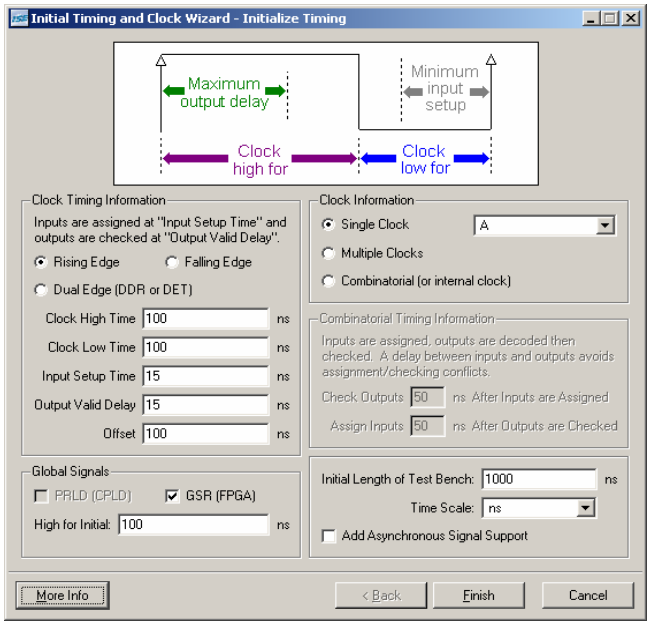


13. Now that the schematic is finished, it must be synthesized. In the **Processes window** select the **processes tab** and in the **Sources window** select the **source tab** as shown below. Before any schematic can be synthesized, the schematic must be saved as shown in Step 12.



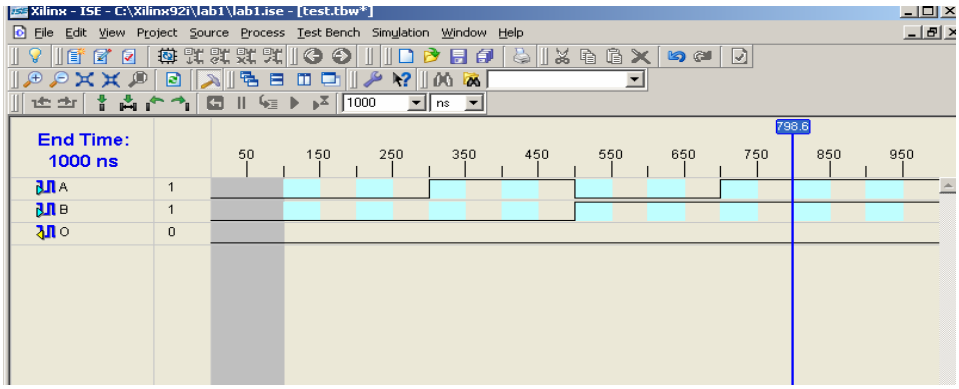
14. The process window lists the processes that are available for this project. Highlight the top-level (indicated by three cubes) schematic "lab1 (lab1.sch)" in the Sources window. To synthesize this project just double click on the **Synthesize - XST** item in the Process window. A blue 3D cube will start spinning and the message window at the bottom will start displaying messages. If there are errors in the synthesis, a red circle with an X will appear. By clicking on the Error tab in the message window at the bottom of the screen, the user can view the error. Likewise, if there are warnings present, the user can view the warning by clicking the warning tab. If the ISE was able synthesize the schematic, a **green check** will appear in front of the Synthesize-XST process. Next, the project must implemented. Double click the **Implement Design** process. If the ISE is able to implement the design, a green check box will appear in front of the Implement Design process. If a warning icon appears, ignore them at this point and continue on with the rest of the steps of this experiment.

15. At this point the user can simulate the design. An input test bench must be created that defines the inputs and outputs to be used in the simulation process. Under **project menu** select **new sources** item. Choose **test bench waveform** source type and select a name for the test bench. The name must be different than the project name. Note: anytime the number of inputs or outputs change, the user should create a new test bench file and the old one removed from the project. The ISE has difficulty updating the test bench file to reflect these changes. Select the next button twice and finally the finish button. The following dialog box appears as shown below. This box allows the user to select either a clock input or combinatorial input as inputs to the schematic. If the user selects one of the inputs as a clock input, then the user can select various timing parameters associated with this clock. For this experiment, the inputs are simply combinatorial inputs. The user should

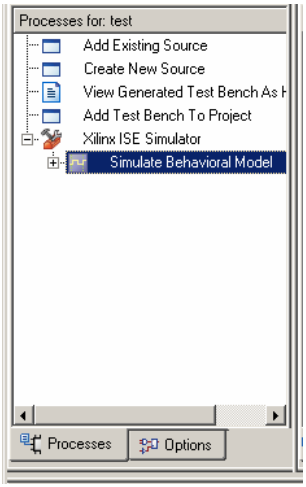


choose the **combinatorial (or internal clock)** option and click the finish button.

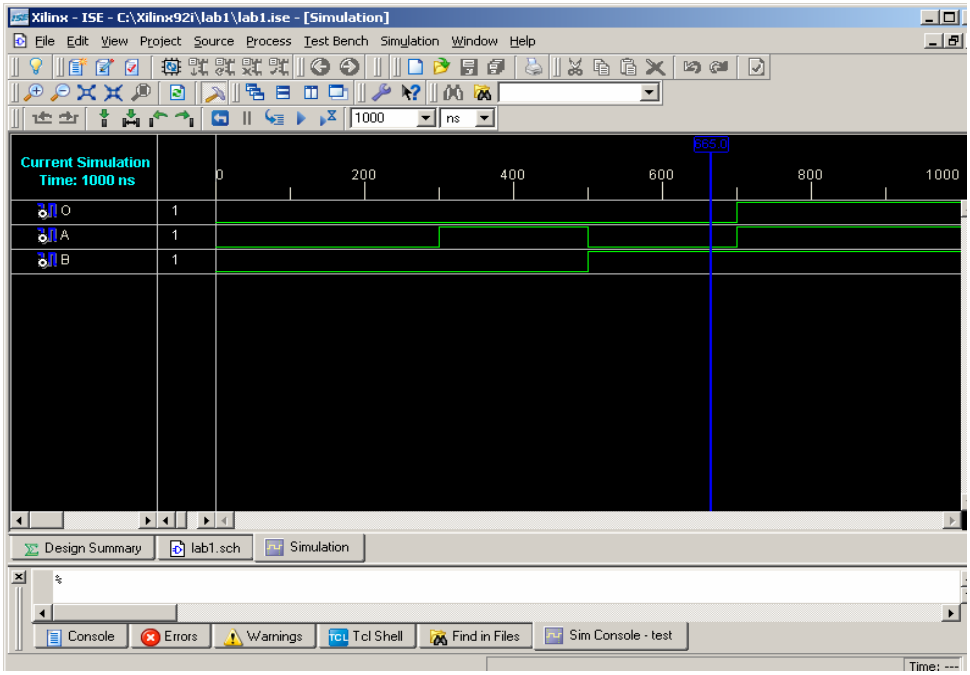
16. The test bench window is now displayed showing both the inputs and the output for the two-input AND gate circuit. By moving over the input waveforms and clicking the mouse the user can change the inputs from a zero to a one and back to zero. Also, if the user right clicks on the mouse and selects set end of test bench, the total time used for the test bench can be changed. The default time is 1000 nano-seconds. For this part of the experiment, the user should define the inputs: **A=0, B=0**, then **A=1, B=0**, then **A=0, B=1**, and finally **A=1, B=1** as shown below. The blue marker can be moved from right to left and the value on the inputs ('1' or '0') can be seen next the input name. When finished, this test bench should be saved using the **File-Save** option. The user should now close the test bench window.



17. To simulate this project, the user needs to select in the **Sources window** in the top drop box selection area the **Behavioral Simulation** option. The default selection is the Synthesis/ Implementation option. Highlight the test bench file (*.tbw) listed in the Sources window. Next, in the Processes window click the plus symbol next to the Xilinx ISE Simulator Process. The **Simulate Behavioral Model** will be displayed. Double click on this item to start the simulation.

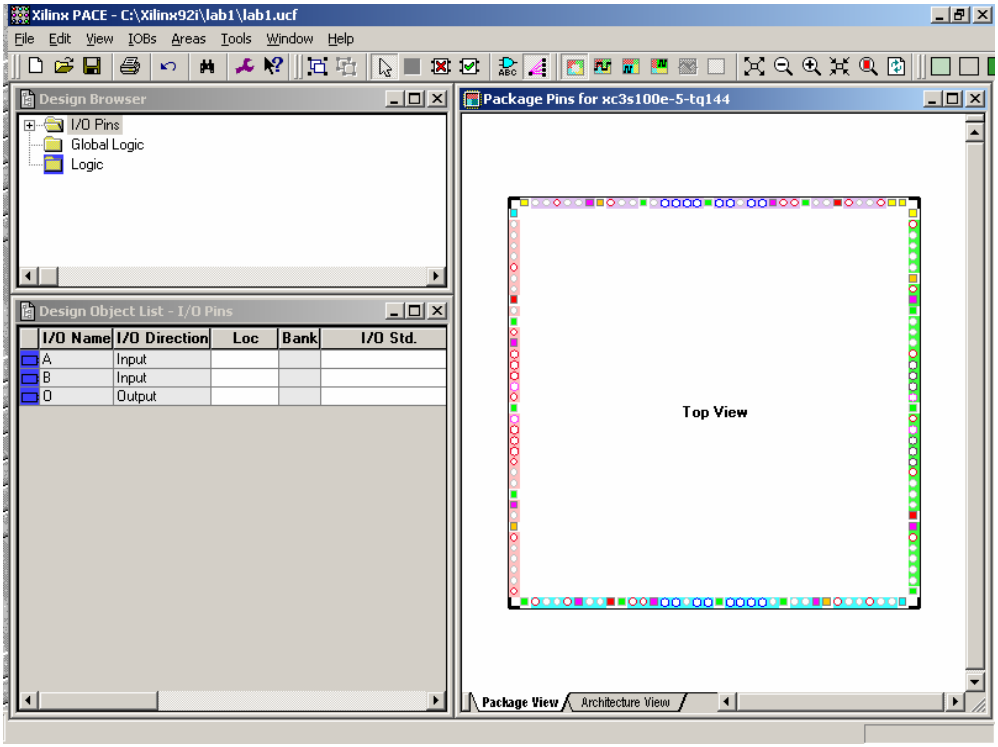


18. If there are no errors, the simulation window appears showing the inputs as defined by the test bench file and the appropriate output(s) as shown below. The magnifying glasses can be used to zoom in or out. The enlarged view of the simulation window was obtained by closing the Processes and Sources windows. These windows can be reopened using the View menu and by selecting these windows. **The user should check the output and verify that this output follows the truth table for a two-input AND gate.** In this window, make sure that the simulation time starts at '0' ns, else move the scroll bar to the right and back to the left. **Remember to save a screen shot of this window for your report.** When the user is finished with the simulation window, the user can close it under the File menu - Close item option.



19. The last step is to assign the I/O pins in the schematic to the physical pins on the FPGA and to generate the FPGA programming file. In the **Sources window** select the **Synthesis/ Implementation** option. Next, click on the plus symbol on the **user constraints item** in the **Processes window**. Double click on **Create Area Constraints** Item. A dialog box appears asking the user to create *.UCF file. Click on the Yes button. If this dialog box does not appear, rerun the synthesize XST and Design again. The **PACE program** will open as shown below. Select the **package view** tab at the bottom. The input and output markers (blue box) from the schematic are listed in the **Design Object List Window** on the left. **The user should simply drag the blue box next to the input/output to the desired pin.** From Appendix D, for Basys board, **SW0 is located on pin 38**, **SW1 is located on pin 36** and **LED0 is located on pin 15**. From Appendix D, for Basys 2 board, SW0 is located on pin P11, SW1 is located on pin L3 and LED0 is located on pin M5. Note the package pin number and type by

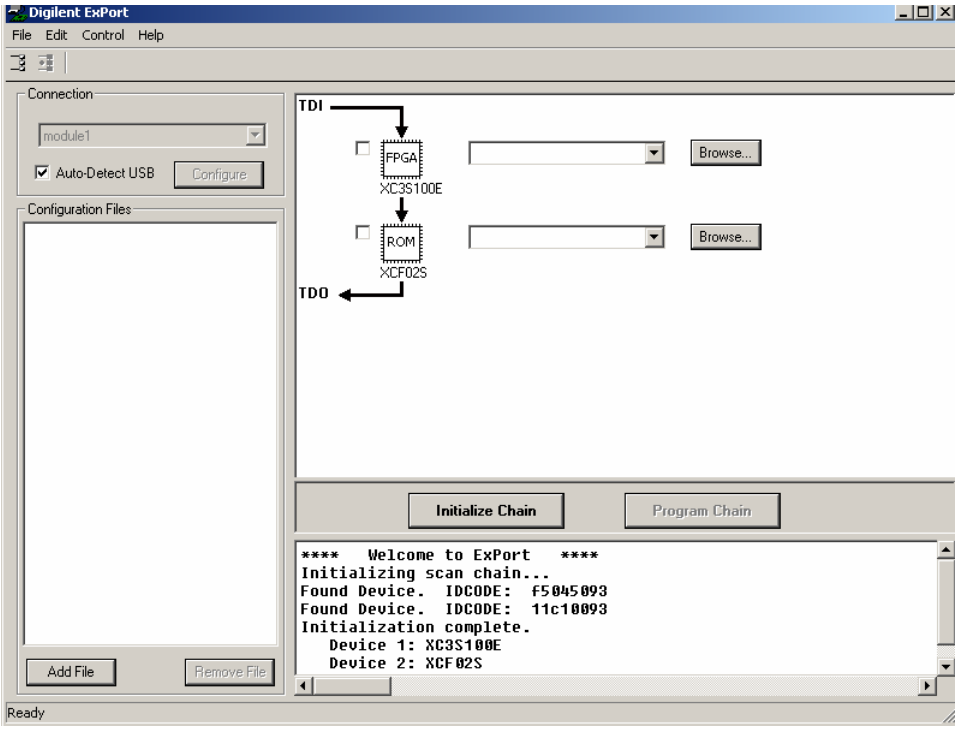
moving the mouse over the pins. Once you have identified a pin number, move the blue box from the Design Object List Window to the pin. Select pins for A (pin 38), B (pin 36), and 0 (pin 15). Save this configuration file using the **File-Save** option. The Pace program can now be closed.



20. The **Implement Design** will have to be rerun. This is indicated by a question mark next to this process in the Processes window. Double click on the Implement Design option. Make sure you have a green check mark in front of the Implement Design. Finally, double click on the **Generate Programming File**. This generates the *.BIT program needed to program the FPGA. A Xilinx WEBTalk window will open. Close this window. The programming file for the FPGA will now be located in the project folder.

21. Now open the **EXPORT program** (look for a short cut on the desktop or Windows Start – All Programs – Digilent – Adept – ExPort). On some computers, this program is called 'Adept' inside the folder called 'Adept'. This program will be used to download the *.BIT file to the BASYS board. **Make sure that the BASYS board is plugged into the desktop computer via an USB port.** Once EXPORT is running, select the **Initialize Chain** button. Two devices appear. The first one is for the FPGA and the second one is for the FPGA configuration ROM. Using the **Browse button** select the ***.BIT file** in the project directory for both devices. The default location of this file is C:/XILINX92I/project name/*.bit. A dialog box will appear stating that the file is unknown. Ignore this dialog box and select the Yes button. The *.BIT file will appear in the file name boxes for both devices. The last step is to program the devices selecting the **Program Chain** button.

If successful, a dialog box will open indicating that both devices have been programmed successfully. Select the OK button to proceed. **The FPGA has now been programmed.**



22. Check the operation for the two-input AND GATE and fill in the following table. Remember we have selected switch SW0 for input A, switch SW1 for input B and led LED0 for the output O. Toggle the switches for the states shown in the table below and fill in the output by observing LED0. This table should confirm the truth table for a two-input AND gate.

SW0	SW1	LED0
0	0	
0	1	
1	0	
1	1	

Part 2. Implementation of the OR, NAND, NOT and Ex-OR GATES using Xilinx's ISE

1. Repeat steps 1-22 in part one of this experiment, but this time for a two-input NAND gate (NAND2) under the logic category in the Symbols option of the Sources window. Open a New Project, do not try to include a new schematic file in the same project used for the 2 input AND gate.

2. Repeat steps 1-22 in part one of this experiment, but this time for a two-input OR gate (OR2) under the logic category in the Symbols option of the Sources window.

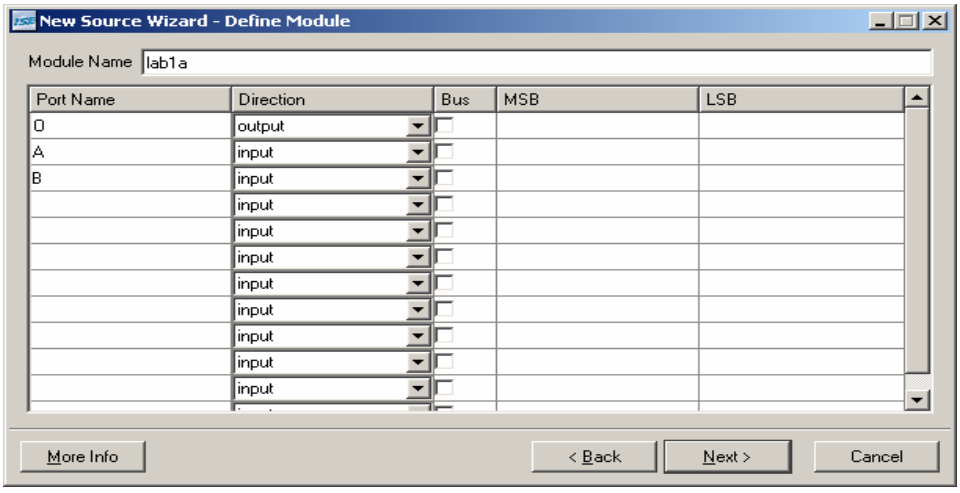
3. Repeat steps 1-22 in part one of this experiment, but this time for a two-input exclusive-or gate (XOR2) under the logic category in the Symbols option of the Sources window.

4. Repeat steps 1-22 in part one of this experiment, but this time for an inverter (INV) under the logic category in the Symbols option of the Sources window. Use this truth table for the inverter circuit.

SW0	LED0
0	
1	

Part 3. Implementation of an AND Gate using VERILOG and Xilinx's ISE

1. Repeat steps 1-3 of part 1 to open a new project and to select a new source. This time select the source as a **VERILOG module** instead of a schematic. After selecting the VERILOG module, by selecting the Next button the following input/output dialog box will appear. For this part of the project two inputs and one output pins must be defined. Enter in the port name O and select the direction to be an output. Next, define inputs A and B in the port name column and select direction as input. Finally, select the Next button.



2. Click the finish button. A dialog box appears and states that the VERILOG file does not exist. Click the Yes button to create the file. Click the Next button to continue. The Add Source dialog box will now appear. This dialog

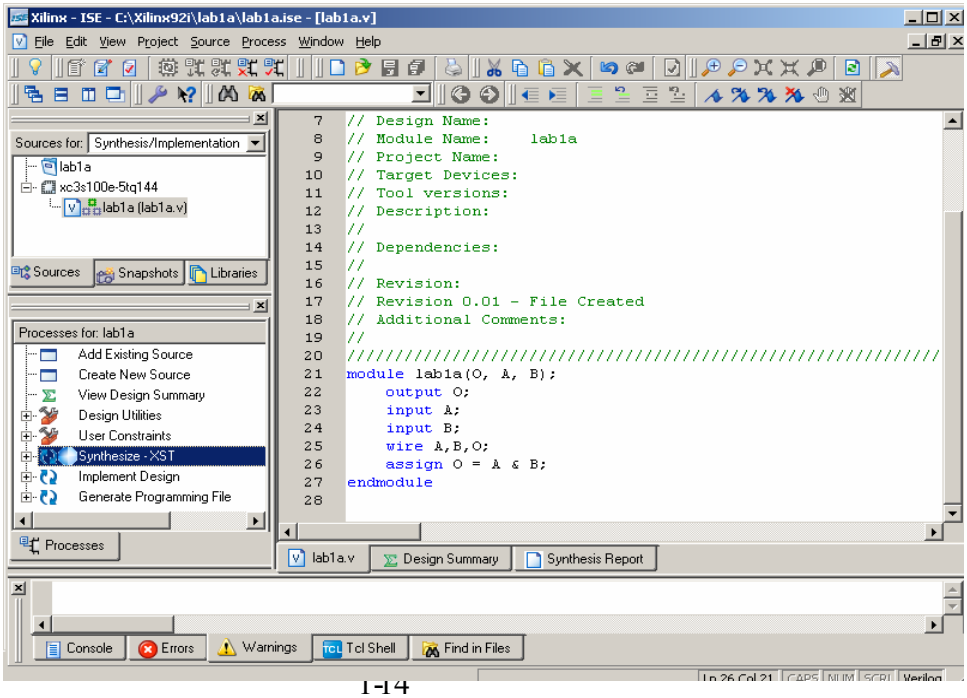
box allows the user to add an existing source to the project. For Experiment #1, click the Next button.

3. A project summary dialog box appears, click the Finish button. The ISE will now create the project as shown below. The Sources window shows the name of the project and the name of the top-level source given by the 3 box-icon next to the VERILOG source file name *.v. The design summary window gives a summary of the project and the process window lists all of the processes that are available. The process window is an important window. It is this window that will allow the user to simulate and implement the FPGA.

4. Click on the *.V tab in the bottom of the design window. The VERILOG source file appears where the schematic window was located in part 1. Note that the module shows the defined inputs and outputs that were selected previously. The syntax for VERILOG is very similar to C programming language. All lines must end in a semicolon, and all comments use either // or / */. One difference is that all inputs and outputs need two definitions. The first defines if the I/O connection is an input or an output port and the second defines if this input is a "wire" or a register "reg". For this experiment, only wires will be used. Also, to set an output equal to an input the “assign” function must be used. A summary of the VERILOG syntax is given in Appendix E.

5. The '~' symbol is the NOT operator, the ‘|’ symbol is the OR operator, the ‘&’ symbol is the AND operator and ‘^’ symbol is the EXOR operator. Add the following lines to the VERILOG program after the input and output definitions:

```
wire a, b, o;
assign o = a & b;
```



This defines the code required to implement a two-input AND gate module called lab1a:

```
module lab1a(O, A, B);
   output O;
   input A;
   input B;
   wire A, B, O;
   assign O = A & B;
endmodule
```

6. Repeat Steps 13 through 20 in part one to implement, design, simulate and download this two-input AND gate using the VERILOG language. The only difference between creating a schematic and using VERILOG to design is the source file. Wherever the steps refer to the schematic and the *.SCH file, replace the wording with the VERILOG and the source file *.V.

7. Download the *.BIT file to the FPGA using EXPORT as defined in step 21 of part one. Verify that the two-input AND gate is functioning as designed on the BASYS board as described in step 21.

8. Repeat steps 1-7 in part three of this experiment, but this time for a two-input NAND gate (~ (A & B)). Refer Appendix E for VERILOG syntax.

9. Repeat steps 1-7 in part three of this experiment, but this time for a two-input OR gate.

10. Repeat steps 1-7 in part three of this experiment, but this time for a two-input exclusive-or gate.

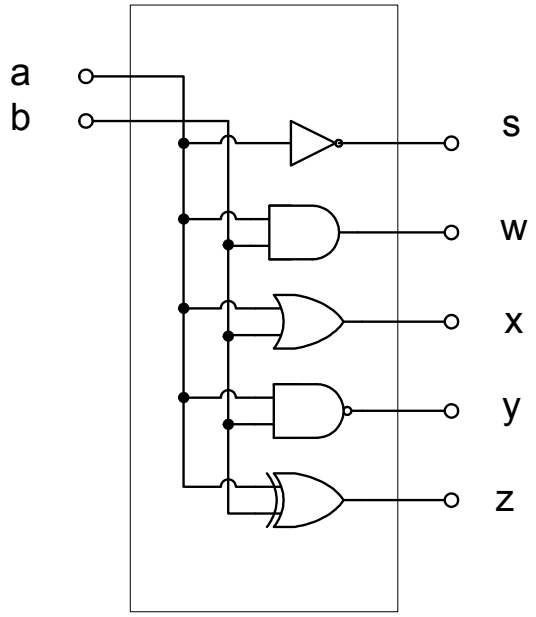
11. Repeat steps 1-7 in part three of this experiment, but this time for an inverter (INV). Use this table for the inverter circuit.

SW0	LED0
0	
1	

Part 4. Implementation of a two-input five-output logic circuit.

1. Implement the two-input and five-output logic circuit using schematic capture in the ISE. Simulate the five-outputs for all possible inputs. Download the finished design to the BASYS board and verify its functionality using the following table.

A	B	LED0 (S)	LED1(W)	LED2(X)	LED3(Y)	LED4(Z)
0	0					
1	0					
0	1					
1	1					



2. Implement the two-input and five-output logic circuit using VERILOG in the ISE. Simulate the five-outputs for all possible inputs. Download the finished design to the BASYS board and verify its functionality using the following table.

A	B	LED0 (S)	LED1(W)	LED2(X)	LED3(Y)	LED4(Z)
0	0					
1	0					
0	1					
1	1					

Report: Please follow the procedures in this laboratory manual for writing the report for this experiment.

1. Summarize in your own words the steps required to complete Part one of this experiment. Include screen shots (Alt-Printscreen to place the screen into the clipboard) in your report to show the steps taken to complete part one.

2. Include and discuss the simulated results from part one for the two-input AND gate.

3. Give the table from part one step 21 for the two-input AND gate.

4. Show the wiring diagram of the AND gate as connected to SW0, SW1 and LED0 include the pin numbers of the FPGA used.

5. Repeat Steps 2, 3, and 4 in the reporting section for the OR, NOT, NAND and the EXOR gate simulations as implemented in part two of this experiment. Do not forget the input-output (truth) tables for the OR, NAND and the EXOR gate. The inverter (NOT gate) used a different truth table.

6. Repeat Steps 2, 3, and 4 in the reporting section for the AND, OR, NOT, NAND and the EXOR gate simulations as implemented in part three of this experiment using the VERILOG design language. Do not forget the input-output tables for the AND, OR, NAND and the EXOR gate. The inverter used a different table.

7. Repeat Steps 2, 3, and 4 in the reporting section for the two-input and five-output logic circuit given in part four of this experiment. Include the simulation results and the input-output tables for both the schematic and VERILOG implementations.

8. Do not forget the Test Plan and the Design Specification Plan required for this experiment. Refer to the Introduction section of this manual for details of what is expected in this section. Steps in this experiment that are very similar can use the same plans.

EXPERIMENT #2

Simple Combinational Logic

Objectives:

- To further investigate the operation of Xilinx's ISE by implementing a simple combinational logic circuit.
- To design a circuit from a verbal description namely, a burglar alarm controller circuit.

Discussion:

A simple burglar alarm is to be designed and simulated using Xilinx's ISE development tools for FPGA. The burglar alarm is to sound if and only if the alarm is turned on and the window is open or the door is open. Figure 2-l shows a block diagram of such a controller. Be careful on the interpretation of the operation of the burglar alarm, especially the placement of the AND gate and the OR gate. To further describe the conditions required to trip the alarm, Figure 2-2 lists all of the conditions in which the alarm will sound.

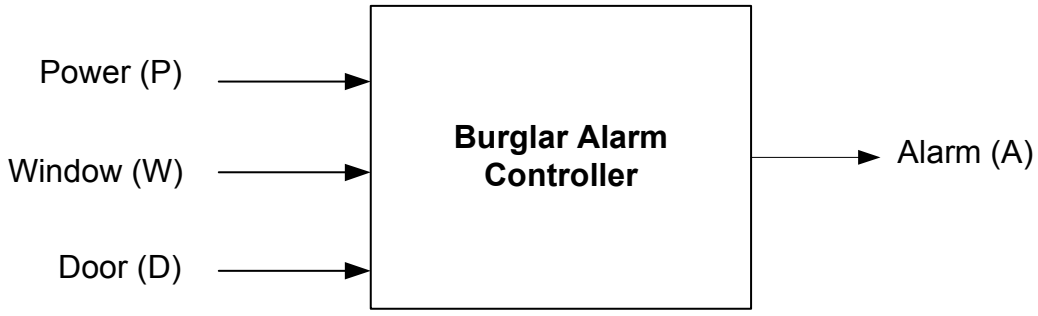


Figure 2-1: Block Diagram of Burglar Alarm

Power	Window	Door
On	Closed	Open
On	Open	Closed
On	Open	Open

Figure 2-2: Conditions in Which the Alarm Will Sound

To obtain the Boolean expression for the controller circuit, certain conditions must be interpreted as either a TRUE or a FALSE. For this circuit, anything that is **open or on will be considered TRUE** while anything that is **closed or off will be considered FALSE**. Let P represent the power state, W represent the window state, D represent the

door state, and A represent the alarm state. In other words, the alarm (A) will be a function of P, W, and D:

$$A = F\ (P, W, D).$$

Now that all of the information has been coded to either TRUE or FALSE, the logic equations can be written. The first line in Figure 2-2 states that the alarm will sound when the power is on (TRUE), the window is closed (FALSE) and the door is open (TRUE). Note the ANDs in the previous sentence. In order for the Alarm to sound (or be TRUE) then the three conditions must produce a TRUE value. At this point the incomplete function will be:

$$A(PDW) = PDW'+...$$

The ‘ next to the W represents an *Inverting* logic operation of the variable which effectively makes the FALSE a TRUE.

Going through the same procedure for the other two lines from Figure 2-2, the following equation can be obtained:

$$A(PDW) = PD'W + PDW'+PDW$$

Any of the three conditions can be true in order for the alarm to sound so the three parts are joined together by an *OR* which is represented by a "+".

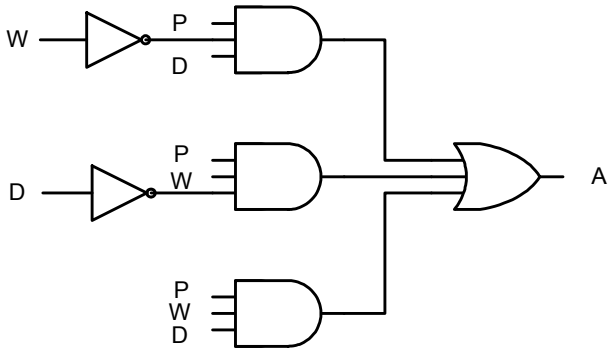


Figure 2-3: Logic Diagram of Burglar Alarm

The resulting **unsimplified** circuit is presented in Figure 2-3. Notice, that the invert operation is represented by the triangular symbol, the *AND* is represented by the rounded symbol, and the *OR* is represented by the pointed symbol.

Pre-Laboratory Assignment:

1. Read this experiment carefully to become familiar with the requirement of this experiment.

2. In your laboratory notebook, fill in the truth table for the burglar alarm for all possible inputs for P, W, and D.

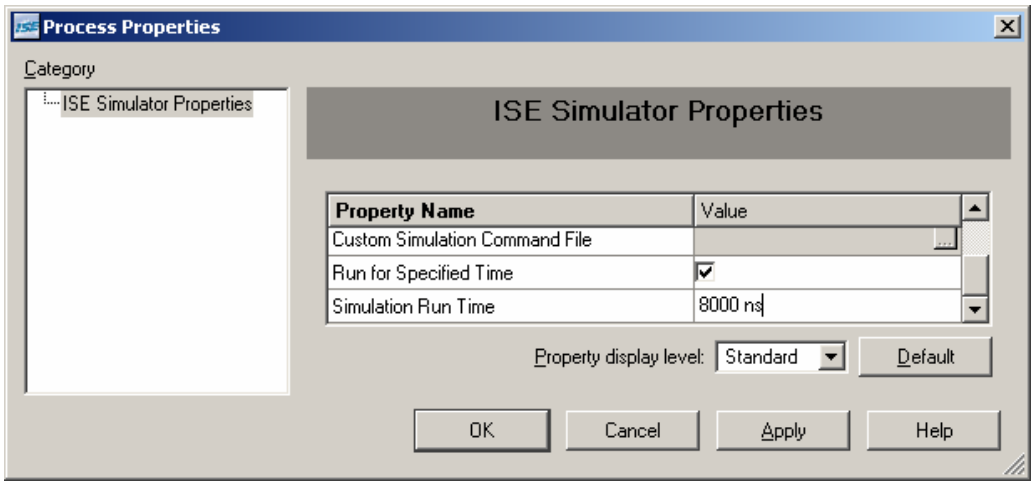
3. In your laboratory notebook, write the Boolean expression describing the burglar alarm controller:

4. Draw the logic diagram for the burglar alarm.

5. Give the schematic and board layout for parts 1 and 2 of this experiment.

Procedure:

1. Simulate the burglar alarm using Xilinx's ISE using the schematic capture tool and the simulation tool. Generate printouts of the schematic circuit, timing diagram and test bench inputs. Be sure that all eight input conditions for P, W and D are met. You will need to set the test bench end time to 2000 nano-seconds and the simulation end time also to 2000 nano-seconds to have enough time cycle through all possible inputs. To set the test bench end time right click on the mouse in the test bench program with the test bench file open and under Set End of Test Bench set the end of the test bench time to 2000 nano-seconds. To select the end time for the simulation, right-click the Simulation Behavioral Model process under the Xilinx ISE Simulator process in the Processes window (Behavior Simulation chosen in the Sources window). Select the property menu item. Next, move the slider bar until the Simulation Run Time is shown under the Property Name. Highlight the Value and change it to 2000 ns.



2. Build the above un-simplified design using *AND's*, *OR's*, and inverters using the experimental board in the lab and using the standard 74LSXXXX transistor-transistor-logic (TTL) parts as given in Appendix A. Use three switches for P, W, and D and an LED for A.
3. Verify that the built circuit behaves exactly as the problem statement describes by verifying all possible inputs.

4. Repeat Step 2 but this time implement the burglar circuit using the FPGA BASYS board using schematic capture tool in the ISE. Assign P to SW0, W to SW1, D to SW2, and A to LED0 (Refer Appendix D). Download the FPGA configuration file to the BASYS board using the EXPORT program as described in Experiment #1 of the laboratory.

Questions:

(To be incorporated within the Conclusion section of your lab report)

1. Can the logic be simplified in any way? Explain.
2. How would the controller logic be simplified if the power is always on?
3. Have you met all the requirements of this lab (Design Specification Plan)?
4. How should your design be tested (Test Plan)?
5. The function is represented by the ORing of the terms associated with the 1's in the truth table. Can an expression for A be found which is derived by the ANDing of terms associated with the 0's in the truth table?
6. Write the Boolean expression describing a burglar alarm, which also sounds when a sensor has been crossed.
7. Discuss which design method is easier to implement (FPGA design or using Small-Scale-Integrated (SSI) circuits such as the 74LSXXXX TTL logic parts).

EXPERIMENT #3

Multi-Function Gate

Objective:

To design and build a Multi-Function Gate using the Xilinx's FPGA tools and to document the design. Xilinx's FPGA tools will be used to design, simulate and implement this multi-function gate to the BASYS Board FPGA.

Discussion:

The Multi-Function gate in this experiment is a double input, single output gate that can be instructed to perform four different logic operations by placing a control value on the inputs X and Y. The instruction to this Multi-Function Gate is provided by the operation select bits, which thus determine how the gate will act. *Figure 3.1* shows the block diagram of such a gate. A and B form the data inputs and F the single output. X and Y are the operation select lines.

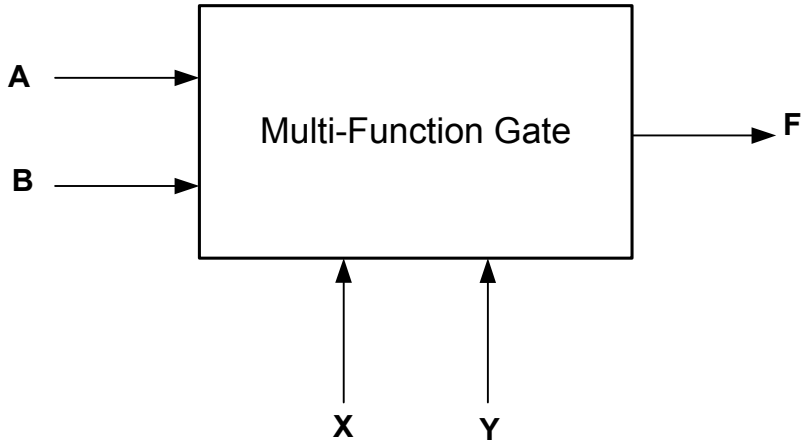


Figure 3-1: Block Diagram of Multi-Function Gate

Design Specifications:

The circuit should be synthesized such that for a given X and Y, F is a certain function of A and B. When X=0 and Y=0, the multi-function gate acts as an AND gate, therefore, F=A AND B. When, X=0 and Y=1, the multi-function gate acts as an OR gate. When X=1 and Y=0, the multi-function gate acts as a NOR gate. Finally, when X=1 and Y=1 the multi-function gate acts as a NAND gate. The function codes are summarized in the table below.

X	Y	Function
0	0	AND
0	1	OR
1	0	NOR
1	1	NAND

Figure 3-1: Function codes for the multi-function gate

Pre-Laboratory Assignments:

1. Read this experiment carefully to become familiar with the experiment.

2. Represent the output F as a function of X, Y, A and B on a truth table.

3. Write the minimum logic expression, as a sum-of-products for the function F.

4. Draw logic diagrams for the above expressions using AND's, OR's, and Inverters.

Procedure:

1. Design and simulate this circuit using Xilinx's ISE using the schematic capture tool and the simulation tool. Generate printouts of the schematic circuit, timing diagram and test bench inputs. Be sure that all the sixteen input conditions have been met for A, B, X, and Y. You will need to set the test bench end time to 4000 nano-seconds and the simulation end time also to 4000 nano-seconds to have enough time to cycle through all possible inputs. See Experiment #2 (pg 3) on how to set the test bench end time and the simulation end time.

2. Now implement the design in Step 1 of this procedure and configure the FPGA so that A is on SW0, B is on SW1, X is on SW6, and Y is on SW7. Also use LED7 for the output F. Appendix D gives the pin details for the switches and the LEDs. Download the FPGA configuration file to the BASYS board using the EXPORT program as described in Experiment #1 of the laboratory. Verify that for all 16 inputs the output F matches the truth table in Step1.

3. Repeat Step 1 but this time design and simulate the Boolean circuit using the VERILOG language in the ISE. Generate printouts of the VERILOG file, timing diagram and test bench inputs. Be sure that all the sixteen input conditions have been met for A, B, X, and Y. Set the test bench end time to 4000 nano-seconds and the simulation end time also to 4000 nano-seconds. You may need additional wires beyond the input and output wires (eg. Wire a, b, c, d, e, etc).

4. Implement the design in Step 3 of this procedure on the BASYS board using the same configuration given in Step 2. Verify that for all the 16 inputs the output F matches the truth table in Step 1.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. Can this Multi-Function Gate be operated as an Inverter? If yes, explain how.

2. Will the change in the number of inputs or outputs affect the number of operation select lines? Explain.

3. Will the change in the number of functions alter the number of operation select lines? Explain.

4. Have you met all the requirements of this lab (Design Specification Plan)?

5. How should your design be tested (Test Plan)?

EXPERIMENT #4

Three-Bit Binary Adder

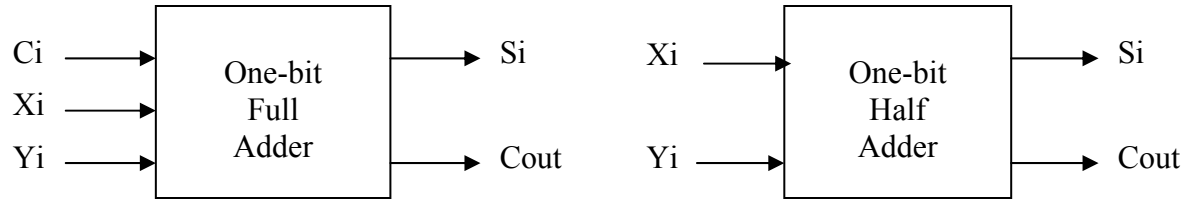
Objective:

- To design a Binary Adder, which will add two binary words, three bits each, using discrete gates.
- To introduce iterative cell design techniques.

Discussion:

A Binary Adder can be designed as a parallel or serial adder with accumulation. For this experiment, the **parallel adder** will be designed to add two binary digits, three bits each, X (expressed as $X_2X_1X_0$) and Y (expressed as $Y_2Y_1Y_0$). The adder can be designed via the "brute force" method in which three, six variable Karnaugh maps are used to implement the functions representing the outputs of a three-bit addition. However, this method is not the most efficient so a different design approach, called the **iterative cell technique**, will be used. In the iterative cell technique, two binary numbers are presented in parallel to the cell as inputs. The rightmost cell adds the least significant bit X_0 and Y_0 to form a sum digit S_0 and carry digit C_0. The next cell adds the carry C_0 to bits X_1 and Y_1 to form a sum digit S_1 and a carry digit C_1. The last cell adds the carry C_1 to bits X_2 and Y_2 to form a sum digit S_2 and a carry digit C_{out}.

To design a network, a typical cell should be designed which adds a carry C_i to bits X_i and Y_i to generate a sum digit S_i and a new carry C_{out} as shown below. The circuit that realizes this function is referred to as a **full adder** cell. Please note that the operation on the least significant bits of X and Y does not include a carry-in signal. Thus a **half adder** circuit can be used for the rightmost cell. The diagrams below illustrate the functional blocks for a one-bit full adder and a one-bit half adder.



Pre-Laboratory Assignments:

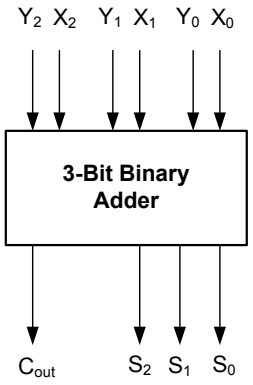
1. Read this experiment carefully to become familiar with the requirements for this experiment.

2. Prepare and complete a truth table for the full adder cell. Transfer this information to a Karnaugh Map and obtain minimum expressions in both sum of products and product of sums forms.

3. Use Boolean algebra to reduce sum of products expression to a more workable expression. (i.e. *XOR's*).

4. Represent the full adder and the half adder as logic diagrams.

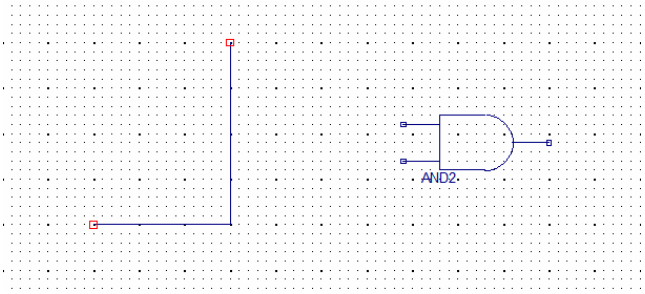
5. Prepare a schematic diagram of the complete three-bit adder circuit (your best design). A logic schematic includes not only pin assignments but switch and LED assignments as well. See Appendix D for pin details used by the BASYS and BASYS 2 boards.



Procedure:

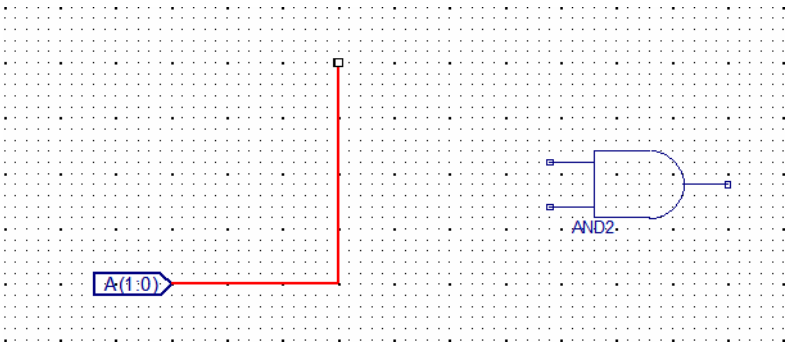
1. Use Xilinx's schematic tool to implement the design obtained in step 4 of the pre-laboratory procedures. For this experiment, the use of buses in the ISE will be introduced. A total of three buses (three bits each) will be used for X, Y, and S. Also a carry out will be needed.

2. To create a bus in the ISE, the **Add Wire tool** and the **Bus Tap tool** will be used. A simple 2 input AND gate will be used as an example. Using the **Symbol** tab and the **Add Wire** tool add a two-input AND gate and a wire to the schematic as shown below.

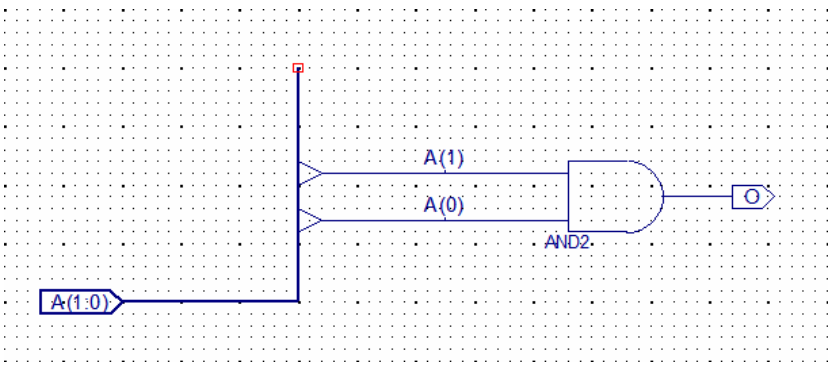


To change the wire to a bus, the only thing that is required is to change the wire name. Click on the wire and right click the mouse and select the **Object Property**

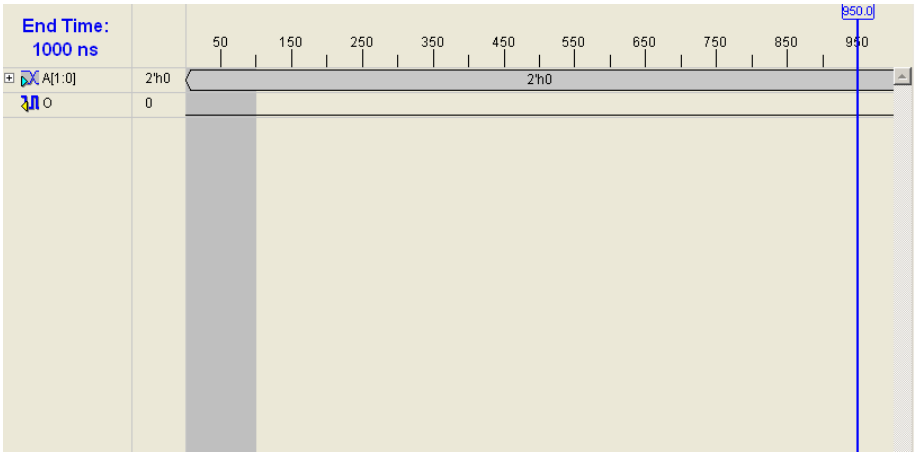
item. To change the wire to a bus the name needs to be appended with [MSB:LSB]. For example, a two bit bus name can be **A[1:0]**. Name the wire as A[1:0] and hit OK. Note how the wire width has changed and has become thicker indicating a bus. An I/O marker can be added to the bus by placing the I/O marker on the bus. Note the name of the I/O marker in the Figure below as A(1:0). This indicates that this bus is two bits wide.



To access the individual bits in a bus, the bus tap tool is used. Click on the **Bus Tap tool**  . Notice how the mouse cursor has changed to the bus tap tool symbol. Make sure the **Options tab** next to the Processes tab is selected. Do not insert the bus tap in the schematic, click the '+' pointer of the mouse on the bus (Bus A in this case) to select the bus. Once the bus is selected, observe how the **Selected Bus Name** in the Options tab displays A(1:0) and the **Net Name** displays A(1). This indicates that the bus tap refers to the bit A(1), the most significant bit in this case. You can change the Net Name to A(0) or A(1) by typing in the box or moving the arrows on the right side of the name box. The **Orientation** in the Options Tab can be used to change the orientation of the bus tap depending on which side of the bus tap is placed on the bus. The bus symbol must be oriented so it connects the bus as shown in the Figure below. After making these changes in the Option Tab place the right end of the bus tap on one of the inputs of the AND gate (look for the four box symbol when the bus tap is aligned to the AND gate input). The bus tap automatically extends and connects to the bus. The name A(0) or A(1) should appear on this bus tap as shown in Figure below. Repeat this process for the other input.

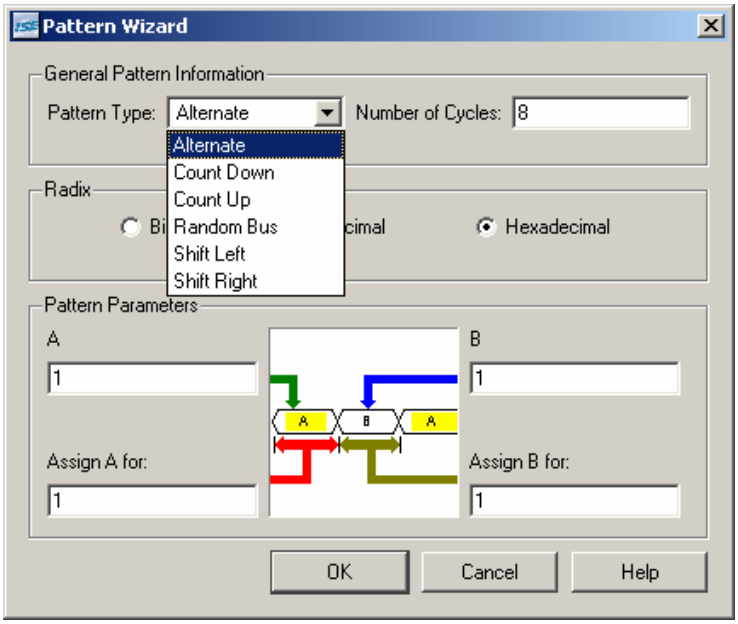


The Test Bench tool of the ISE will also change to reflect a bus as shown in the following Figure. The plus button next to the bus name allows the user to expand the bus so that the individual bits of the bus can be viewed and modified as you have previously done.



Double clicking on the bus waveform brings up the **Set Value** box. Click on the **Pattern Wizard** dialog box. This brings up the Pattern Wizard box as shown in the Figure below. The many options in the box show the advantage of using a bus. The options allows the user to have the bus count up or down, assign random values, shift the values to the left or right, or alternate the bus between two different values. Select an appropriate option in the box. Save, close and view the **Behavioral simulation** as you have done in the previous experiments.

When using the **Create Area Constraint** option under the **User Constraint** process, all the bus bits will be shown so they can be assigned to the I/O pins of the FPGA.



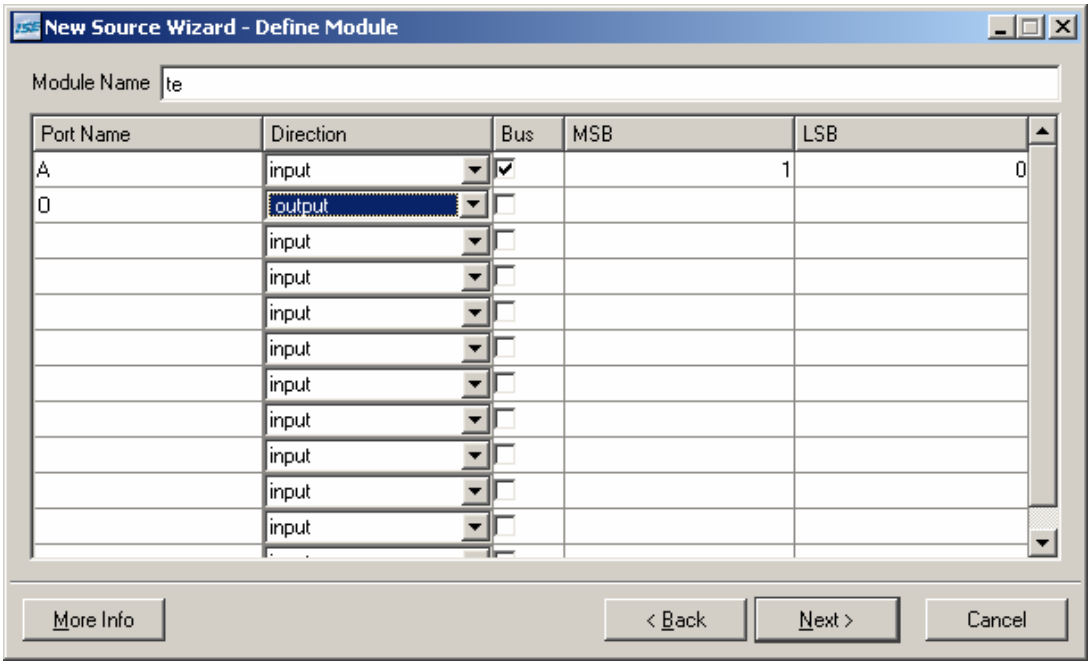
Now that you have learned how to create and use a bus, implement and simulate the adder circuit using bus for X, Y and S. Verify that the adder adds two binary numbers correctly and produces the correct carry output.

3. Assign SW0-SW2 to X0 - X2, SW5-SW7 to Y0 -Y2, LED0 - LED2 to S0 - S2 and LED7 to C_{out} (carry out) on the BASYS board. Download your design to the BASYS board using the EXPORT program.

4. Verify that the three-bit adder adds the two binary numbers correctly on the BASYS board. Include this verification in your lab report.

5. Use Xilinx's VERILOG language to implement the design obtained in step 4 of the pre-laboratory procedures. Using a bus in VERILOG is very similar to naming buses with the schematic tool.

6. When you create a new module, select 'Verilog Module' as the file type. Xilinx will pop a window like the one in the figure below. This window allows you to specify all the inputs and outputs of the module. In this window, you should also specify the number of bits in each input/output. In the figure below, there is an input 'A' of 2 bits and an output 'O' of 1 bit. The terms MSB (Most Significant Bit) and LSB (Least Significant Bit) are the leftmost and rightmost bits in the signal and are used to define the signal width. For example, a signal of 8 bits will have MSB=7 and LSB=0.



After we provide the input/output signals as in the figure above, Xilinx creates an empty code template, as shown below:

```
module temp(A, O);
   input [1:0] A;
   output O;
endmodule
```

The name of the module is 'temp', which we have provided earlier to Xilinx. The next two lines are the declarations of the input 'A' and the output 'O'. The last line contains the keyword 'endmodule' that should be the last line in a Verilog code.

After Xilinx creates the Verilog template, we write the code of the module. In the code below, we'll make the output 'O' equal to the AND of the two bits in the input 'A'. We did this using the line 'assign O = A[0] & A[1];'. First, notice the use of the 'assign' keyword. We use this keyword since Verilog makes a continuous assignment. It means anytime A[0] or A[1] change, the output O will re-evaluate automatically. This is different from C code where the statement is done once. Therefore, we use the keyword 'assign'. The line of code also shows how we can access one signal in a bus. We used A[1] and A[0] to access the two bits in the input 'A'.

```
module temp(A, O);
   input [1:0] A;
   output O;
   assign O = A[0] & A[1];
endmodule
```

Now that a bus can be created; implement and simulate the three input binary adder of step one. Verify that the adder adds two binary numbers correctly and produces the correct carry output.

7. Assign SW0-SW2 to X0 - X2, SW5-SW7 to Y0 -Y2, LED0 - LED2 to S0 - S2 and LED7 to C_{out} carry out on the BASYS board. Download your design to the BASYS or BASYS 2 board using the EXPORT program.

8. Verify that the three-bit adder adds the two binary numbers correctly on the BASYS boards. Include this verification in your lab report.

Questions:

(To be incorporated in the Conclusion section of your laboratory report.)

1. Using full adder and half adder block diagrams, draw an 8-bit adder diagram.

2. Comment on the feasibility of designing an 8-bit adder using the brute force method.

3. Identify the advantages and disadvantages of the brute force method.

4. Identify the advantages and disadvantages of the iterative cell method.

5. Have you met all the requirements of this lab (Design Specification Plan)?

6. How should your design be tested (Test Plan)?

EXPERIMENT #5

Multiplexers in Combinational logic design

Objective:

The goal of this experiment is to introduce multiplexers in the implementation of combinational logic design. In addition, procedural programming in VERILOG will be introduced to the student.

Discussion:

Multiplexer or Selector: The basic function of the circuit is to select one of several inputs to connect to a single output line. Typical multiplexers (MUX's) come in 2:1, 4:1, 8:1 and 16:1. A MUX is composed of n selections bits that maps 2^n inputs to a single output. A TTL series 8:1 MUX is 74151, which is an eight to one (8:1) multiplexer. Examine the spec sheet in Appendix A for 74151. The data select lines are S_2, S_1, and S_0 (often called the control / selection lines). Each of the 8 possible combinations of S_2, S_1, and S_0 selects one of the 8 *AND* gates which maps one of 8 possible inputs to the output Y (see page 8 of Appendix A). The output of the selected *AND* gate will be the same as the input signal on the corresponding data input line. All other *AND* gate outputs will be '0'. The output of the *OR* function will be the same as the output of the selected *AND* gate. In this way, the input data associated with a selected line is routed to the output signal line. This operation is called Multiplexing. The 74151 has another input called the Enable Input ($\bar{E}$). The bar on the symbol specifies that the Enable input is active low. This means the output is enabled when the input signal is zero. Otherwise, the output is set to one when the recommended pull-up resistor is used. The $\bar{E}$ input allows two cascaded 8:1 multiplexers to be combined together to form a single 16:1 multiplexer. Although multiplexers are primarily used for switching data paths they can also be used to realize general logic functions.

Function:

F(w,x,y,z) = (To be given out by the laboratory instructor.)

Pre-Laboratory Assignment:

1. Given the Function F(w, x, y, z) generate this function's truth table. Next, determine the min-terms (the combination of w, x, y, and z for which the output function F is one). These combinations determine the input locations of the multiplexer which are set to one. The combinations, for which the output is zero, form the input locations of the multiplexer which are set to zero.

2. Draw a logic schematic in your notebook using two 8:1 multiplexers. Consider the enable input as active high (Xilinx has the Enable input as active high - E). Three of the input variables in the given function F(w,x,y,z) are used as control inputs to the selection / control inputs of each of the 8:1 multiplexers and the fourth input variable is used to select between the two 8:1 multiplexers using the E input. An OR gate is used to OR the two 8:1 multiplexer outputs.

3. Write the test plan of this experiment on how it should be tested.

4. Write the Design Specification Plan for this experiment

5. Read this experiment carefully to become familiar with the experiment.

Procedure:

1. Using Xilinx's ISE and its schematic capture tool, design a 16:1 multiplexer using two 8:1 multiplexers (symbol M8_1E is located in the MUX category). Three of the input variables in the given function F(w,x,y,z) are used as control inputs to the selection / control inputs of each of the 8:1 multiplexers and the fourth input variable is used to select between the two 8:1 multiplexers using the E input (active high). An OR gate is used to OR the two 8:1 multiplexer outputs.

2. Define a four bit input BUS S and a single output O. This four bit bus should be connected to SW0, SW1, SW2, and SW3 and the O output should be linked to LED7 on the BASYS board.

3. The sixteen inputs should be hard coded as "0 = ground" and "1 = VCC" using the 'gnd' and 'vcc' from the Add Symbol Tool  next to the Add Instance tool. There should only be four bits for input S and one bit for output O for this experiment.

4. Use the count up option in the test bench waveform tool to vary the select input S from 0000 binary to 1111 binary. Download the implemented design of Steps 1-3 to the FPGA using the EXPORT program. Create a new truth table to verify that the design has implemented correctly.

5. At this point, steps 1 to 4 need to be repeated but this time using the VERILOG programming language. For the previous experiments 1 to 4, the VERILOG modules created thus far have been combinational logic designs. Another area of VERILOG programming is procedural programming, wherein 'if statement', 'for loop', and 'case statement' can be used. For this experiment, a procedural program will be written that implements a 16:1 multiplexer.

A procedural Verilog code looks like this:

```
always @(<sensitivity list>)
begin

<insert code here…>

end
```

The procedural code starts with the 'always' keyword and the sensitivity list. The sensitivity list will be replaced with events. When the event happens, the code between 'begin' and 'end' will be executed. Otherwise, this code won't be run.

Two possible choices in the sensitivity list are the 'posedge' and 'negedge' keywords. They are used to indicate the positive edge and the negative edge of a signal, usually a clock signal. In the code line below, the procedural code is triggered when a positive edge of the clock happens.

```
always @( posedge clock)
begin

<insert code here…>

end
```

Another option in the sensitivity list is a signal's name. For example, let's say our Verilog module has an input called 'S'. The procedural code below is triggered when the signal 'S' changes. That is, when 'S' changes from 0 to 1 or when 'S' changes from 1 to 0, the procedural code between 'begin' and 'end' runs.

```
always @( S)
begin

<insert code here…>

end
```

We have the choice or putting more than one signal in the sensitivity list. The code below triggers the procedural code when either 'S' or 'Q' changes. Notice here, we use the keyword 'or'. This is not an OR gate. For a regular OR gate, use the symbol '|'. Alternatively, we can trigger the procedural code when both of the signals 'S' and 'Q' change, as in the second line below.

```
always @(S or Q)     …
always @( S and Q)   …
```

Now, let's look at the combinational code (or logic code) that was in the previous lab and convert this code to procedural code. This is the example code from the previous lab:

```
// *** Combinational (or Logic) Code ***
module temp(A, O);
        input [1:0] A;              // input of 2 bits
        output O;                   // output of 1 bit
        assign O = A[0] & A[1];
endmodule
```

The equivalent procedural Verilog code is shown below. The header and the input/output declarations are the same. Notice, however, in the procedural code, the output 'O' is also declared as a register in the line 'reg O;'. It is a rule that any output that is assigned a value in a procedural code (that is, between the 'begin' and 'end' keywords) should be defined as a register. We can also define 'O' as output and as register in one line by writing "output reg O;".

The next thing is to decide what to put in the sensitivity list. In this code, when any bit in the input changes (whether A[1] or A[0]), we need a new evaluation of the output 'O'. So we put A in the sensitivity list. If either A[1] or A[0] changes, the total value of 'A' will change. We could have also used 'always@(A[1] or A[0])'.

Between 'begin' and 'end' keywords, we assign 'O' its value. Notice that, here, we don't use the 'assign' keyword since 'O' is declared as a register. The register is a container that contains data so we don't need to do a continuous assignment which is done by 'assign'.

```
// *** Procedural Code ***
module temp(A, O);
input [1:0] A;
output O;
reg O;                  // Declare the output as register!
always @(A)
        begin
                O = A[1] & A[0];
        end
endmodule
```

Verilog also has some advanced features that we can use in the procedural code such as 'nonblocking assignment' and 'blocking assignment'. Assume we have a register 'A'=5 and a register 'C'=10. The code below uses blocking assignments since it uses the '=' sign. It means, C is assigned to A (therefore C=5) and, after that, B is assigned to C (therefore B=5). This is the same way as a programming language like C or Java.

```
always@(<sensitivity list>)
begin
        C = A;
        B = C;
end
```

Another way to do assignments is nonblocking assignments where the assignments occur in parallel. In the code below, the assignments are done using the '<=' sign. It means the two assignments happen simultaneously. Starting with 'A'=5 and 'C'=10, we get: C=5 and B=10, unlike the result above.

```
always@(<sensitivity list>)
begin
        C <= A;
        B <= C;
end
```

'If conditional statements', 'for loops' and 'case statements' can be used within a procedural block. The code below shows an example of using a case statement for a two-bit input B:

```
module test(clock, A, B);
input clock;            // A clock signal
input [1:0] A;          // input of 2 bits
output B;
reg B;                  // Declare B as register since we use procedural code

always @( posedge clock)
        begin
            case (A)                    // Starting the case statement
               2'b00:   B = 0;
                2'b01:  B = 1;
                2'b10:  B = 1;
                2'b11:  B = 0;
            endcase                     // Ending the case statement
        end
```

Notice the syntax that we used to check the value of A. The first line is: 2'b00. This means the value we're checking is 2 bits and in binary. For example, if we want to compare to the 4-bit binary value of 1101, we would write: 4'b1101. We could also provide the values in decimal. The code above becomes:

```
always@(posedge clock)
begin
case (A)
   0:       B = 0;
   1:       B = 1;
   2:       B = 1;
   3:       B = 0;
endcase
end
```

Another way to write this procedural code is by using if-else statements as in the code below.

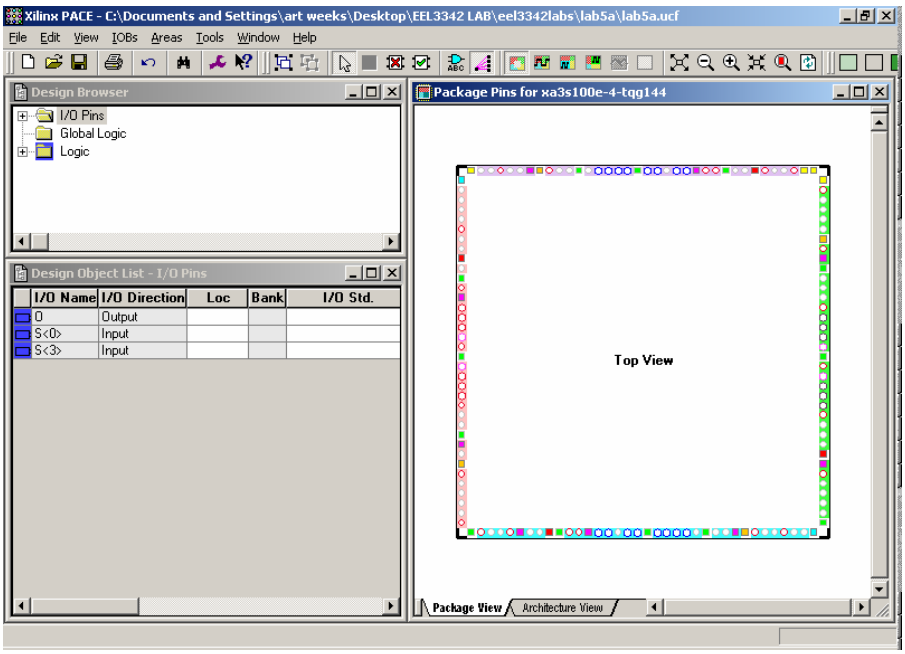
```
always@(posedge clock)
begin
        if(A==0)
                B=0;
        else if(A==1)
                B=1;
        else if(A==2)
                B=1;
        else    B=0;
end
```

After studying the above VERILOG programming examples, write a VERILOG programming module that implements a 16:1 multiplexer using procedural programming. Assign a 4-bit bus S for the select lines of the multiplexer and a signal O for the output (O needs to be reg type). For the sixteen inputs, hardcode the function F(w, x, y, z) using the **assign** operator. Simulate this VERILOG 16:1 multiplexer and compare the simulation output to the truth table developed for the experiment. Use the count up option in the test bench waveform tool to vary the select input S from 0000 binary to 1111 binary.

6. At this point, the VERILOG version of the multiplexer should be implemented assigning S[0] to S[3] to SW0 - SW3 and the output O to LED0.

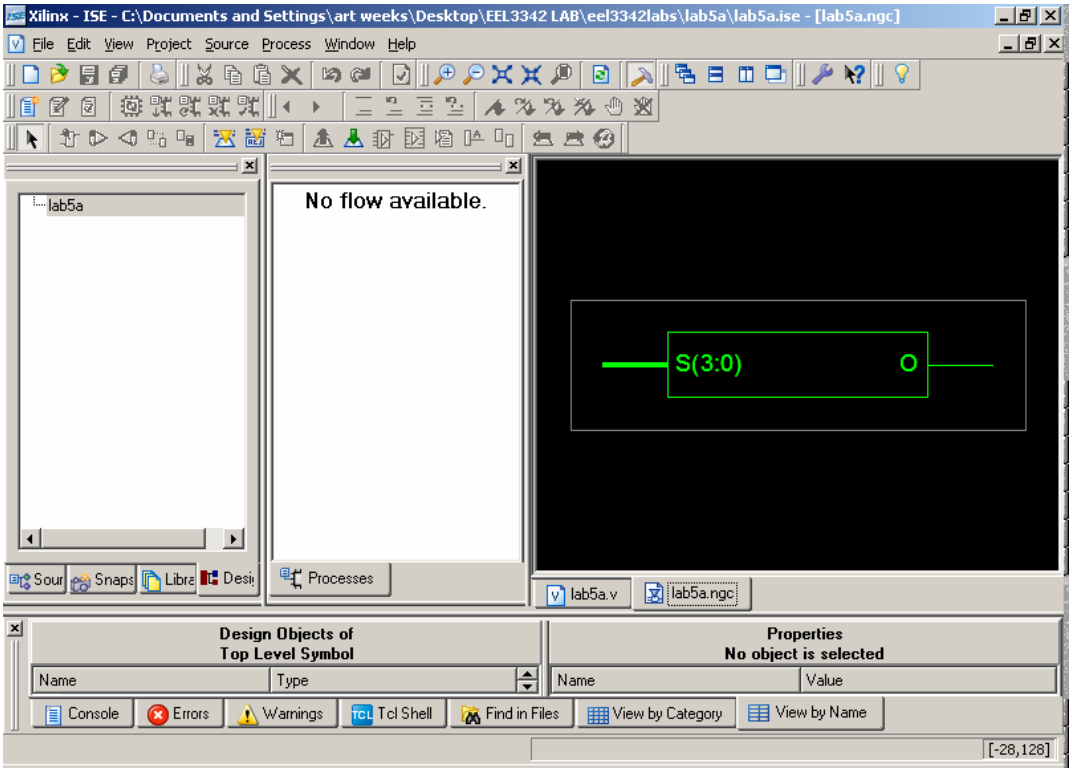
NOTE: An interesting thing may happen depending on the set of 16 inputs to multiplexer. The figure below shows the pace program for one set of 16 inputs to multiplexer.

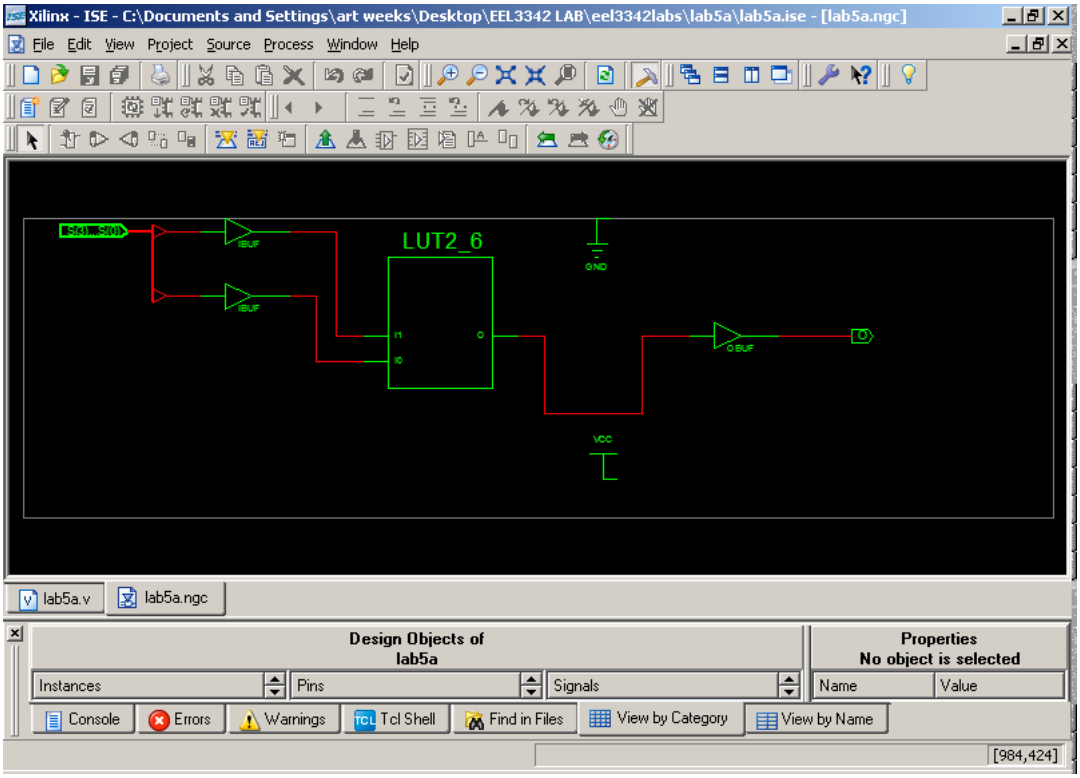
The output O appears along with the bus S bits 0 and 3. Bits 1 and 2 of bus S are missing. What happened is that the VERILOG compiler optimized the design by evaluating the 16 inputs to the multiplexer for the 4 bits of selection / control to the multiplexer and determined that S[1] and S[2] are 'do not cares terms' X. If the 16 inputs to the multiplexer change, the Select bus bits used in the implementation will change.



The real question is that the user designed a 16:1 multiplexer, what did the compiler implement since inputs S[1] and S[2] were ignored? This can be found by looking under **View Technology Schematic** under the S**ynthesize-XST** in the **Processes** window. Double click this option and the following window appears showing the top level schematic representation. Make sure the design has been synthesized and implemented first. This schematic shows the top-level diagram of what was implemented. It is a good check to make sure that all inputs and outputs are defined as expected. The schematic shows an input bus S of four bits and an output O.

Double click the schematic to obtain the next level of detail as shown in the next Figure. Both the Processes and the Sources windows were closed to make it easier to see the detailed schematic. This schematic shows what is actually implemented within the FPGA. The FPGA selected to use a 4 word by 1 bit lookup table (LUT) to implement the multiplexer and used only two bits S[3] and S[0] to address this lookup table (more on using a memory device to perform logic functions is discussed later in this manual). The output from the lookup table is then fed to a buffer and to the output O. Bits S[1] and S[2] are not used in this design. The actual bus bits used can be determined by placing the mouse cursor over the bus tap and reading which bit is used as displayed in the highlight popup window.





Using only the available bits for the select input S as defined by the PACE program assign these bits to SW0 - SW3 appropriately and assign the output O to LED0. For

example, if S[2] is missing in the input and output list in the PACE program, leave SW2 blank. Next re-build the VERILOG 16:1 multiplexer so that the synthesis and implementation is up to date. Finally, generate the programming files to be downloaded to the BASYS board.

7. Download the *.BIT file to the BASYS board and verify that the VERLOG version of the 16:1 multiplexer works as designed.

Questions:

(To be incorporated within the Conclusion section of your lab report)

1. Investigate the function of a lookup table and describe how one works.

2. Consider a 16 word by 1 bit lookup table. Give the values stored in each location 0000 binary (word zero) to 1111 binary (word fifteen) for the function F(w, x, y, z). The truth table that was generated in the pre-laboratory will help here.

3. Have you met all the requirements of this lab (Design Specification Plan)?

4. How should your design be tested (Test Plan)?

EXPERIMENT #6

Decoder and Demultiplexer

Objective:

To introduce decoders and their use in selecting one output at a time. Both the schematic capture tool and the VERILOG design language will be used to implement a 3 to 8 and a 4 to 16 decoder.

Discussion:

Decoder and Demultiplexer - The Decoder performs an opposite function to that of the multiplexer. It connects one input line to one of several output lines. A decoder is another combinational logic device that acts as a "minterm detector". A decoder can be made by using a demultiplexer with its input as '1'. Examine the spec sheet in Appendix A (starting on page 9 of Appendix A) for 74155. For an input word of n-bits, the decoder will have 2^n outputs, one for each minterm 0 to 2^{n-1}. When a bit pattern is placed on the decoder's inputs (which corresponds to a minterm), the corresponding decoder output will be '0' while the non-selected outputs will be '1'. Since the outputs are inverted (active low), a *NAND* gate is used to *"OR"* the minterms. The enable inputs allow for connecting two or more 74155's together to decode longer words. Logic functions can be implemented rather easily with a decoder. Rather than wiring logic gates to realize a sum-of-products, the desired minterms can be obtained by *OR*-ing the appropriate outputs from the decoder with a *NAND* gate if the outputs are active low or an OR gate if the decoder outputs are active high.

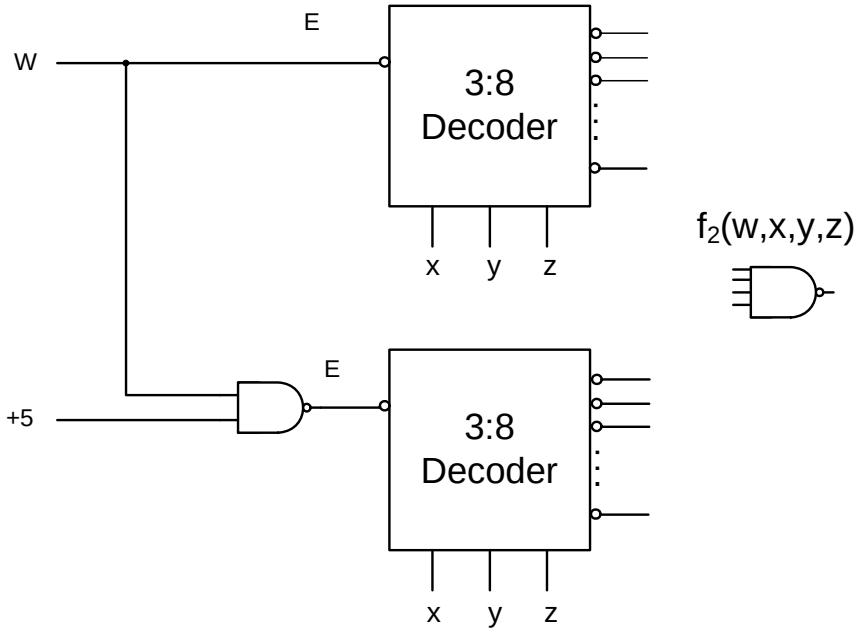


Figure 6-1: Cascaded Decoders

Functions:

- f(a, b, c) = SPECIFIED BY THE LAB INSTRUCTOR
- g(w, x, y, z) = SPECIFIED BY THE LAB INSTRUCTOR

(You may have to expand the equation to obtain minterms containing all the input variables.)

Pre-Lab Assignments:

1. Read this experiment to become familiar with this experiment.

2. Draft the Design Specification Plan.

3. Draft the Test Plan for the experiment.

4. Represent the two functions in a truth table and in the minterm list form. For the three input functions, obtain a schematic diagram using one 3-to-8 decoder (active high) and an OR gate. For the four input function, two 3-to-8 decoders will be needed. Implement the four input function using two 3-to-8 decoders and the required OR gate. Use the E line to connect the two 3-to-8 decoders together as shown in the figure above (remember now E is active high and the figure is shown for active low E). Assume for the 3-to-8 decoders that the outputs are active high and that the device is enabled when the E line is '1'. All of the outputs are zero when E = 0 independent of the inputs A0, A1, and A2. And as an example, D4 will be high when E=1, A0=0, A1=0, and A2=1.

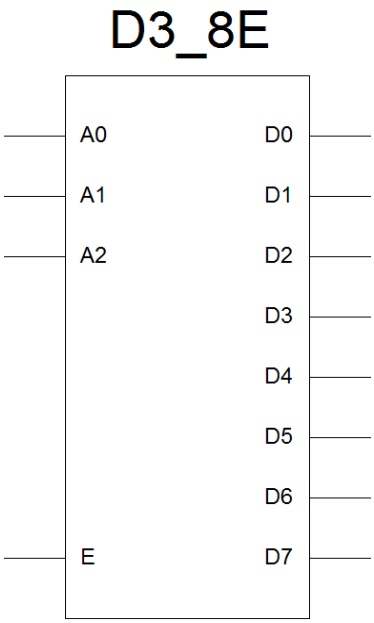


Figure 6-2: 3-to-8 Decoder Component

Procedure:

Part 1:

1) Design a circuit using a D3_8E to realize the first function f(a, b, c), using the schematic capture tool of the ISE. Draw a logic schematic in your notebook. Use an OR gate to OR the selected outputs together.

2) Simulate the design implemented in Step1 and verify that it implements correctly the function f(a, b, c).

3) Download the design in Step 1 to the BASYS board with A0 = SW0, A1 = SW1, A2 = SW2 and the output = LED0 using the EXPORT program. Generate a new truth table to verify that the design works correctly.

4) Design a circuit using two D3_8E 3-to-8 decoders to realize one 4-to-16 decoder to implement the function, g(w, x, y, z) above. The enable inputs can be used to include the fourth input bit (w) as shown in the Figure above. The three bits x, y, and z are common to both 3-to-8 decoders. Use an OR gate to OR the appropriate outputs together. Draw a logic schematic in your notebook of your design.

5) Simulate the design implemented in Step 4 and verify that it correctly implements the function g(w, x, y, z).

6) Download the design in Step 1 to the BASYS board with A0 = SW0, A1 = SW1, A2 = SW2, A3 = SW4 and the output = LED0 using the EXPORT program. Generate a new truth table to verify that the design works correctly.

Part 2:

1. In this part, you will write the VERILOG procedural code for a 2-to-4 decoder. Then, in a later step, you will use the 2-to-4 decoder that you created to make a 3-to-8 decoder. One way to implement the decoder is to use if-else statements, which have the following format:

```
if(expression)
   begin
      program code
   end
else if(expression)
    begin
      program code
    end
else
    begin
      program code
    end
```

The conditional operators that can be used with an **if** statement, are the same as C programming language:

<	less than
<=	less than or equals
>	greater than
>=	greater than or equals
==	equal to
!=	not equals

2. For the VERILOG program, create a new module by clicking on 'Project → New Source'. Select 'Verilog Module' as the file type. Call the module 'mydecoder24vlog'. The next window will prompt you for the input and output signals of the module. It should have a 1-bit input called 'en' (the enable signal), a 2-bit input 'in' (the input) and a 4-bit output called 'out'.

3. Xilinx will create a template for your code. You need to fill the procedural code that implements a decoder.

```
module mydecoder24vlog (en, in, out);
      input en;              // Enable signal
      input [1:0] in;
      output [3:0] out;
      ????                   // Declare 'out' as a register

      always@(????)          // Fill the sensitivity list
      begin
            if(en==1)
            case(in)
            2'b00:     out = 4'b0001;
            2'b01:     ???         // Fill this line
            ???                    // Fill this line
            ???                    // Fill this line
            endcase

            else if(en==0)
            begin
            out = ???              // What should 'out' be?
            end
      end
endmodule
```

First, add a line to declare the output 'out' as a register. We need to do this since we'll assign a value to 'out' in the procedural code (between 'begin' and 'end' keywords). Remember, the register of 'out' should be the same size as the signal 'out' (that is, 4 bits).

Then, fill the sensitivity list. You should ask the question: when should this code refresh, that is, run again? Typically, a code should refresh if any one of its inputs changes.

The first if-statement checks if the enable signal is 1. If it is, the decoder will see the value of the input to determine which output to assert. Only one output will be '1' in this case and the other outputs are '0'. In the first line of the case statement, if the input is 00, the output is 0001. That is, the line out[0] is '1' and the other output lines are '0'. Fill the remaining lines in the case statement.

Finally, if the enable is '0', all the outputs should be '0'. This assignment can be done in one line. Fill it in the code.

To synthesize this code, right click on it ('mydecoder24vlog') in the left side window of Xilinx and select 'Set as Top Module'. This means when you synthesize, this module will be synthesized and not the one containing the schematic. Since this module doesn't reference (or call) the schematic module, the two modules are independent. You can use 'Set as Top Module' to select which of the two codes you want to synthesize or simulate.

4. Now you will write a VERILOG module of a 3-to-8 decoder which will use two instances of the 2-to-4 decoder 'mydecoder24vlog'. Start by creating a new module in the project by clicking on "Project → New Source". Select 'Verilog Module' as the file type and name the file 'mydecoder38vlog'. In the next window, specify an input 'in' of 3 bits and an output 'out' of 8 bits. We won't use an enable signal for this decoder.

   The figure below shows you the 3-to-8 decoder that we will build in VERILOG. We will use a programming mode in VERILOG called 'structural'. In the structural mode, we describe the structure of the circuit, which we have in the figure below. By looking at the figure, we can see that we need two 2-to-4 decoders (we'll use the procedural model that we created 'mydecoder24vlog') and one inverter. VERILOG has the inverter ready made for us, so we can use it.

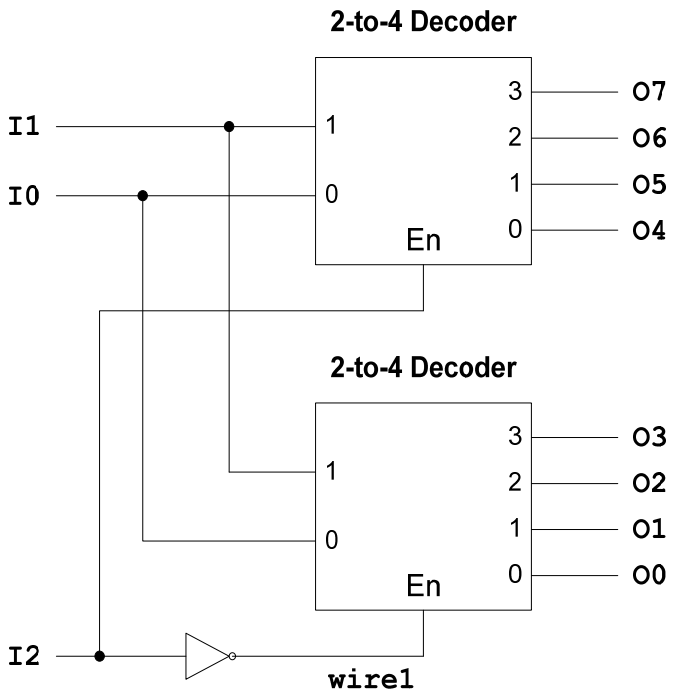


Figure 6-3: A 3-to-8 Decoder Built with Two 2-to-4 Decoders

Now, let's look at the code template that Xilinx created for us for the 3-to-8 decoder in the file 'mydecoder38vlog'. In the code below, we added a new variable of type 'wire' that we called 'wire1'. Why do we need the wire? If we look at the figure above, we can see that every signal is either an input (I0, I1, I2) or an output (O0 through O7). However, the line going out of the inverter and into the enable of the lower decoder is neither an input nor an output to the 3-to-8 decoder. It is an internal wire. So we can use the 'wire' type in VERILOG to take care of this connection.

```
module mydecoder38vlog (in, out);
   input [2:0] in;            // 3-bit input
   output [7:0] out;          // 8-bit output
   wire wire1;                // Declaring a wire

   //module mydecoder24vlog (en, in, out);
   mydecoder24vlog decoder1 (in[2], in[1:0], out[7:4]);
   mydecoder24blog decoder0 (???, ???, ???);
   not inverter0 (wire1, in[2]);

endmodule
```

Now, we'll declare two instances of the module 'mydecoder24vlog' to make the 3-to-8 decoder. Since the 2-to-4 decoder module is in the same folder, Xilinx will find it. The first line declares a 2-to-4 decoder and calls is 'decoder1'. You can call this name anything you want. It's like in C language when we write 'int x;'. The term 'int' is the

variable type and the term ‘x’ is the variable name that we can call anything we want. Here, the type is ‘mydecoder24vlog’, the module that we wrote earlier. Also, notice that we put the header of ‘mydecoder24vlog’ commented out. We need the header because when we make an instance of the 2-to-4 decoder, we need to provide the input and output in the same order as in that module. Therefore, we should provide the enable signal first, the input (a bus of 2 bits) second and finally, the output, a bus of 4 bits. By looking at the figure, we can see that the enable signal of the upper decoder is ‘I2’, therefore, we replace it with ‘in[2]’. We have also filled the input and output signals of the upper decoder. Fill the instance line of the second decoder. For the inverter, we should provide the output first, ‘wire1’, and then the input second, ‘in[2]’, as in the figure.

5. Implement the 3-to-8 decoder using the sample code above, called from a top-level module called lab6.

6. Simulate the design in step 5 verifying that the 3-to-8 decoder is working correctly.

7. Download the 3-to-8 decoder of step 5 to the BASYS board using the EXPORT program. Set ‘en’ = SW7, ‘in0’ = SW0, ‘in1’ = SW1, ‘in2’ = SW2, ‘in3’ = SW3 and ‘out0’ to ‘out7 = LED0 to LED7.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. If the decoder has active low outputs (74155) instead of active high outputs as the case for D3_8E 3-to-8 decoder, why can a *NAND* gate be used to logically OR its outputs?

2. Which MSI function, multiplexer or decoder would best implement multiple output functions (i.e. many functions of the same input variables)? Why?

3. What are the advantages of using an FPGA over MSI devices or SSI devices?

4. In this experiment, the enable line is used to obtain a 4-to-16 decoder. What needs to be added to obtain a 5-to-32 decoder using the D3_8E 3-to-8 decoders? Draw a schematic of a 5-to-32 decoder (using four D3_8E 3 to 8 decoders).

5. Write the Test Plan of how this experiment should be tested.

6. Write the Product Specification Plan to verify that the all the requirements have been meet.

EXPERIMENT #7

Random Access Memory

Objective:

To examine the use of RAM (and ROM) as means of realizing combinational logic circuits.

Discussion:

Read/Write Memory (RAM): RAM is a memory device which can be used as another means of implementing combinational logic. A memory device is a MSI (Medium Scale Integration), LSI (Large Scale Integration) or VLSI (Very Large Scale Integration) circuit, depending on the memory size circuit. The RAM chip contains an array of semiconductor devices, which are interconnected to store an array of binary data. Data words are stored in different locations on a RAM, each location being unique. These locations are accessed by addresses. The word length and the number of addresses depend on the size of a particular RAM. Data words can be written into or read from the RAM by accessing it with the desired or respective address. RAM is a volatile memory device. It must be "powered" on whenever data is to be written or read from it, and it loses all the stored information when the power is shut off. One of the advantageous features of this device is that it allows for the change in the stored data during the operation. A typical FPGA can implement memory devices in several different word lengths and word widths. For this experiment, the memory device of interest will be the 32 word by 4 bit (ram32x4s) static random access memory. Memory expansion or "cascading" is achieved through multiple chip select inputs.

Read Only Memory (ROM): A Read-Only Memory (ROM) is an LSI or VLSI circuit, which consists of coupling devices (most often diodes). Binary data is stored in the ROM by coupling an address minterm line (word line) to an output line (bit line). It can be read out whenever desired, but the data which is stored cannot be changed under normal operating conditions. A ROM with n input lines and m output lines contains an array of 2^n words and each word is m bits long. The input lines serve as an address to select one of the 2^n words. When an input combination is applied to the ROM, the pattern of 0's and l's which is stored in the corresponding word in the memory appears at the output lines. The basic structure of the ROM consists of a decoder and a memory array. The n inputs are directed to the n inputs of the decoder and when a pattern of n 0's and l's is applied to the input, exactly one of the decoder outputs is 1. The decoder output pattern stored in this word is transferred to the memory output lines. Unlike the RAM, the ROM is a non-volatile memory device and what is stored at each location can not be changed.

Pre-Laboratory Assignment:

1. Read this experiment carefully to become familiar with the experiment.

2. Read section 9.6 of the Roth text for further discussion on ROM.

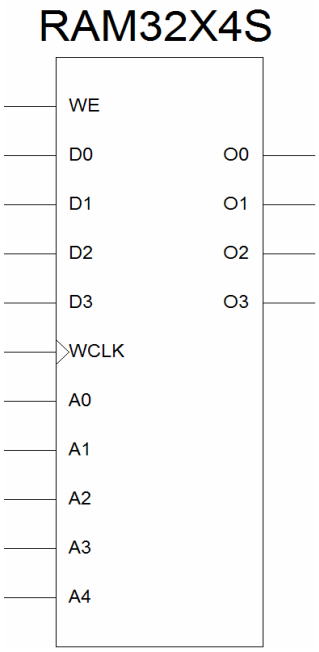
3. Given the Function F1(w, x, y, z) and F2(x1, x0, y1, y0), write the truth table for each function.

   F1(w, x, y, z) - Specified by the lab instructor
   F2(x1, x0, y1, y0) is a two bit adder. The function F2(x1, x0, y1, y0) has 3 outputs - 2 bits for the sum and 1 bit for the carry out C_{out}.

4. The variables w, x, y and z or x1, x0, y1 and y0 are inputs that are tied to the address lines of the RAM or ROM. Prepare a truth table with inputs A3, A2, A1 and A0 from 0000 to 1111 (16 combinations representing w, x, y and z or x1, x0, y1 and y0) and outputs O0, O1, O2 and O3. O0 represents the output of the function F1(w, x, y, z), O1 and O2 represents the two bits for the sum and O3 represents the carry out C_{out} bit. The truth table should have the following columns:

A3	A2	A1	A0	O0	O1	O2	O3
0	0	0	0				
0	0	0	1				
1	1	1	1				

   Each of the input word 0000 to 1111 represents one memory location of the RAM. The output of each word represents the data to be written in the corresponding memory location. Since in this experiment you have 16 input words, you need 4 address lines A3 – A0. The line A4 can be grounded.



5. Draw a logic schematic in your notebook for this experiment using the ram32x4s memory device showing all the inputs and outputs.

   The address lines are A0 through A4. The input data lines to program the RAM device are D0 through D3 and the output data lines are O0 through O3. To write data to this RAM, WE (write enable) must be '1' and the WCLK (write clock) line must toggle from 0 to a 1 (the rising edge of the WCLK).

6. Write the test plan of this experiment on how it should be tested.

7. Write the Design Specification Plan for this experiment

Procedure:

1. Using Xilinx's ISE and its schematic capture tool, implement the 32 word by 4 bit design given in the pre-laboratory section and the schematic generated in Step 5 of the pre-laboratory assignment. The RAM32x4s is present in **Symbols - Categories – Memory**. Click on the **Symbol Info** and study the functioning of this device. In the schematic, use bus for inputs D0-D3 and A0-A3. The input A4 can be connected to ground. Add I/O markers for the two buses, the WE and WCLK lines and the outputs O0 –O3.

2. Synthesize and implement this device. The data will be written into the RAM on the FPGA board.

3. In the Create Area Constraints, attach D0 - D3 to SW4 - SW7, attach A0 - A3 to SW0 - SW3, attach WE to push button BT0 and WCLK to push button BT1. For outputs, attach O0-O3 to LED0 - LED3. Refer Appendix D for the pin details. During the pin

assignments using Xilinx's PACE tool, the WCLK cannot be auto placed by dragging this I/O marker to a pin on the FPGA. The pin is reserved for a global clock input and can only be assigned to global clock pins located at the top and bottom of the FPGA pin package. We need to override this by simply typing the pin number of push button one (P48) in the Loc column next to WCLK pin name. Do not forget the P in the pin name. Save, close and re-implement the design in the sources window.

4. Generate programming file. Download the implemented design to the FPGA using the EXPORT program.

5. To write to this memory device, the address 0000 to 1111 will be used. The inputs A3 - A0 of the truth table in Step 4 of the pre-lab form the input to the address lines. The data to be written (O3-O0 of the truth table) is selected on D0 - D3. The WE is selected high first and while pressing WE (state '1') press the WCLK line (state '0' to '1' when the push button is pressed and back to '0' when the push button is released). On the rising edge of WCLK, the data on the data lines (SW4 – SW7) will be written in the address specified by the address lines (SW0 – SW3).

6. Write into all the 16 address locations 0000 to 1111 of the RAM by each time selecting a particular address on SW0-SW3 and the corresponding data on SW4-SW7 and enabling WE and creating a '0' to '1' pulse on WCLK.

7. When the push button for WE is not pressed then the state is WE = 0 i.e. the RAM is in the read mode. Now feed in the 16 address locations in the address line (SW0 – SW3) and verify that the LED outputs match with the truth table of Step 4 of the pre-lab.

8. Verify that the function F1(w, x, y, z) is written into the RAM and verify that it implements this function correctly. Generate a truth table of the implemented design.

9. Verify that the function F2(x1, x0, y1, y0) is written into the RAM and verify that it implements this function correctly. Do not forget that this function has 3 outputs. Verify that function F2 performs a two-bit addition correctly. Generate a truth table of the implemented design.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. Discuss the advantages of using RAM for implementing combinational functions.

2. Which one is more practical, read/write memory or read only memory? Explain.

3. Some memory devices have only one set of data lines for both input and output. What is the advantage and disadvantage of having the input and output lines separately or shared?

4. Have you met all the requirements of this lab (Design Specification Plan)?

5. How should your design be tested (Test Plan)?

6. What is the advantage of using RAM or ROM memory to implement combinational logic?

7. How can two 32x4s devices be combined together to form one 64 by 4 bit memory system? (Hint: 8 - 2 to 1 multiplexers can be used in the design).

EXPERIMENT #8

Flip-Flop Fundamentals

Objective:

To build and investigate the operation of an asynchronous, a clocked, and a master slave SR flip-flop.

Discussion:

Asynchronous SR Flip-flop:

An asynchronous (un-clocked) SR flip-flop is easily represented by two cross-coupled NAND gates preceded by two inverters as shown in Figure 8-1. When inputs S and R are both '0', the outputs are stable; when S = l and R = 0, Q is set to '1' ($\bar{Q}$ is set to '0'); and when S = 0 and R = l, Q is reset to '0' ($\bar{Q}$ is set to '1'). The SR flip-flop operation is undefined when both inputs S and R are set to '1'. This condition is analogous to the flip-flop being set and reset at the same time thus causing an ambiguous output condition where Q = $\bar{Q}$ = '1'. The state transition table for this SR flip-flop is shown in Figure 8.3. Q^v is the present state and Q^{v+1} is the new state for the output Q.

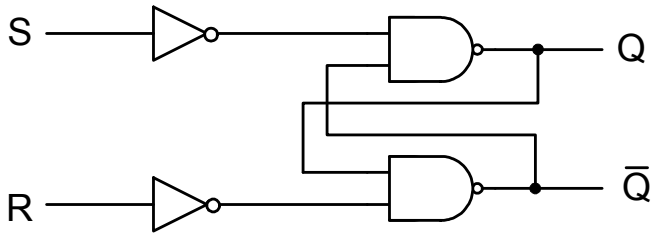


Figure 8-1: Asynchronous SR Flip-Flop

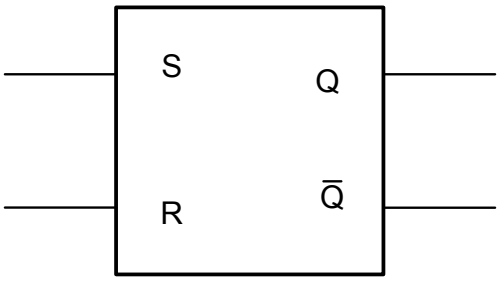


Figure 8-2: Asynchronous SR Flip-Flop Block Diagram

R	S	Q^v	Q^{v+1}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1 (Not Allowed)	1 (Not Allowed)	0	1
1 (Not Allowed)	1 (Not Allowed)	1	1

Figure 8-3: Asynchronous SR Flip-Flop State Transition Table

Clocked SR Flip-flop:
A clocked SR flip-flop is created by ANDing (or windowing) the S and R inputs with a clock. The SR flip-flop as shown in Figure 8-1 has inverters on the inputs. The AND gate and the inverter can be replaced with a NAND gate as shown in Figure 8-4. Figure 8-5 gives its schematic representation.

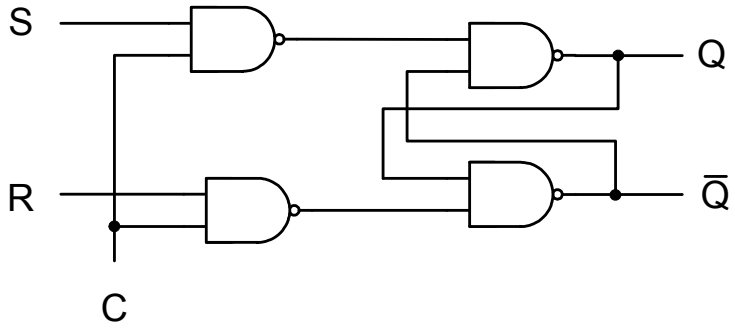


Figure 8-4: Clocked SR Flip-Flop

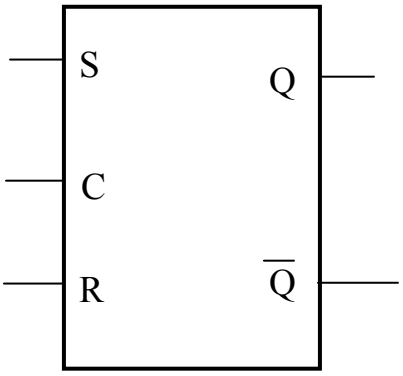


Figure 8-5: Clocked SR Flip-Flop Block Diagram

The clock is represented by the line C. When the clock is high or '1', then the device operates similar to an asynchronous SR flip-flop. However, when the clock is low or '0' then the outputs are stable and not permitted to change or "latched". The clock acts as a window. When it is high, the flip-flop becomes sensitive to the inputs but when the clock falls low, the output state can no longer change and is therefore determined by the last set or reset condition on the S and R inputs. Thus the device is no longer sensitive to changes on the inputs beginning at the falling edge of the clock. Only when the clock goes back to '1' will the device become sensitive to the inputs again. This clocked SR flip-flop is also sometimes referred to as a positive level SR flip-flop.

D Flip-flop:

A D Flip-flop or LATCH can be created by adding an inverter between the S and R inputs of the clocked SR flip-flop, as shown in Figure 8.6. Its schematic input is given in Figure 8.7. The input is now referred as the Data or D input. When the clock is one, whatever appears on the D input appears on Q and when the clock goes from a '1' to a '0', the last value on the D input is the value held on the Q output.

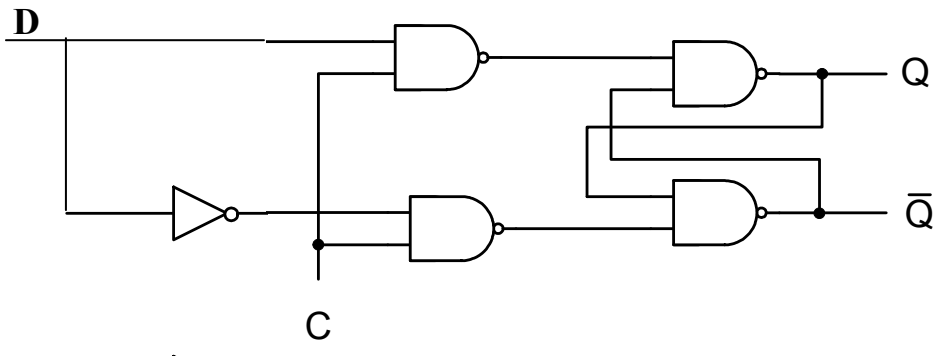


Figure 8-6: Clocked D Flip-Flop

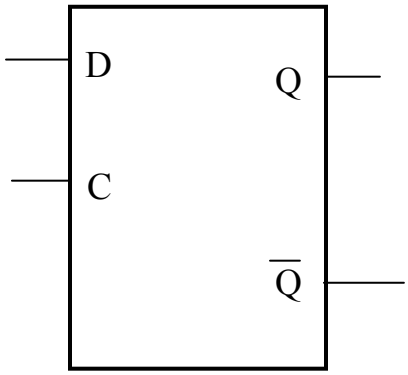


Figure 8-7: Clocked D Flip-Flop Block Diagram

The state transition table for the D flip flop is given in Figure 8-9. This table only has 4 rows, one for D and one for the present state Q^v. The new output Q^{v+1} is equal to the D data input.

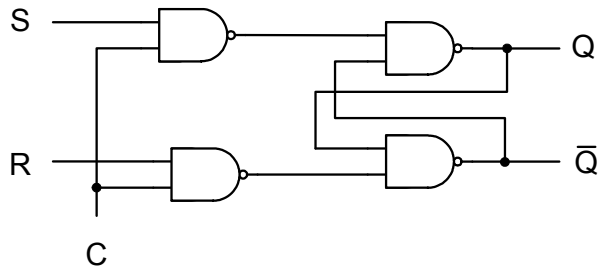


Figure 8-8: Clocked D Flip-Flop Block Diagram

D (Data)	Q^v	Q^{v+1}
0	0	0
0	1	0
1	0	1
1	1	1

Figure 8-9: Clocked D Flip-Flop State Transition Table

There is an important difference in the testing of the circuits with memory. The test sequence is no longer an application of all of the possible inputs. The sequence must consider the state or value of the memory of the circuit. For the SR flip-flop, the S and R inputs must first put the flip-flop in a known state (e.g., SR = 01 resets Q to 0). The next inputs of SR=00 are used to check the condition SRQ+ = 000. Since Q will remain 0, the next inputs of SR = 01 will test the condition SRQ+ = 010. In this way, a test sequence is developed and used to program the test bench of Xilinx's ISE for testing (i.e. the sequence SR = 01, 00, 01… developed above is the start of the test sequence). For the SR flip-flop, there are two inputs S and R. The state transition table of Figure 8-3 has eight rows to count for the present state Q^v. The inputs that form the transition table are S, R, and Q^v.

Pre-Laboratory Assignments:

1. Draw the schematic and block diagrams of the asynchronous SR, clocked SR flip-flops and clocked D flip-flops. Include all switches and LEDs required.

2. Draw a timing diagram for each flip-flop undergoing a setting, resetting and latching (hold) condition.
3. Develop a test sequence for each of the flip-flops. This will be a part of the required test plan.
4. A draft of the Test Plan.
5. A draft of the Design Specification Plan.
6. Have printouts of the circuit and the sketched timing diagram.

Procedure:

1. Design and simulate the asynchronous SR flip-flop circuit using Xilinx's ISE using the schematic capture tool and the simulation tool. Generate printouts of the schematic circuit, timing diagram and test bench inputs. During the simulation make sure that all rows of the transition table are simulated. Set the test bench end time to 4000 nano-seconds and the simulation end time to 4000 nano-seconds to have enough time cycle through all possible inputs and present values for Q^v. See Experiment #2 on how to set the test bench end time and the simulation end time.

2. Now implement the design in Step 1 of this procedure and configure the FPGA so that R is on SW0 and S is on SW1. Also use LED0 for the output Q and LED1 for $\bar{Q}$.

3. Download the FPGA configuration file to the BASYS board using the EXPORT program. Verify that the SR flip-flop works correctly.

4. Design and simulate the clocked SR flip-flop circuit using Xilinx's ISE using the schematic capture tool and the simulation tool. Generate printouts of the schematic circuit, timing diagram and test bench inputs. During the simulation make sure that all rows of the transition table are simulated. Set the test bench end time to 4000 nano-seconds and the simulation end time also to 4000 nano-seconds to have enough time cycle through all-possible inputs and present values for Q^v. See Experiment #2 on how to set the test bench end time and the simulation end time.

5. Now implement the design in Step 4 of this procedure and configure the FPGA so that the clock input is on SW7, R is on SW0, and S is on SW1. Also use LED0 for the output Q and LED1 for $\bar{Q}$.

6. Download the FPGA configuration file for Steps 4 and 5 to the BASYS board using the EXPORT program. Verify that the clocked SR flip-flop works correctly.

7. Design and simulate the clocked D flip-flop circuit using Xilinx's ISE using the schematic capture tool and the simulation tool. Generate printouts of the schematic circuit, timing diagram and test bench inputs. During the simulation make sure that all rows of the transition table are simulated. Set the test bench end time to 4000 nano-seconds and the simulation end time also to 4000 nano-seconds to have enough time cycle through all-possible inputs and present values for Q^v. See Experiment #2 on how to set the test bench end time and the simulation end time.

8. Now implement the design in Step 7 of this procedure and configure the FPGA so that the clock input is on SW7, and D is on SW0. Also use LED0 for the output Q and LED1 for $\bar{Q}$.

9. Download the FPGA configuration file for Steps 7 and 8 to the BASYS board using the EXPORT program. Verify that the clocked D flip-flop works correctly by verifying the transition table given in Figure 8.9.

10. Repeat Step 7 for the clocked D flip-flop but this time design and simulate this circuit using the procedural programming approach in VERILOG with Q defined as a register variable. Do not forget to use the always @(expression) operator. Make sure that the expression used covers all conditions of D and of the clock input. Generate printouts of the VERILOG file, timing diagram and test bench inputs. Be sure to simulate all possible conditions for D input and the present value of Q^v . Set the test bench end time to 4000 nano-seconds and the simulation end time also to 4000 nano-seconds.

11. Now implement the design in Step 10 of this procedure and configure the FPGA so that the clock input is on SW7, and D and is SW0. Also use LED0 for the output Q and LED1 for $\bar{Q}$.

12. Download the FPGA configuration file for Steps 10 and 11 to the BASYS board using the EXPORT program. Verify that the clocked D flip-flop works correctly.

13. Finalize the Test Plan of this experiment on how it should be tested.

14. Finalize the Design Specification Plan for this experiment

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. What are the advantages and disadvantages of each flip-flop?

2. Draw an asynchronous SR flip-flop schematic using NOR gates.

3. Using the lectures textbook, look up level flip-flops versus edge flips-flops and discuss the difference.

4. What are some applications that might require the use of flip-flops?

5. Another type of edge flip-flop is the Toggle (T) flip-flop. Discuss how this flip-flop works and give its state transition table.

6. How can the VERILOG program that implemented the level D flip-flop in Steps 7 and 8 be converted to a positive edge or a negative edge D flip-flop?

EXPERIMENT #9

Designing with D-Flip flops: Shift Register and Sequence Counter

Objective:

To introduce the student to the design of sequential circuits using D flip-flops. A 4-bit parallel load register used as a right shift register and as a left shift register will be implemented. Additionally, a set of D flip-flops will be used to design and build a sequence counter.

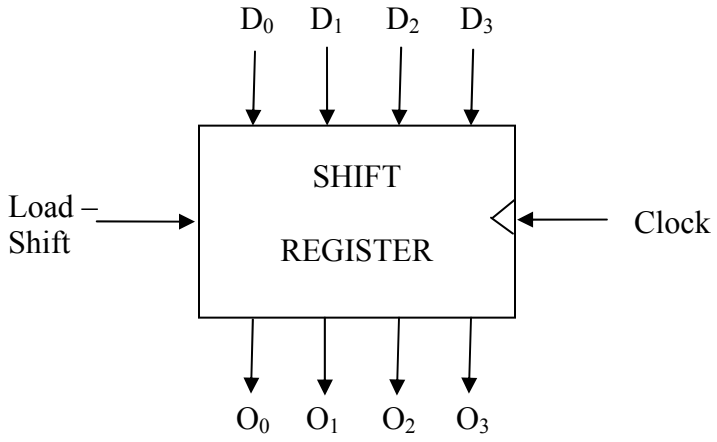
Discussion:

Part 1: Shift Register

Given 4 D flip-flops with inputs D_0, D_1, D_2, D_3, and outputs 0_0, 0_1, 0_2, 0_3, a right shift produces an output of $D_0 \rightarrow 0_0$, $0_0 \rightarrow 0_1$, $0_1 \rightarrow 0_2$, and $0_2 \rightarrow 0_3$. In this case, the original content of 0_3 is lost.

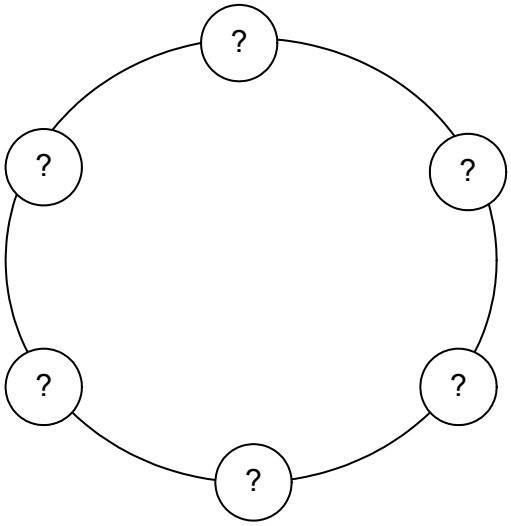
A shift left operation produces an output $0_0 \leftarrow 0_1$, $0_1 \leftarrow 0_2$, $0_2 \leftarrow 0_3$, and $0_3 \leftarrow D_3$. In the shift left case, the original content of 0_0 is lost.

A shift register can also have a parallel load feature that allows all of the D flip-flops to be loaded in one clock edge to produce $D_0 \rightarrow O_0$, $D_1 \rightarrow 0_1$, $D_2 \rightarrow 0_2$, and $D_3 \rightarrow 0_3$.



Part 2: General Sequence Counter

A sequence counter, which counts in a fixed loop, is a sequential machine that transitions from one state to the next at every clock edge. This counter has no inputs and is designed, using memory elements (typically D flip-flops), to follow a specified sequence.



Pre-Laboratory Assignments:

Part 1: Shift Register

1. Read the description of the 7495 Shift Register found in Appendix A (A-5, A-6) and analyze the logic diagram. This will help you design and understand how a parallel load shift register works.

2. Design a four bit right and left shift register that implements the following truth table. Use AND gates, OR gates and 4 positive edge clocked D flip-flops.to implement the design.

3. Draw the schematic for the design in Step 2 with D_0 through D_3, set to SW0 – SW3 and O_0 through O_3 set to LED0 to LED3. The LOAD input should be connected to SW7, L/R should be on SW6, and the CLK input should be connected to an input push button BTN0.

CLK	LOAD	L/R Shift	0_0	0_1	0_2	0_3
↑	1	X	D_0	D_1	D_2	D_3
↑	0	0	D_0	0_0	0_1	0_2
↑	0	1	0_1	0_2	0_3	D_3

4. A draft of the test plan.

5. A draft of the design specification plan

Part 2: General Sequence Counter

1. Obtain from your lab instructor the state sequence.

2. Referring to the lecture text, read the section on sequential design using D flip-flops. Understand the steps to convert a state diagram to a state transition table.

3. Design the sequence counter assigned by the lab instructor, using D flip-flops.

4. Include in your design the state diagram, state transition table, legend, next state equations, and schematic diagram.

5. A draft of the test plan.

6. A draft of the design specification plan

Procedure:

Part 1: Shift Register

1. Design the four bit left and right shift register in the schematic tool of the ISE using four D flip-flops (Symbols – Categories (Flip Flop) – Symbols (fd)) one for each bit of storage. Draw a logic schematic in your notebook of the final design. The clock input should be connected to a push button so that every time the push button is pressed, the shift register sees a clock pulse to either load data into it or shift its data left or right. Switches are notorious of producing many edge transitions when they go from 'on' to 'off' or 'off' to 'on.' These extra edges result in extra clock edges to the shift register clock input when a single clock edge is expected. Since the D flip-flops are edge triggered in the shift register, these extra edges result in the shifting of its bits by more than one bit at a time.

2. Consider the switch de-bounce circuit given below. The CLK input should be connected to push button 0 (BTN0) and the CLK25MHz should be connected to pin 54 of the FPGA, which is a 25 MHz clock input. The circuit works as follows: When CLK input is '0', the 4 and 16 bit counters are cleared and held at a count of zero. Also the terminal count TC output on the 4 bit counter is zero during a clear operation. When CLK input is '1', the clear line is set to zero and the count enable CE inputs to the counters are set to one allowing the counters to start counting up. When the count of 2^{20} -1 (20 bits = 16 bits + 4 bits) is reached, the terminal count TC output is set to '1' on the four bit counter. It takes a total time of (2^{20} -1) /25,000,000 = 0.04794 seconds for the TC output to go high. By this time the switch contact noise has diminished.

   When the TC on the four bit counter goes high, it sets one of the inputs to the AND gate to zero and thus prevents the counter from counting any further until the CLK input goes low. Hence, this produces a single rising edge on the TC output from the four bit counter independent of the number of edges on the input clock CLK. If the CLK line is noisy producing many edges then the counter is reset and is delayed until the TC on the four bit counter is set to one. In summary, this circuit solves the problem of many edges on the CLK input due to switch contact noise (switch bounce) and the output from this circuit produces a clean rising edge on the TC output of the four bit counter. This rising edge can be used to clock the shift register without any switch noise.

3. Simulate the circuit designed in Step 1 and verify that it correctly implements the function of a left / right shift register. Ignore the de-bounce circuit discussed in step 2 above for this simulation.

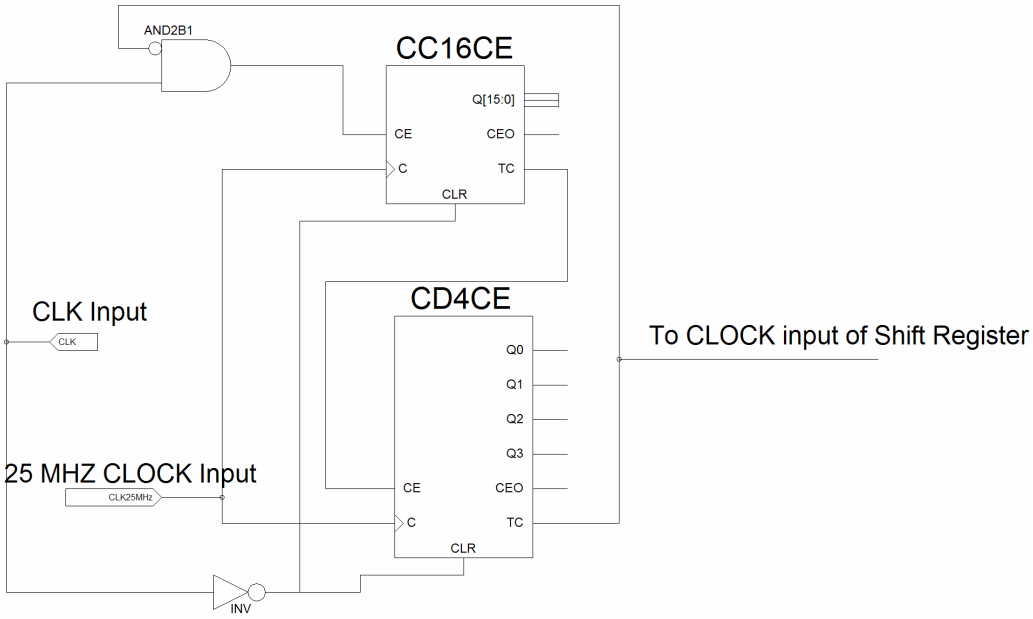


Figure: Switch De-bounce Circuit.

4. Download this circuit implemented in Step 3 to the BASYS board with D_0 through D_3 set to SW0 – SW3 and O_0 through O_3 set to LED0 – LED3. The LOAD input should be connected to SW7, L/R should be on SW6, and the CLK input should be connected to push button 0 (BTN0). You will need to overwrite the pin for the CLK signal. Please ignore the warning in the PACER program due to over riding the global clock pin to push button 0. Ignore the de-bounce circuit for this step.

5. Use the EXPORT program to download the program. Run several cases showing that the shift register works correctly. Describe what is observed during these tests in your laboratory report.

6. Modify the shift register circuit of steps 1, 3, 4, and 5 above to include the de-bounce circuit of step 2. Connect the push button 0 to the CLK input which is one of the inputs of the AND gate shown in figure above. The output of this AND gate is connected to the count enable CE input of the 16 bit counter. Make sure that the 25MHz Clock input is connected to pin P54, which forms the clock inputs for the 4 and 16 bit counters. Finally, tie the terminal count TC output of the four bit counter to the clock inputs of the four D flip-flops of the shift register.

7. Download the design in Step 6 to the BASYS board with D_0 through D_3 set to SW0 – SW3 and O_0 through O_3 set to LED0 - LED3. The LOAD input should be

8. Use the EXPORT program to download the program. Run several cases showing that the shift register works correctly. Describe what is observed during these tests in your laboratory report. Compare the results obtained in this step to that of step 5. Include this comparison in your laboratory report for this experiment.

Part 2: Sequence Counter

1. Design the sequence counter in the schematic tool of the ISE using the require number of D flip-flops (fd) under Flip Flop category (one for each bit of storage). Draw a logic schematic in your notebook of the final design.

2. Simulate the design implemented in Step 1 and verify that is correctly implements the sequence counter. Ignore the de-bounce circuit in step 2 above when performing the simulation.

3. Download the design in Step 1 to the BASYS board with O_0 through O_N set to LED0 to LEDN (N is the number of bits required to implement the sequence counter). The CLK input should be connected to push button 0. Use the EXPORT program to download this circuit to the FPGA. Generate a new state transition table to verify that the design works correctly. Ignore the de-bounce circuit in step 2. Describe what was observed during this test in you laboratory report.

4. Modify the sequence counter circuit of steps 1, 2, and 3 above to include the de-bounce circuit of step 2 of Part 1 of this experiment. Connect the push button 0 to the CLK input which is one of the inputs of the AND gate shown in figure above. The output of this AND gate is connected to the count enable CE input of the 16 bit counter. Make sure that the 25MHz Clock input is connected to pin P54, which forms the clock inputs for the 4 and 16 bit counters. Finally, tie the terminal count TC output of the four bit counter to the clock inputs of the four D flip-flops of the sequence counter.

5. Download the design in Step 4 to the BASYS board with O_0 through O_N set to LED0 to LEDN. The CLK input should be connected to push button 0. Use the EXPORT program to download the program. Generate a new state transition table to verify that the design works correctly. Include the de-bounce circuit in step 2 of Part 1. Describe what is observed during these tests in your laboratory report. Compare the results obtained in this step to that of step 3. Include this comparison in the laboratory report.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. What happens if there is noise on the clock input for both part 1 and part 2?

2. How can the shift register of part 1 be expanded to 8 bits? Draw a schematic of this shift register.

3. What happens to the shift register of part 1 if during the left shift operation O_3 is tied to D_0?

4. What happens to the shift register of part 1 if during the right shift operation O_0 is tied to D_3?

5. What problems may arise due to the placement of “don't cares” in the next states of states that are not desired?

6. How would one avoid the problems that the “don't cares” might produce?

7. Discuss the issues of using switches as clock inputs to edge triggered devices.

EXPERIMENT #10

Sequential Circuit Design: Counter with Inputs

Objective:

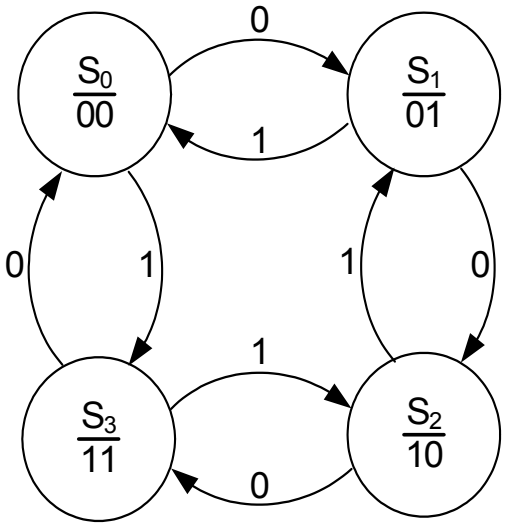
To design and build

- Modulo-4 up-down counter.
- A general sequence counter using the edge triggered D flip-flops of the FPGA.

Discussion:

Part 1: Modulo 4 up-down counter

A Modulo 4 up-down counter is a two-bit counter that can count up or down depending on the input selection. Figure 10-1 shows the state diagram for a two-bit up-down counter with a single input X. When X=0 the counter counts up and when X=1 the counter counts down. The output of this counter is the two Q outputs from the two D flip-flops.



STATE	Y_1Y_0
S_0	00
S_1	01
S_2	10
S_3	11

Figure 10-1: State Assignment for the Modulo 4 counter of part 1.

Part 2: General sequence counter

A general sequence counter is to be designed such that it counts in a fixed loop for X=1 and skips every next state for X=0, as shown in Figure 10-2. The design steps to implement this counter are similar to the design steps taken to implement the Modulo-4 counter in part1.

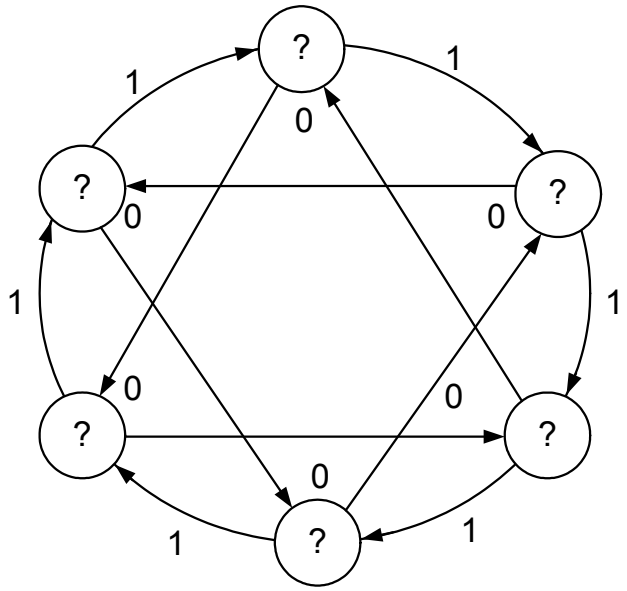


Figure 10-2: State Graph for the sequence counter of part 2.

Pre-Laboratory Assignments:

Part 1: Modulo 4 up-down counter

1. Read this experiment to become familiar with this laboratory assignment.

2. From the course textbook, read the section on sequential design using D flip-flops. Become familiar on how to convert a state diagram to a state transition table.

3. Include in your design the state diagram, state transition table, legend, next state equations, and schematic diagram.

4. A draft of the Test plan for part 1.

5. A draft of the Design Specification Plan for part 1

Part 2: General sequence counter

1. Obtain from the lab instructor the state sequence for the general sequence counter.

2. Design the sequence counter assigned by the lab instructor using D flip-flops.

3. Include in your design the state diagram, state transition table, legend, next state equations, and schematic diagram.

4. A draft of the Test Plan for part 2.

5. A draft of the Design Specification Plan part 2

Procedure:

Part 1: Modulo 4 up-down counter

1. Using the state assignments indicated in Figure 10-1, for the modulo 4 up-down counter, realize the Moore machine described by the state graph in Figure 10-1 using the edge triggered D flip-flops. Design this up-down counter in the schematic tool of the ISE using two D flip-flops (Symbols – Categories (Flip Flop) – Symbols (fd)) one for each bit of storage. Draw a logic schematic of the final design in your notebook. The clock input should be connected to a push button so that every time the push button is pressed, the shift register sees a clock pulse to either load data into it or shift its data left or right. Switches are notorious for producing many edge transitions when they go from 'on' to 'off' or 'off' to 'on' state. These extra edges result in extra clock edges to the shift register clock input when a single clock edge is expected. Since the D flip-flops are edge triggered, these extra edges result in extra counts.

2. Consider the switch de-bounce circuit given below. The CLK input should be connected to push button 0 (BTN0) and the CLK25MHz should be connected to pin 54 of the FPGA, which is a 25 MHz clock input. The circuit works as follows: When CLK input is '0', the 4 and 16 bit counters are cleared and held at a count of zero. Also the terminal count TC output on the 4 bit counter is zero during a clear operation. When CLK input is '1', the clear line is set to zero and the count enable CE inputs to the counters are set to one allowing the counters to start counting up. When the count of 2^{20} -1 (20 bits = 16 bits + 4 bits) is reached, the terminal count TC output is set to '1' on the four bit counter. It takes a total time of (2^{20} -1) /25,000,000 = 0.04794 seconds for the TC output to go high. By this time the switch contact noise has diminished.

   When the TC on the four bit counter goes high, it sets one of the inputs to the AND gate to zero and thus prevents the counter from counting any further until the CLK input goes low. Hence, this produces a single rising edge on the TC output from the four bit counter independent of the number of edges on the input clock CLK. If the CLK line is noisy producing many edges then the counter is reset and is delayed until the TC on the four bit counter is set to one. In summary, this circuit solves the problem of many edges on the CLK input due to switch contact noise (switch bounce) and the output from this circuit produces a clean rising edge on the TC output of the four bit counter. This rising edge can be used to clock the shift register without any switch noise.

3. Simulate the circuit designed in Step 1 and verify that it correctly implements the function of the up / down counter. Ignore the de-bounce circuit discussed in step 2 above for this simulation. There should be two inputs clock and X (up / down input) and two outputs (Q0 and Q1).

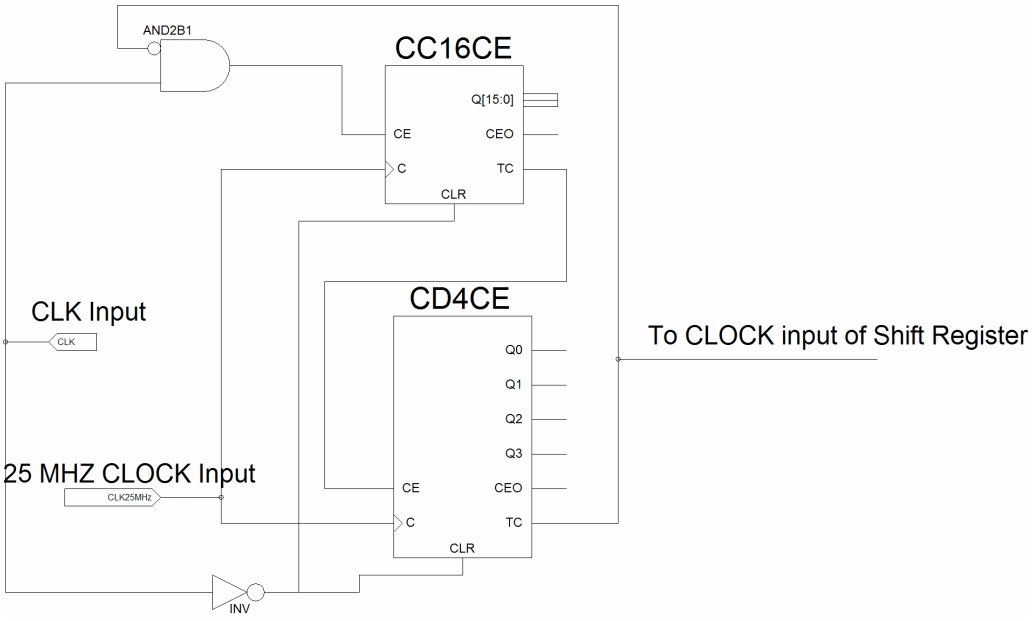


Figure 10-3: Switch De-bounce Circuit.

4. Download this circuit implemented in Step 3 to the BASYS board with X set to SW0 and Q_0 and Q_1 set to LED0 and LED1. The CLK input should be connected to push button 0 (BTN0). You will need to overwrite the pin for the CLK signal. Please ignore the warning in the PACER program due to overriding the global clock pin to push button 0. Ignore the de-bounce circuit for this step.

5. Use the EXPORT program to download the program. Run several cases showing that the modulo 4 up-down counter works correctly. Describe what is observed during these tests in your laboratory report.

6. Modify the modulo 4 up-down counter circuit of steps 1, 3, 4 and 5 above to include the de-bounce circuit of step 2. Connect the push button 0 to the CLK input which is one of the inputs of the AND gate shown in figure above. The output of this AND gate is connected to the count enable CE input of the 16 bit counter. Make sure that the 25MHz Clock input is connected to pin P54, which forms the clock inputs for the 4 and 16 bit counters. Finally, tie the terminal count TC output of the four bit counter to the clock inputs of the two D flip-flops of the up / down counter.

7. Download the design in Step 6 to the BASYS board with X set to SW0, Q_0 and Q_1 set to LED0 and LED1. The CLK input should be connected as described in step 6. This signal will need its pin over written. Ignore the warning in the PACER

program due to overriding the global clock pin to push button 0. This step includes the de-bounce circuit given in step 2.

8. Use the EXPORT program to download the program. Run several cases showing that the modulo 4 up-down counter works correctly. Describe what is observed during these tests in your laboratory report. Compare the results obtained in this step to that of steps 4 and 5. Include this comparison in your laboratory report.

Part 2: General Sequence Counter

1. Obtain from the lab instructor the state sequence for the sequence counter for part 2 of this experiment.

2. Design the sequence counter of Figure 10-2 in the schematic tool of the ISE using the required number of D flip-flops (fd) under Flip Flop category (one for each bit of storage). Draw a logic schematic in your notebook of the final design.

3. Simulate the design implemented in Step 2 and verify that it correctly implements the sequence counter. Ignore the de-bounce circuit described in step 2 of Part 1 for this simulation.

4. Download the design in Step 3 to the BASYS board with O_0 through O_N set to LED0 to LEDN (N is number of bits required for the sequence counter). The CLK input should be connected to push button 0 and the X input to SW0. Use the EXPORT program to download the program. Generate a new state transition table to verify that the design works correctly. Ignore the de-bounce circuit when performing the simulation. Describe what is observed during these tests in your laboratory report.

5. Modify the sequence counter circuit of steps 2, 3 and 4 above to include the de-bounce circuit of step 2 of Part 1 of this experiment. Connect the push button 0 to the CLK input which is one of the inputs of the AND gate as shown in figure. The output of this AND gate is connected to the count enable CE input of the 16 bit counter. Make sure that the 25MHz Clock input is connected to pin P54, which forms the clock inputs for the 4 and 16 bit counters. Finally, tie the terminal count TC output of the four bit counter to the clock inputs of the D flip-flops of the sequence counter.

6. Download the design in Step 5 to the BASYS board with O_0 through O_N set to LED0 to LEDN. The CLK input should be connected to push button 0 and the X input to SW0. Use the EXPORT program to download the program. Generate a new state transition table to verify that the design works correctly. Include the de-bounce circuit in step 2 of Part 1. Describe what is observed during these tests in your laboratory report. Compare the results obtained in this step to that of step 4. Include this comparison in your laboratory report.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. What happens if there is noise on the clock input for both part 1 and 2?
2. Would there be a problem if a non-bounceless switch were used as the input X?
3. Would there be a problem if a non-bounceless switch were used as the clock?
4. What are some typical applications of general sequence counters?
5. What factors determine the maximum clock frequency?

EXPERIMENT #11

Sequential Design

Objective:

To gain experience through the design investigation, actual network design, building, testing and documentation of a sequential network design problem.

Discussion:

Your company has asked you to design a specialized digital circuit for a good customer. The marketing and sales department of your company has developed a set of marketing requirements based upon the need of this customer. As with design problems there are many approaches that can implement these set of requirements. You are an inspiring engineer wishing to impress your boss with the best possible design. You need to look at least two alternative design approaches comparing the advantages and disadvantages of each. Being the competent and thorough designer you are, you must investigate all possibilities and make judgments as to which design you are going to implement. While designing the circuit, you should document your efforts such that at the completion of a working prototype, you can formally document your completed design effort.

Problem:

The company that you are employed as a digital designer specializes in custom electronic systems for unique applications. A good and old customer has met with the marketing and sales department of your company for the need of a sequential network to convert excess-3 code to BCD (Binary Coded Decimal) code. The input to the circuit is a serial input of four-bits with the first bit representing the least significant bit of the excess 3 binary code. The customer has limited the scope of this project so that only valid excess-3 codes will appear as serial inputs. The output is to be four bits in parallel presenting the BCD code and a single bit that is one when all four bits have been received. Finally, the customer requires that the circuit should output this valid BCD code until a reset input is set to one. At this time, the valid BCD code output signal goes to zero and the circuit is ready to accept additional inputs of four bit serial excess 3 codes. If the reset input is set to 1 during the receiving of an excess three code serial bits, it is ignored until the excess 3 code has been entered and a valid BCD output is given. Assume also that the reset input is a synchronous input.

There are several different approaches that can be used to implement this design. As an example, you can use a Moore sequential design or a shift register followed by combinational logic. Use your engineering judgment in making any assumptions, which will affect your design. The customer requires a digital circuit that uses fewest parts possible for both cost and reliability.

Design Approach:

The network specified above can be designed entirely using a Moore machine, as shown in the block diagram in Figure 11-1.

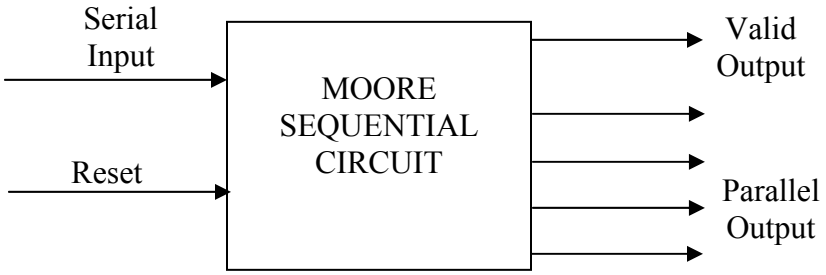


Figure 11-1: Moore Sequential Design

An alternate design uses a serial in, serial out Moore machine and a shift register for the parallel outputs, as shown in Figure 11-2.

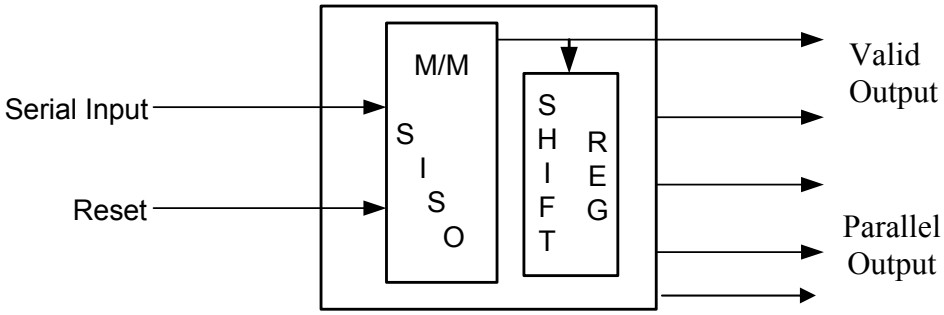


Figure 11-2: Serial In, Serial Out

Also you can implement your design using a finite memory machine as shown in Figure 11-3.

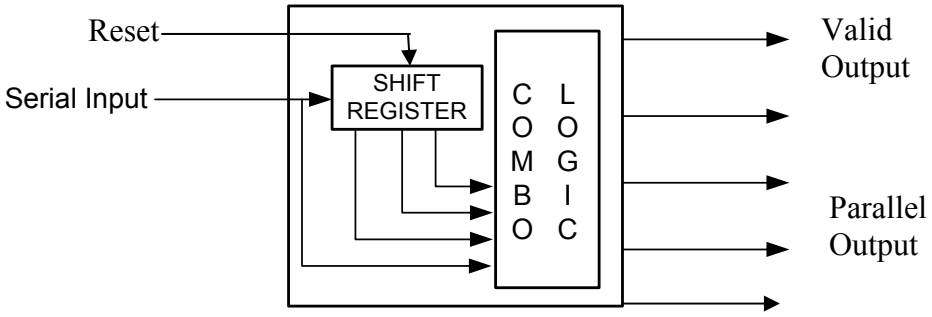


Figure 11-3: Finite Memory Machine

Pre-laboratory Assignment:

1. Choose two design approaches and compare their design and implementation methods. Provide detailed documentation of the advantages and disadvantages of each design. Describe any assumptions that were made in your design.

2. Choose one of the designs in Step 1 above and neatly draw its schematic including any switches, push buttons and LEDs needed. Do not forget to debounce any switches as discussed in the previous labs.

3. Write a detailed Design Specification Plan that describes your design and how it meets the marketing plan above. State clearly any assumptions that were made in your design.

4. Develop a Test Plan to test your design. This test plan needs to be detailed enough to cover all possible inputs.

Procedure:

1. Implement your design using the FPGA on the BASYS board. Include any schematics, state diagrams, state transition tables, and flip-flop equations. The use of D flip-flops with clock enable inputs can be quite helpful in your design.

2. Using the Test Plan from step 4, simulate your design using the ISE showing that your circuit works correctly.

3. Download your design to the BASYS board and use your Test Plan to verify the operational performance of your circuit.

Questions:

(To be incorporated within the Conclusion section of your lab report.)

1. What is a finite memory machine?

2. Discuss the advantages and disadvantages of Moore and Mealy machines.

3. Justify the two designs that where considered. Discuss the approach used in implementing these designs.

4. Justify the final design - pros and cons.

5. Discuss any assumptions that were made during the design process.

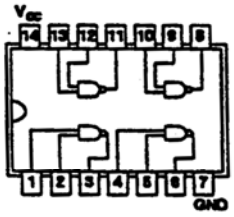
6. Justify that your Design Specification Plan meets the marketing requirement.

7. Justify that your Test Plan accurately tests your design approach. Does your Test Plan cover all possible inputs? If not, why?

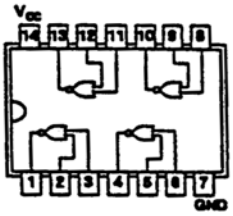
Appendix A

Spec Sheets

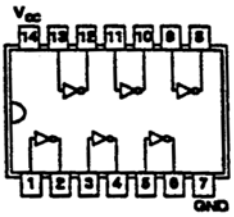
Pin Outs For Common Chips



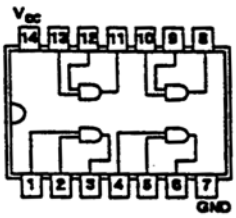
7400



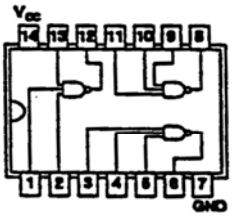
7402



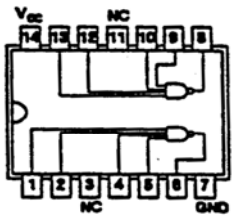
7404



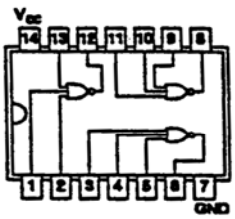
7408



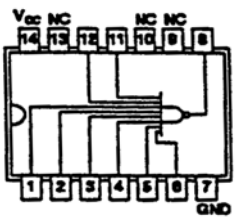
7410



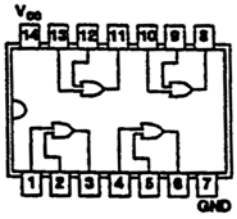
7420



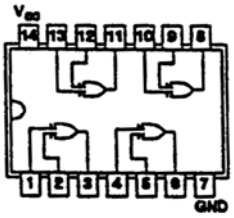
7427



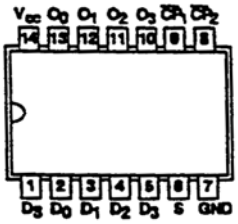
7430



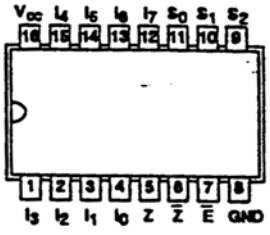
7432



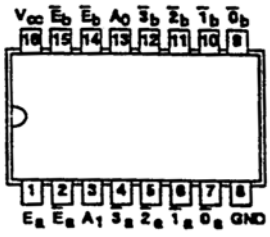
7486



7495



74151



74155

6810

Signetics

7400, LS00, S00
Gates

Quad Two-Input NAND Gate
Product Specification

Logic Products

TYPE	TYPICAL PROPAGATION DELAY	TYPICAL SUPPLY CURRENT (TOTAL)
7400	9ns	8mA
74LS00	9.5ns	1.6mA
74S00	3ns	15mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGE V_{CC} = 5V ±5%; T_A = 0°C to +70°C
Plastic DIP	N7400N, N74LS00N, N74S00N
Plastic SO	N74LS00D, N74S00D

NOTE:
For information regarding devices processed to Military Specifications, see the Signetics Military Products Data Manual.

FUNCTION TABLE

INPUTS		OUTPUT
A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

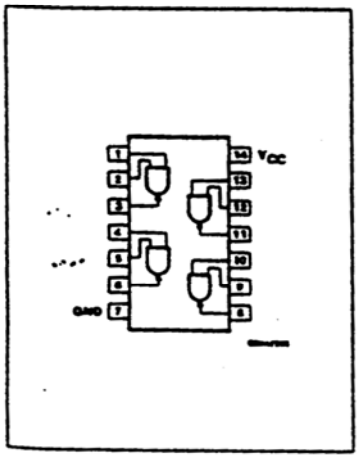
H = HIGH voltage level
L = LOW voltage level

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

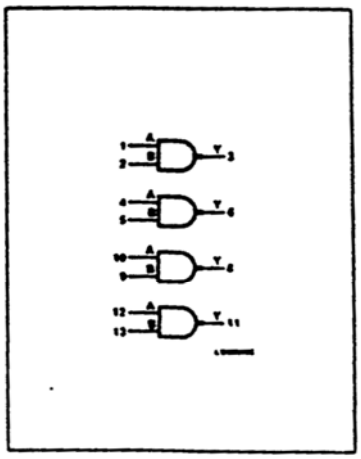
PINS	DESCRIPTION	74	74S	74LS
A, B	Inputs	1ul	1Sul	1LSul
Y	Output	10ul	10Sul	10LSul

NOTE:
Where a 74 unit load (ul) is understood to be 40μA I_{IH} and −1.6mA I_{IL}, a 74S unit load (Sul) is 50μA I_{IH} and −2.0mA I_{IL}, and 74LS unit load (LSul) is 20μA I_{IH} and −0.4mA I_{IL}.

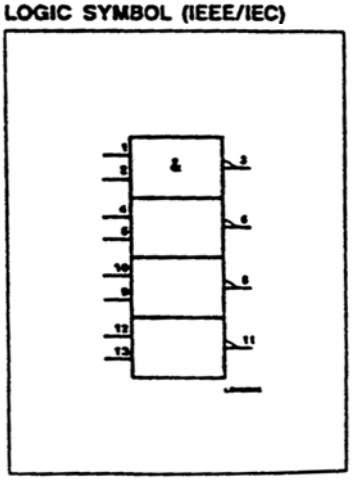
PIN CONFIGURATION



LOGIC SYMBOL



LOGIC SYMBOL (IEEE/IEC)



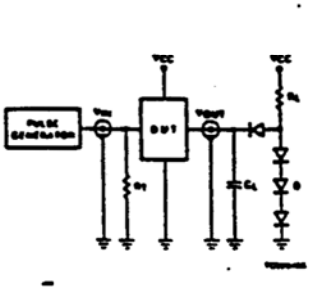
ABSOLUTE MAXIMUM RATINGS (Over operating free-air temperature range unless otherwise noted.)

PARAMETER		74	74LS	74S	UNIT
V_{CC}	Supply voltage	7.0	7.0	7.0	V
V_{IN}	Input voltage	−0.5 to +5.5	−0.5 to +7.0	−0.5 to +5.5	V
I_{IN}	Input current	−30 to +5	−30 to +1	−30 to +5	mA
V_{OUT}	Voltage applied to output in HIGH output state	−0.5 to +V_{CC}	−0.5 to +V_{CC}	−0.5 to +V_{CC}	V
T_A	Operating free-air temperature range	0 to 70			°C

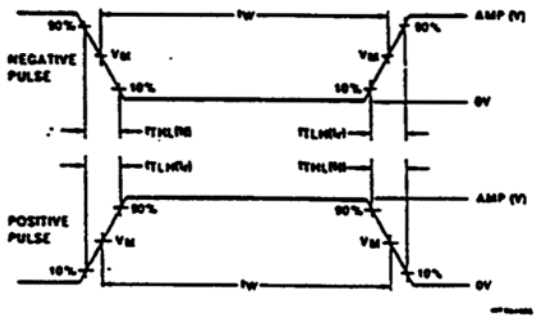
RECOMMENDED OPERATING CONDITIONS

PARAMETER		74			74LS			74S			UNIT
		Min	Nom	Max	Min	Nom	Max	Min	Nom	Max	
V_{CC}	Supply voltage	4.75	5.0	5.25	4.75	5.0	5.25	4.75	5.0	5.25	V
V_{IH}	HIGH-level input voltage	2.0			2.0			2.0			V
V_{IL}	LOW-level input voltage			+0.8			+0.8			+0.8	V
I_{IK}	Input clamp current			−12			−18			−18	mA
I_{OH}	HIGH-level output current			−400			−400			−1000	µA
I_{OL}	LOW-level output current			16			8			20	mA
T_A	Operating free-air temperature	0		70	0		70	0		70	°C

TEST CIRCUITS AND WAVEFORMS



Test Circuit For 74 Totem-Pole Outputs



Input Pulse Definition

DEFINITIONS
R_L = Load resistor to V_{CC}; see AC CHARACTERISTICS for value
C_L = Load capacitance includes jig and probe capacitance, see AC CHARACTERISTICS for value.
R_T = Termination resistance should be equal to Z_{OUT} of Pulse Generators.
D = Diodes are 1N916, 1N3064, or equivalent.
t_{TLH}, t_{THL} Values should be less than or equal to the table entries

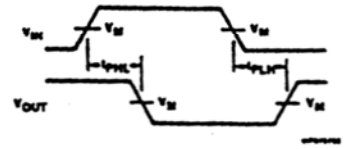
FAMILY	INPUT PULSE REQUIREMENTS				
	Amplitude	Rep. Rate	Pulse Width	t_{TLH}	t_{THL}
74	3.0V	1MHz	500ns	7ns	7ns
74LS	3.0V	1MHz	500ns	15ns	6ns
74S	3.0V	1MHz	500ns	2.5ns	2.5ns

DC ELECTRICAL CHARACTERISTICS (Over recommended operating free-air temperature range unless otherwise noted.)

PARAMETER		TEST CONDITIONS[1]		7400			74LS00			74S00			UNIT
				Min	Typ[2]	Max	Min	Typ[2]	Max	Min	Typ[2]	Max	
V_{OH}	HIGH-level output voltage	V_{CC} = MIN, V_{IH} = MIN, V_{IL} = MAX, I_{OH} = MAX		2.4	3.4		2.7	3.4		2.7	3.4		V
V_{OL}	LOW-level output voltage	V_{CC} = MIN, V_{IH} = MIN	I_{OL} = MAX		0.2	0.4		0.35	0.5			0.5	V
			I_{OL} = 4mA (74LS)					0.25	0.4				V
V_{IK}	Input clamp voltage	V_{CC} = MIN, I_I = I_{IK}				−1.5			−1.5			−1.2	V
I_I	Input current at maximum input voltage	V_{CC} = MAX	V_I = 5.5V			1.0						1.0	mA
			V_I = 7.0V						0.1				mA
I_{IH}	HIGH-level input current	V_{CC} = MAX	V_I = 2.4V			40							µA
			V_I = 2.7V						20			50	µA
I_{IL}	LOW-level input current	V_{CC} = MAX	V_I = 0.4V			−1.6			−0.4				mA
			V_I = 0.5V									−2.0	mA
I_{OS}	Short-circuit output current[3]	V_{CC} = MAX		−18		−55	−20		−100	−40		−100	mA
I_{CC}	Supply current (total)	V_{CC} = MAX	I_{CCH} Outputs HIGH		4	8		0.8	1.6		10	16	mA
			I_{CCL} Outputs LOW		12	22		2.4	4.4		20	36	mA

NOTES:
1. For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions for the applicable type.
2. All typical values are at V_{CC} = 5V, T_A = 25°C.
3. I_{OS} is tested with V_{OUT} = +0.5V and V_{CC} = V_{CC} MAX + 0.5V. Not more than one output should be shorted at a time and duration of the short circuit should not exceed one second.

AC WAVEFORM



V_M = 1.3V for 74LS; V_M = 1.5V for all other TTL families.

Waveform 1. Waveform For Inverting Outputs

AC ELECTRICAL CHARACTERISTICS T_A = 25°C, V_{CC} = 5.0V

PARAMETER		TEST CONDITIONS	74		74LS		74S		UNIT
			C_L = 15pF, R_L = 400Ω		C_L = 15pF, R_L = 2kΩ		C_L = 15pF, R_L = 280Ω		
			Min	Max	Min	Max	Min	Max	
t_{PLH} t_{PHL}	Propagation delay	Waveform 1		22 15		15 15		4.5 5.0	ns

Signetics

7495, LS95B
Shift Registers

4-Bit Shift Register
Product Specification

Logic Products

FEATURES
- Separate negative-edge-triggered shift and parallel load clocks
- Common mode control input
- Shift right serial input
- Synchronous shift or load capabilities

DESCRIPTION
The '95 is a 4-Bit Shift Register with serial and parallel synchronous operating modes. It has serial Data (D_S) and four parallel Data ($D_0 - D_3$) inputs and four Parallel outputs ($Q_0 - Q_3$). The serial or parallel mode of operation is controlled by a Mode Select input (S) and two Clock inputs ($\overline{CP}_1$ and $\overline{CP}_2$). The serial (shift right) or parallel data transfers occur synchronously with the HIGH-to-LOW transition of the selected Clock input.

When the Mode Select input (S) is HIGH, $\overline{CP}_2$ is enabled. A HIGH-to-LOW transition on enabled $\overline{CP}_2$ loads parallel data from the $D_0 - D_3$ inputs into the register. When S is LOW, CP_1 is enabled. A HIGH-to-LOW transition on enabled $\overline{CP}_1$ shifts the data from Serial input D_S to Q_0 and transfers the data in Q_0 to Q_1, Q_1 to Q_2, and Q_2 to Q_3 respectively (shift right). Shift left is accomplished by externally connecting Q_3 to D_2, Q_2 to D_1, Q_1 to D_0, and operating the '95 in the parallel mode (S = HIGH).

In normal operations the Mode Select (S) should change states only when both Clock inputs are LOW. However, changing S from HIGH-to-LOW while $\overline{CP}_2$ is LOW, or changing S from LOW-to-HIGH while $\overline{CP}_1$ is LOW will not cause any changes on the register outputs.

TYPE	TYPICAL f_{MAX}	TYPICAL SUPPLY CURRENT (TOTAL)
7495	36MHz	39mA
74LS95B	36MHz	13mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGE V_{CC} = 5V ±5%; T_A = 0°C to +70°C
Plastic DIP	N7495N, N74LS95BN

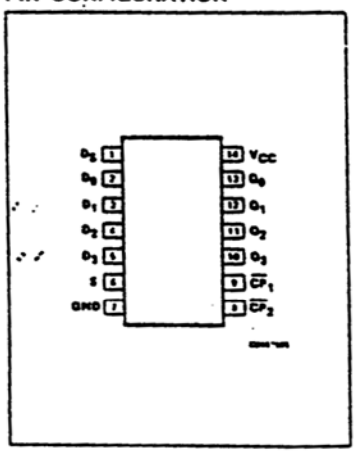
NOTE:
For information regarding devices processed to Military Specifications see the Signetics Military Products Data Manual.

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

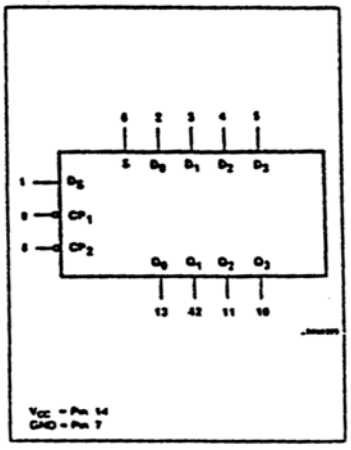
PINS	DESCRIPTION	74	74LS
S	Input	2ul	1LSul
Other	Inputs	1ul	1LSul
O	Output	10ul	10LSul

NOTE:
Where a 74 unit load (ul) is understood to be 40μA I_{IH} and −1.6mA I_{IL}, and a 74LS unit load (LSul) is 20μA I_{IH} and −0.4mA I_{IL}.

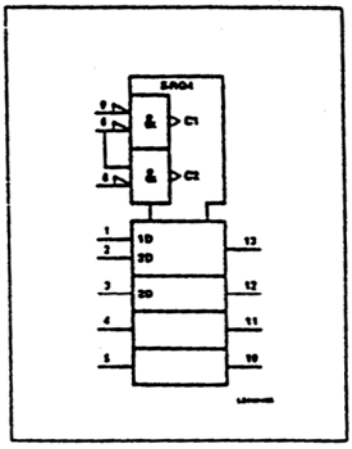
PIN CONFIGURATION



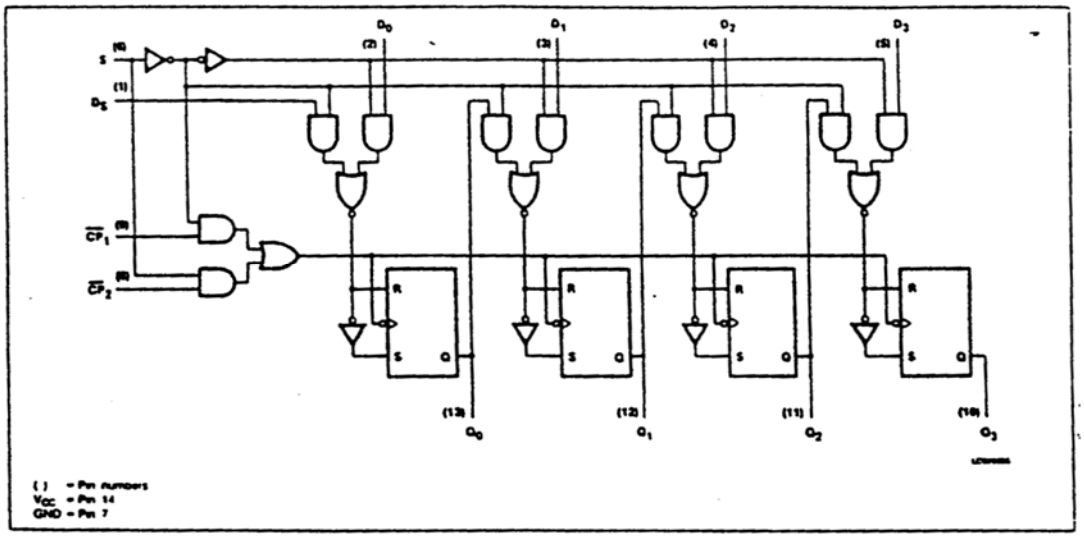
LOGIC SYMBOL



LOGIC SYMBOL (IEEE/IEC)



LOGIC DIAGRAM



FUNCTION TABLE

OPERATING MODE	INPUTS					OUTPUTS			
	S	CP_1	CP_2	D_S	D_N	Q_0	Q_1	Q_2	Q_3
Parallel load	H H	X X	↓ ↓	X X	l h	L H	L H	L H	L H
Shift right	L L	↓ ↓	X X	l h	X X	L H	q_0 q_0	q_1 q_1	q_2 q_2
Mode change	↑ ↑ ↓ ↓	L H X X	X X L H	X X X X	X X X X	no change undetermined no change undetermined			

H = HIGH voltage level steady state.
h = HIGH voltage level one set-up time prior to the HIGH-to-LOW clock transition
L = LOW voltage level steady state
l = LOW voltage level one set-up time prior to the HIGH-to-LOW clock transition
q = Lower case letters indicate the state of the referenced output one set-up time prior to the HIGH-to-LOW clock transition
X = Don't care.
↓ = HIGH-to-LOW transition of clock or mode select.
↑ = LOW-to-HIGH transition of mode select.

ABSOLUTE MAXIMUM RATINGS (Over operating free-air temperature range unless otherwise noted.)

	PARAMETER	74	74LS	UNIT
V_{CC}	Supply voltage	7.0	7.0	V
V_{IN}	Input voltage	−0.5 to +5.5	−0.5 to +7.0	V
I_{IN}	Input current	−30 to +5	−30 to +1	mA
V_{OUT}	Voltage applied to output in HIGH output state	−0.5 to +V_{CC}	+0.5 to +V_{CC}	V
T_A	Operating free-air temperature range	0 to 70		°C

Signetics

74151, LS151, S151
Multiplexers

8-Input Multiplexer
Product Specification

Logic Products

FEATURES
- Multifunction capability
- Complementary outputs
- See '251 for 3-state version

DESCRIPTION
The '151 is a logical implementation of a single-pole, 8-position switch with the switch position controlled by the state of three Select inputs, S_0, S_1, S_2. True (Y) and Complement ($\overline{Y}$) outputs are both provided. The Enable input ($\overline{E}$) is active LOW. When $\overline{E}$ is HIGH, the $\overline{Y}$ output is HIGH and the Y output is LOW, regardless of all other inputs. The logic function provided at the output is:

$$Y = \overline{E} \cdot (I_0 \cdot \overline{S}_0 \cdot \overline{S}_1 \cdot \overline{S}_2 + I_1 \cdot S_0 \cdot \overline{S}_1 \cdot \overline{S}_2 + I_2 \cdot \overline{S}_0 \cdot S_1 \cdot \overline{S}_2 + I_3 \cdot S_0 \cdot S_1 \cdot \overline{S}_2 + I_4 \cdot \overline{S}_0 \cdot \overline{S}_1 \cdot S_2 + I_5 \cdot S_0 \cdot \overline{S}_1 \cdot S_2 + I_6 \cdot \overline{S}_0 \cdot S_1 \cdot S_2 + I_7 \cdot S_0 \cdot S_1 \cdot S_2)$$

In one package the '151 provides the ability to select from eight sources of data or control information. The device can provide any logic function of four variables and its negation with correct manipulation.

TYPE	TYPICAL PROPAGATION DELAY (ENABLE TO $\overline{Y}$)	TYPICAL SUPPLY CURRENT (TOTAL)
74151	18ns	29mA
74LS151	12ns	6mA
74S151	9ns	45mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGE V_{CC} = 5V ±5%; T_A = 0°C to +70°C
Plastic DIP	N74151N, N74LS151N, N74S151N
Plastic SO	N74LS151D, N74S151D

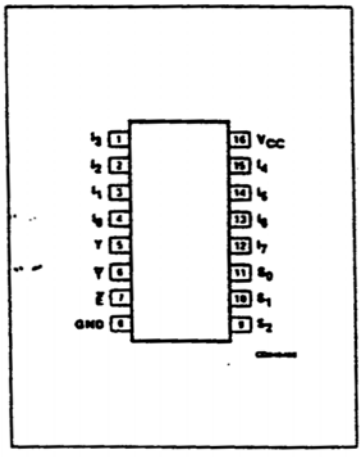
NOTE:
For information regarding devices processed to Military Specifications, see the Signetics Military Products Data Manual.

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

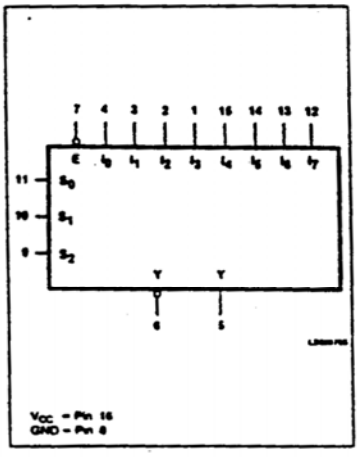
PINS	DESCRIPTION	74	74S	74LS
All	Inputs	1ul	1Sul	1LSul
All	Outputs	10ul	10Sul	10LSul

NOTE:
Where a 74 unit load (ul) is understood to be 40μA I_{IH} and −1.6mA I_{IL}, a 74S unit load (Sul) is 50μA I_{IH} and −2.0mA I_{IL}, and 74LS unit load (LSul) is 20μA I_{IH} and −0.4mA I_{IL}.

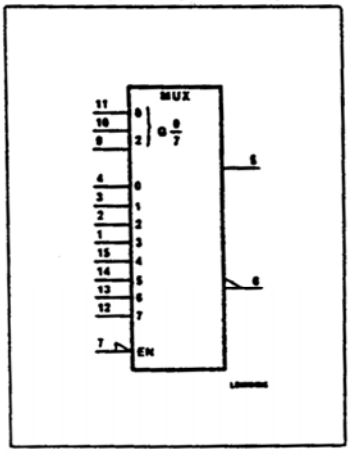
PIN CONFIGURATION



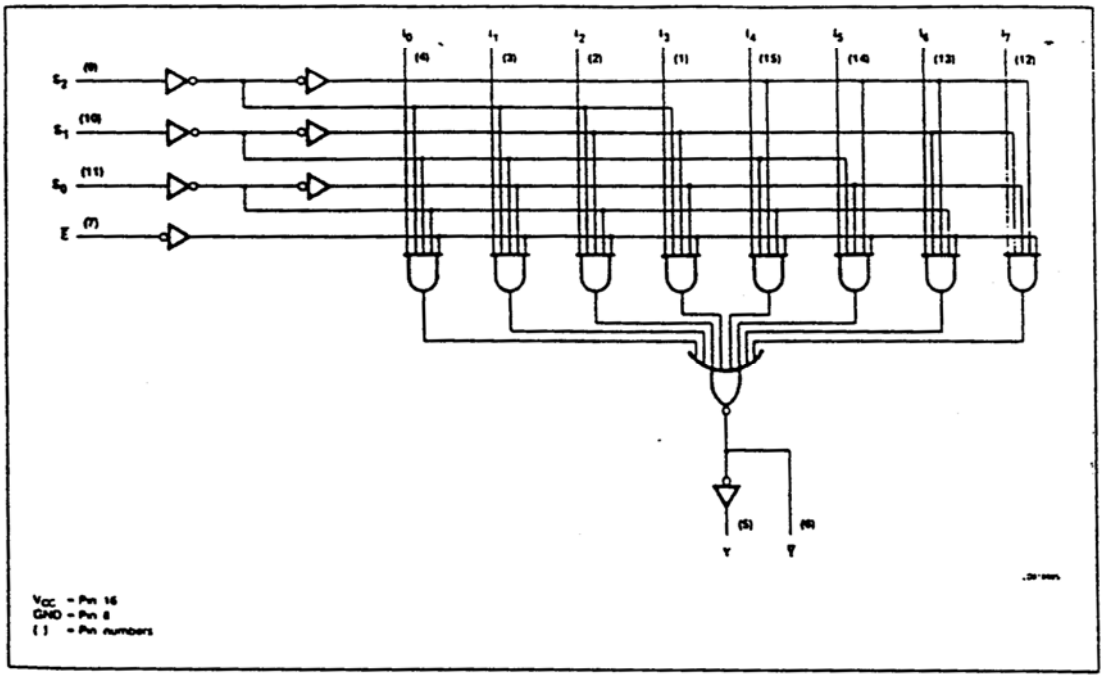
LOGIC SYMBOL



LOGIC SYMBOL (IEEE/IEC)



LOGIC DIAGRAM



FUNCTION TABLE

INPUTS												OUTPUTS	
$\overline{E}$	S_2	S_1	S_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	$\overline{Y}$	Y
H	X	X	X	X	X	X	X	X	X	X	X	H	L
L	L	L	L	L	X	X	X	X	X	X	X	H	L
L	L	L	L	H	X	X	X	X	X	X	X	L	H
L	L	L	H	X	L	X	X	X	X	X	X	H	L
L	L	L	H	X	H	X	X	X	X	X	X	L	H
L	L	H	L	X	X	L	X	X	X	X	X	H	L
L	L	H	L	X	X	H	X	X	X	X	X	L	H
L	L	H	H	X	X	X	L	X	X	X	X	H	L
L	L	H	H	X	X	X	H	X	X	X	X	L	H
L	H	L	L	X	X	X	X	L	X	X	X	H	L
L	H	L	L	X	X	X	X	H	X	X	X	L	H
L	H	L	H	X	X	X	X	X	L	X	X	H	L
L	H	L	H	X	X	X	X	X	H	X	X	L	H
L	H	H	L	X	X	X	X	X	X	L	X	H	L
L	H	H	L	X	X	X	X	X	X	H	X	L	H
L	H	H	H	X	X	X	X	X	X	X	L	H	L
L	H	H	H	X	X	X	X	X	X	X	H	L	H

H = HIGH voltage level
L = LOW voltage level
X = Don't care

Signetics

74155, LS155
Decoders/Demultiplexers

Dual 2-Line To 4-Line Decoder/Demultiplexer
Product Specification

Logic Products

FEATURES
- Common Address inputs
- True or complement data demultiplexing
- Dual 1-of-4 or 1-of-8 decoding
- Function generator applications

DESCRIPTION
The '155 is a Dual 1-of-4 Decoder/Demultiplexer with common Address inputs and separate gated Enable inputs. Each decoder section, when enabled, will accept the binary weighted Address input (A_0, A_1) and provide four mutually exclusive active-LOW outputs ($\overline{0} - \overline{3}$). When the enable requirements of each decoder are not met, all outputs of that decoder are HIGH.

TYPE	TYPICAL PROPAGATION DELAY	TYPICAL SUPPLY CURRENT (TOTAL)
74155	18ns	25mA
74LS155	17ns	6.1mA

ORDERING CODE

PACKAGES	COMMERCIAL RANGE V_{CC} = 5V ±5%; T_A = 0°C to +70°C
Plastic DIP	N74155N, N74LS155N
Plastic SO	N74LS155D

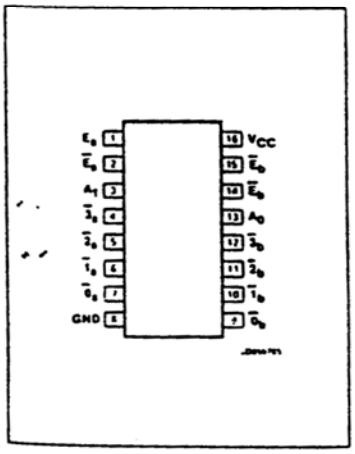
NOTE:
For information regarding devices processed to Military Specifications, see the Signetics Military Products Data Manual.

INPUT AND OUTPUT LOADING AND FAN-OUT TABLE

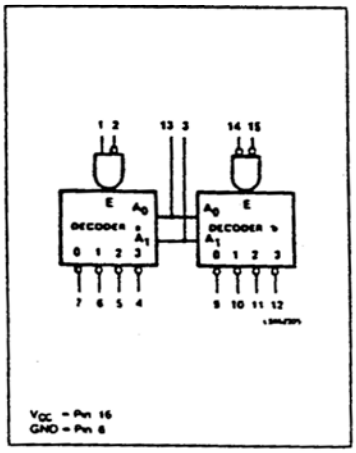
PINS	DESCRIPTION	74	74LS
All	Inputs	1ul	1LSul
All	Outputs	10ul	10LSul

NOTE:
Where a 74 unit load (ul) is understood to be 40μA I_{IH} and −1.6mA I_{IL}, and a 74LS unit load (LSul) is 20μA I_{IH} and −0.4mA I_{IL}.

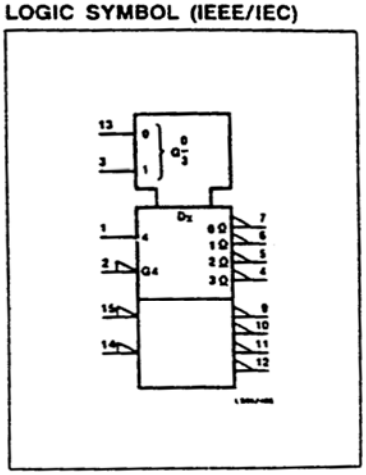
PIN CONFIGURATION



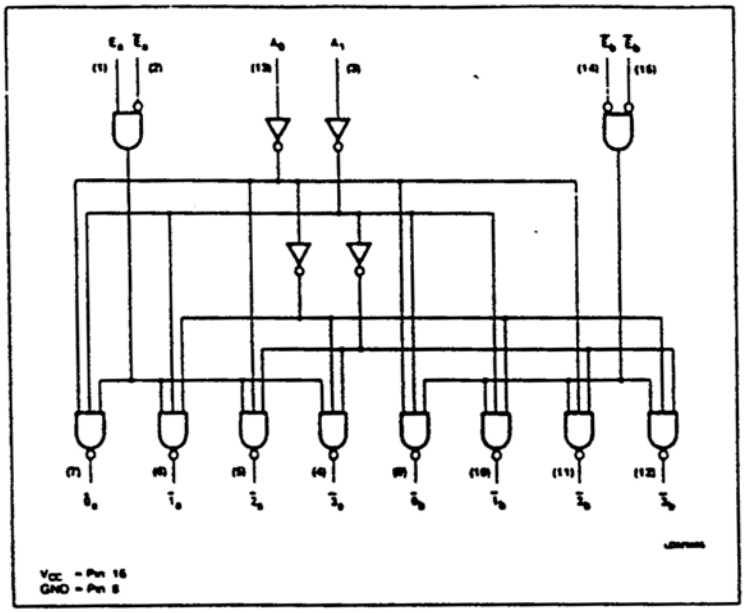
LOGIC SYMBOL



LOGIC SYMBOL (IEEE/IEC)



LOGIC DIAGRAM



Both decoder sections have a 2-input enable gate. For decoder "a" the enable gate requires one active-HIGH input and one active-LOW input ($E_a \cdot \overline{E}_a$). Decoder "a" can accept either true or complemented data in demultiplexing applications, by using the $\overline{E}_a$ or E_a inputs respectively. The decoder "b" enable gate requires two active-LOW inputs ($\overline{E}_b \cdot \overline{E}_b$). The device can be used as a 1-of-8 decoder/demultiplexer by tying E_a to $\overline{E}_b$ and relabeling the common connection address as (A_2); forming the common enable by connecting the remaining $\overline{E}_b$ and $\overline{E}_a$.

FUNCTION TABLE

ADDRESS		ENABLE "a"		OUTPUT "a"				ENABLE "b"		OUTPUT "b"			
A_0	A_1	E_a	$\overline{E}_a$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	E_b	$\overline{E}_b$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
X	X	L	X	H	H	H	H	H	X	H	H	H	H
X	X	X	H	H	H	H	H	X	H	H	H	H	H
L	L	H	L	L	H	H	H	L	L	L	H	H	H
H	L	H	L	H	L	H	H	L	L	H	L	H	H
L	H	H	L	H	H	L	H	L	L	H	H	L	H
H	H	H	L	H	H	H	L	L	L	H	H	H	L

H = HIGH voltage level
L = LOW voltage level
X = Don't care

ABSOLUTE MAXIMUM RATINGS (Over operating free-air temperature range unless otherwise noted.)

PARAMETER		74	74LS	UNIT
V_{CC}	Supply voltage	7.0	7.0	V
V_{IN}	Input voltage	−0.5 to +5.5	−0.5 to +7.0	V
I_{IN}	Input current	−30 to +5	−30 to +1	mA
V_{OUT}	Voltage applied to output in HIGH output state	−0.5 to +V_{CC}	−0.5 to +V_{CC}	V
T_A	Operating free-air temperature range	0 to 70		°C

MOTOROLA

MCM6810

128 × 8-BIT STATIC RANDOM ACCESS MEMORY

The MCM6810 is a byte-organized memory designed for use in bus-organized systems. It is fabricated with N-channel silicon-gate technology. For ease of use, the device operates from a single power supply, has compatibility with TTL and DTL, and needs no clocks or refreshing because of static operation.

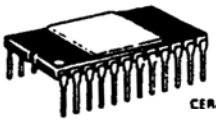
The memory is compatible with the M6800 Microcomputer Family, providing random storage in byte increments. Memory expansion is provided through multiple Chip Select inputs.

- Organized as 128 Bytes of 8 Bits
- Static Operation
- Bidirectional Three-State Data Input/Output
- Six Chip Select Inputs (Four Active Low, Two Active High)
- Single 5-Volt Power Supply
- TTL Compatible
- Maximum Access Time = 450 ns – MCM6810
  360 ns – MCM68A10
  250 ns – MCM68B10

MOS
(N-CHANNEL, SILICON-GATE)

128 × 8-BIT STATIC
RANDOM ACCESS
MEMORY

P SUFFIX
PLASTIC PACKAGE
CASE 709



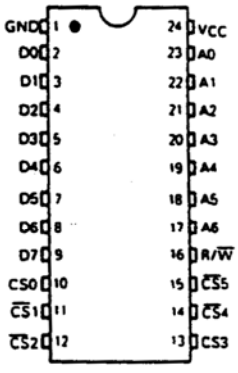
L SUFFIX
CERAMIC PACKAGE
CASE 716

S SUFFIX
CERDIP PACKAGE
CASE 623

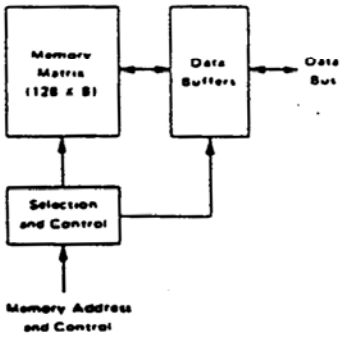
ORDERING INFORMATION

Package Type	Frequency (MHz)	Temperature	Order Number
Ceramic L Suffix	1.0	0°C to 70°C	MCM6810L
	1.0	−40°C to 85°C	MCM6810CL
	1.5	0°C to 70°C	MCM68A10L
	1.5	−40°C to 85°C	MCM68A10CL
	2.0	0°C to 70°C	MCM68B10L
Plastic P Suffix	1.0	0°C to 70°C	MCM6810P
	1.0	−40°C to 85°C	MCM6810CP
	1.5	0°C to 70°C	MCM68A10P
	1.5	−40°C to 85°C	MCM68A10CP
	2.0	0°C to 70°C	MCM68B10P
Cerdip S Suffix	1.0	0°C to 70°C	MCM6810S
	1.0	−40°C to 85°C	MCM6810CS
	1.5	0°C to 70°C	MCM68A10S
	1.5	−40°C to 85°C	MCM68A10CS
	2.0	0°C to 70°C	MCM68B10S

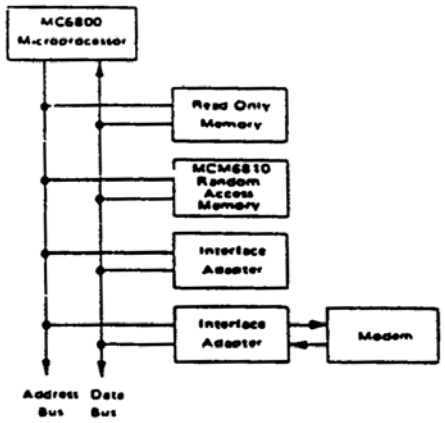
PIN ASSIGNMENT



MCM6810 RANDOM ACCESS MEMORY
BLOCK DIAGRAM



M6800 MICROCOMPUTER FAMILY
BLOCK DIAGRAM



MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.3 to +7.0	V
Operating Temperature Range MCM6810, MCM68A10, MCM68B10 MCM6810C, MCM68A10C	T_A	T_L to T_H 0 to +70 −40 to +85	°C
Storage Temperature Range	T_{stg}	−65 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields, however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage (e.g., either V_{SS} or V_{CC}).

THERMAL CHARACTERISTICS

Characteristics	Symbol	Value	Unit
Thermal Resistance Ceramic Plastic Cerdip	θ_{JA}	 60 120 65	°C/W

POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from

$$T_J = T_A + (P_D \theta_{JA}) \quad (1)$$

Where

T_A = Ambient Temperature, °C

θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W

$P_D = P_{INT} + P_{PORT}$

$P_{INT} = I_{CC} \times V_{CC}$, Watts — Chip Internal Power

P_{PORT} = Port Power Dissipation, Watts — User Determined

For most applications $P_{PORT} \ll P_{INT}$ and can be neglected. P_{PORT} may become significant if the device is configured to drive Darlington bases or sink LED loads.

An approximate relationship between P_D and T_J (if P_{PORT} is neglected) is

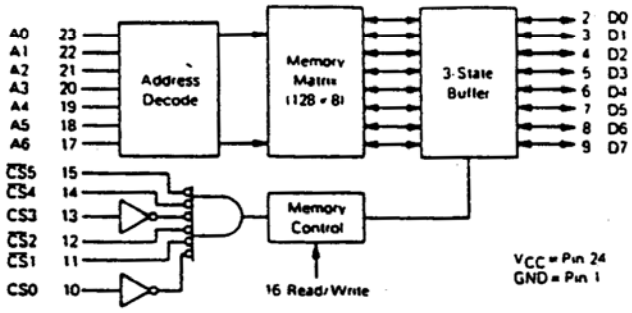
$$P_D = K \div (T_J + 273°C) \quad (2)$$

Solving equations 1 and 2 for K gives

$$K = P_D \cdot (T_A + 273°C) + \theta_{JA} \cdot P_D^2 \quad (3)$$

Where K is a constant pertaining to the particular part. K can be determined from equation 3 by measuring P_D (at equilibrium) for a known T_A. Using this value of K the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

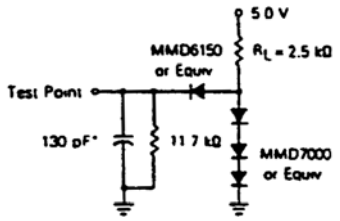
BLOCK DIAGRAM



DC ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 Vdc ± 5%, V_{SS} = 0, $T_A = T_L$ to T_H unless otherwise noted)

Characteristic	Symbol	Min	Max	Unit
Input High Voltage	V_{IH}	V_{SS} + 2.0	V_{CC}	V
Input Low Voltage	V_{IL}	V_{SS} − 0.3	V_{SS} + 0.8	V
Input Current (A_n, R/$\overline{W}$, $\overline{CS}_n$) (V_{in} = 0 to 5.25 V)	I_{in}	–	2.5	μA
Output High Voltage (I_{OH} = −205 μA)	V_{OH}	2.4	–	V
Output Low Voltage (I_{OL} = 1.6 mA)	V_{OL}	–	0.4	V
Output Leakage Current (Three-State) (CS = 0.8 V or $\overline{CS}$ = 2.0 V, V_{out} = 0.4 V to 2.4 V)	I_{TSI}	–	10	μA
Supply Current (V_{CC} = 5.25 V, All Other Pins Grounded) 1.0 MHz / 1.5, 2.0 MHz	I_{CC}	– / –	80 / 100	mA
Input Capacitance (A_n, R/$\overline{W}$, CS_n, $\overline{CS}_n$) (V_{in} = 0, T_A = 25°C, f = 1.0 MHz)	C_{in}	–	7.5	pF
Output Capacitance (D_n) (V_{out} = 0, T_A = 25°C, f = 1.0 MHz, CSO = 0)	C_{out}	–	12.5	pF

AC TEST LOAD



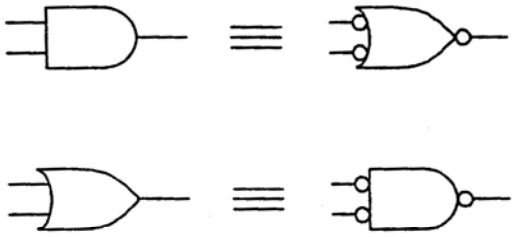
*Includes Jig Capacitance

Appendix B

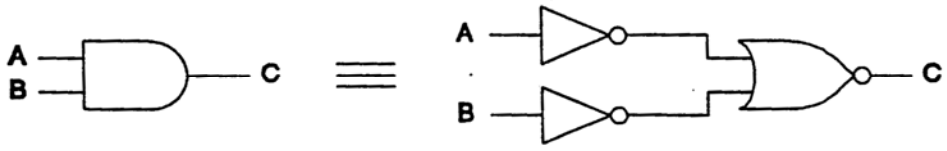
NAND/NOR Translations

Theory:

The following logic circuits are equivalent. Note: The bubble represents an inversion operation. Please reference Section 4.4 of the Roth text for elaboration.



Thus an *AND* gate can be replaced with three gates: a *NOR* gate and two inverter gates.

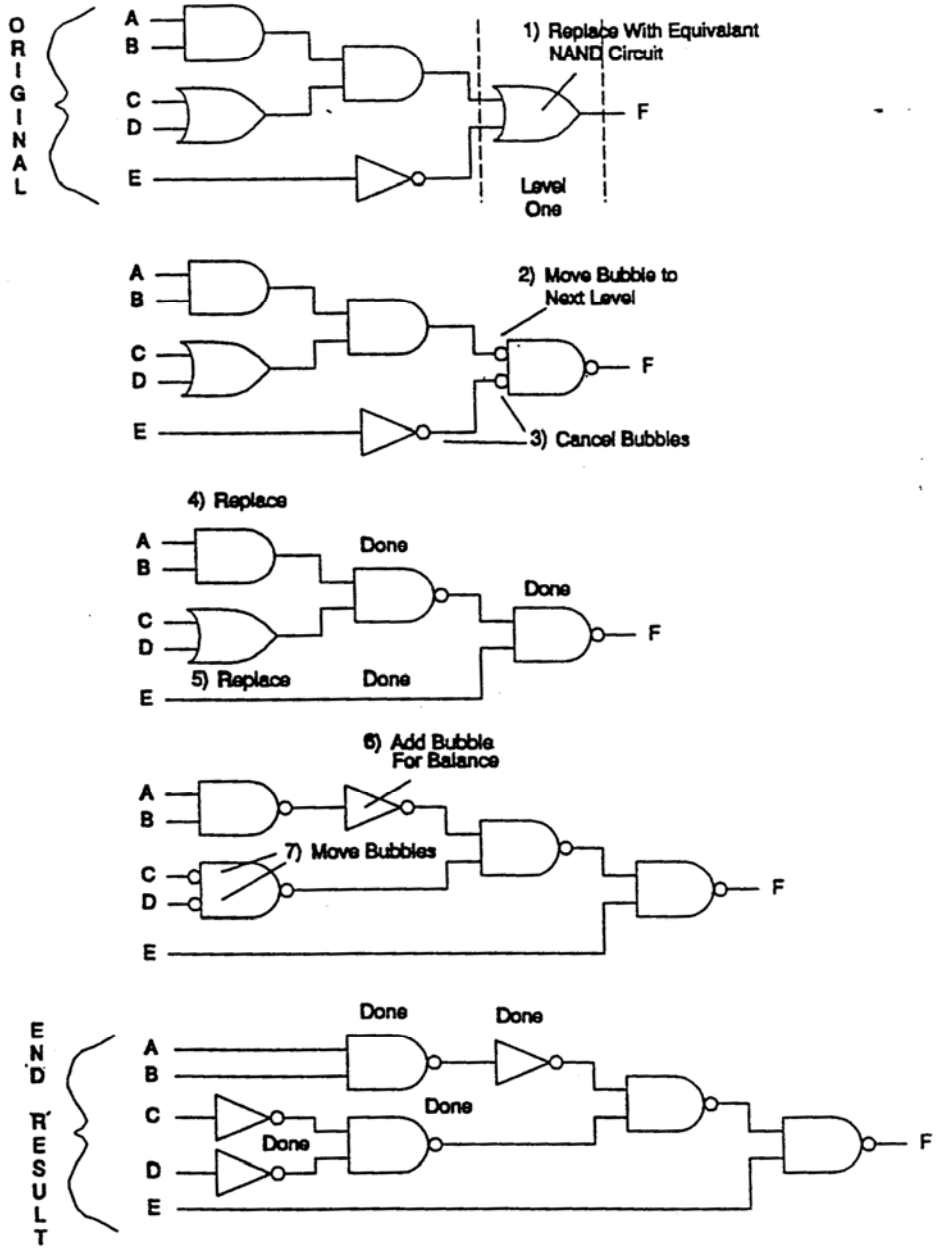


Similarly an *OR* gate can be replaced with a *NAND* gate and two invert gates. The translation of AOI (*AND*, *OR*, Invert) logic to *NAND/NOR* logic can be accomplished by replacing, adding, and moving symbols on a logic diagram or equivalently by manipulating the corresponding Boolean expressions.

General Procedure:

Replace *AND* or Invert gates with *NAND* or *NOR* gates and add "bubbles" to maintain equivalence. If two bubbles are in series (next to each other in the same path), omit both bubbles. Please note that the procedure in Section 8.5 of the Roth text is similar. In the latter procedure, the logic diagram must be represented as levels of *OR*s (*AND*s) followed by levels of *AND*s (*OR*s). If a level is missing (e.g., an *AND* gate followed by an *AND* gate) then a ONE input *OR* (*AND*) is inserted. Section 8.5 of the Roth text can then be applied. See examples which follow. Start at level ONE. The steps in the procedures are noted.

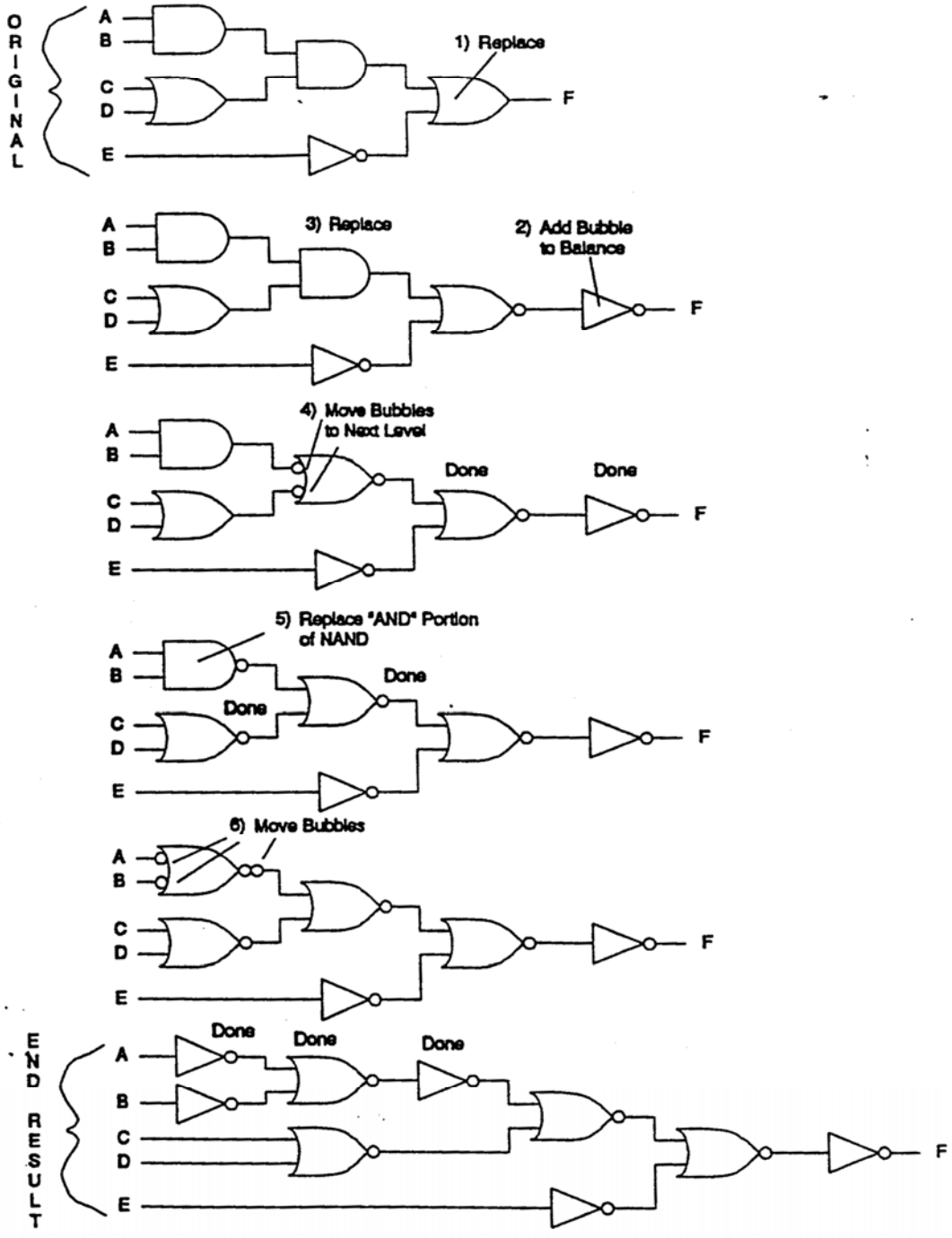
Example 1: Translation to *NAND/NAND*



Summary:

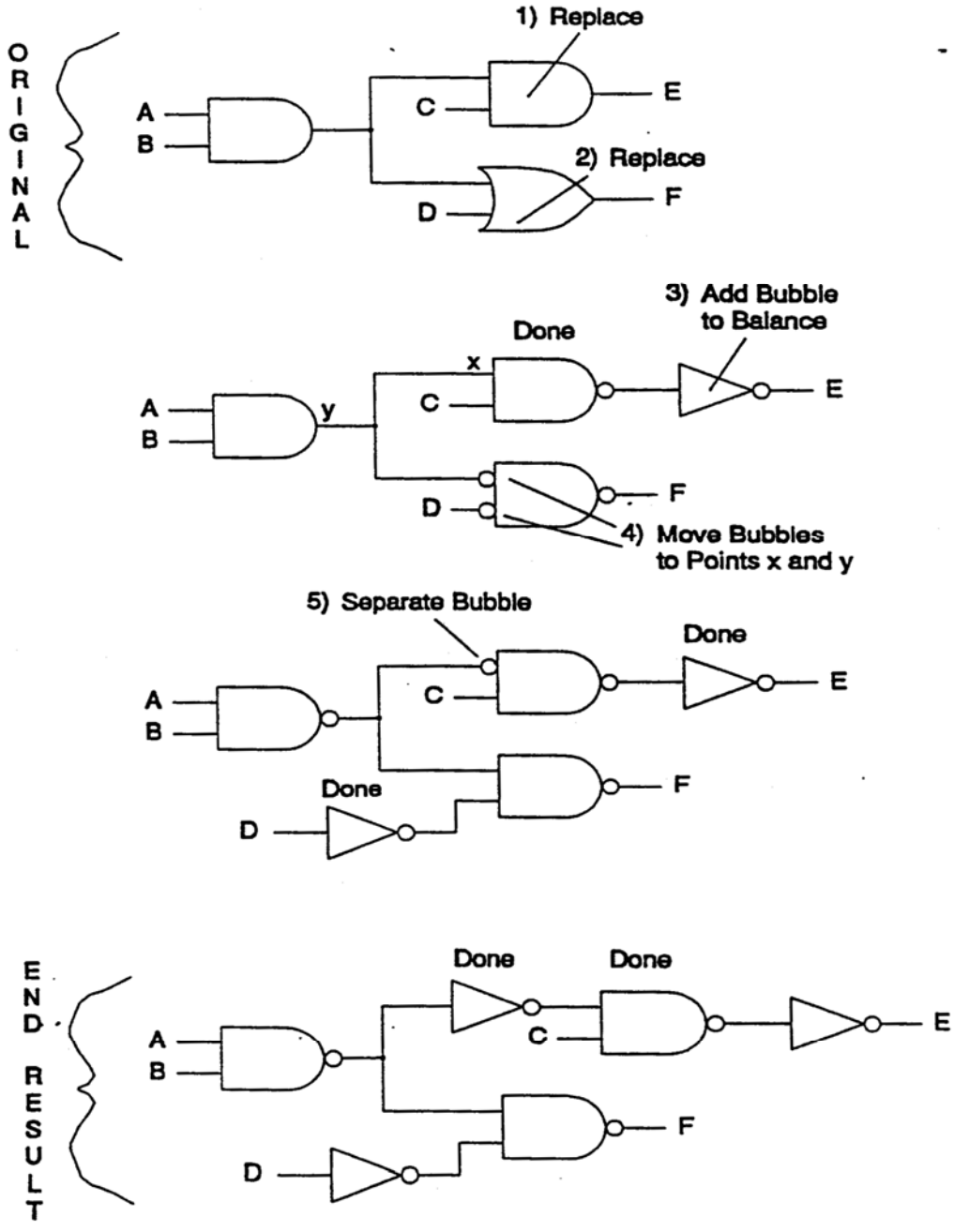
NAND realization of the original logic circuit requires 7 two-input *NAND* gates. (An inverter can be implemented with a two-input *NAND* gate with both inputs tied together.)

Example 2: Translation to *NOR/NOR*



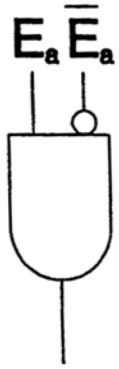
Example 3: Circuits With Shared Gates

Note: Special caution is required for circuits which share gates.



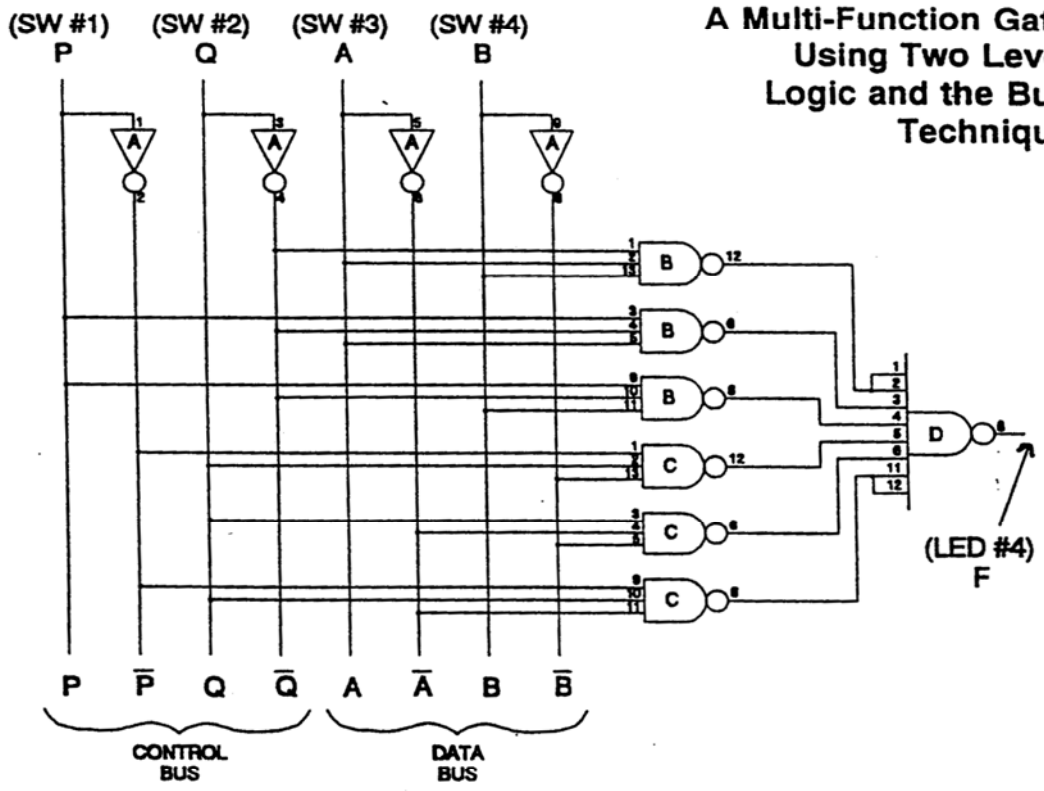
Interpretation of Data Sheet Diagrams

Logic circuits implemented in *NAND* or *NOR* logic are often difficult to interpret in terms of the basic logic functions of *AND*, *OR* and Invert. Very often a circuit is presented in a manner to facilitate such an interpretation. An example is the input enable gate of the IC74155.



The gate is shown as an *AND* gate. The inputs are E_a and $\overline{E}_a$. The $\overline{E}_a$ signal is called an active low signal. The gate performs a functional *AND*ing of two signals, one an active high (E_a) and one an active low ($\overline{E}_a$). Both signals must be active for the output to be active (which is an active high). The operation is an *AND* operation of two active levels. This manner of interpretation is conveniently represented by a bubble rather than an invert symbol at the input of the gate.

Sample Schematic Diagram



A	B	C	D
7404	7410	7410	7430

P	Q	Function
0	0	*AND*
0	1	*NAND*
1	0	*OR*
1	1	*NOR*

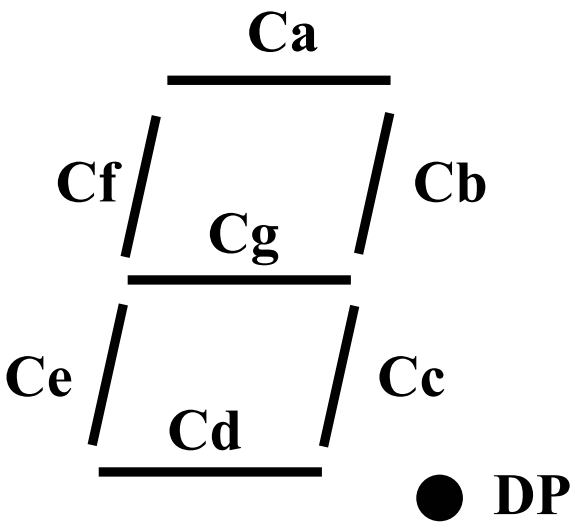
$$F(P,Q,A,B) = \overline{Q}AB + P\overline{Q}A + P\overline{Q}B + \overline{P}Q\overline{B} + \overline{A}Q\overline{B} + \overline{P}Q\overline{A}$$

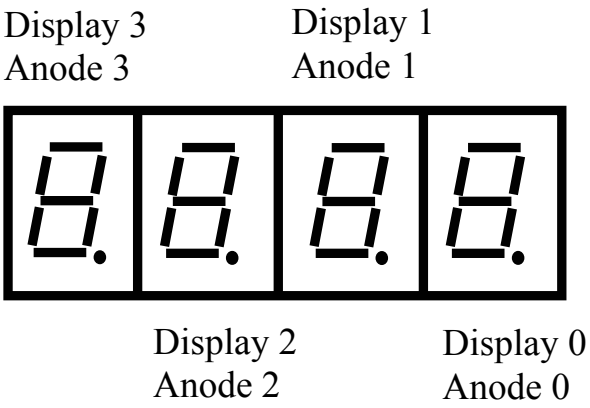
APPENDIX D
BASYS BOARD

DEVICE	PIN #
Switches	
SW0	38
SW1	36
SW2	29
SW3	24
SW4	18
SW5	12
SW6	10
SW7	6
Push Buttons	
BTN0	69
BTN1	48
BTN2	47
BTN3	41
LEDS	
LED0	15
LED1	14
LED2	8
LED3	7
LED4	5
LED5	4
LED6	3
LED7	2
7-SEG Displays	
CA	25
CB	16
CC	23
CD	21
CE	20
CF	17
CG	83
DP	22
ANODE 3	34
ANODE 2	33
ANODE 1	32
ANODE 0	26
25 MHz Clock	54

BASYS 2 BOARD

DEVICE	PIN #
Switches	
SW0	P11
SW1	L3
SW2	K3
SW3	B4
SW4	G3
SW5	F3
SW6	E2
SW7	N3
Push Buttons	
BTN0	G12
BTN1	C11
BTN2	M4
BTN3	A7
LEDS	
LED0	M5
LED1	M11
LED2	P7
LED3	P6
LED4	N5
LED5	N4
LED6	P4
LED7	G1
7-SEG Displays	
CA	L14
CB	H12
CC	N14
CD	N11
CE	P12
CF	L13
CG	M12
DP	N13
ANODE 3	F12
ANODE 2	J12
ANODE 1	M13
ANODE 0	K14
25 MHz Clock	B8





To turn on a segment, a value of 0 is needed on the anodes and a value of 0 on the appropriate cathodes Ca-Cg and the Decimal point DP

APPENDIX E

With permission from and thanks to Peter M. Nyasulu

Introduction to Verilog

Table of Contents

1. Introduction

Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL.

HDL's allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog can be used to describe designs at four levels of abstraction:
(i) Algorithmic level (much like c code with if, case and loop statements).
(ii) Register transfer level (RTL uses registers connected by Boolean equations).
(iii) Gate level (interconnected AND, NOR etc.).
(iv) Switch level (the switches are MOS transistors inside gates).
The language also defines constructs that can be used to control the input and output of simulation.

More recently Verilog is used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also the way the code is written will greatly effect the size and speed of the synthesized circuit. Most readers will want to synthesize their circuits, so nonsynthesizable constructs should be used only for *test benches*. These are program modules used to generate I/O needed to simulate the rest of the design. The words "not synthesizable" will be used for examples and constructs as needed that do not synthesize.

There are two types of code in most HDLs:

Structural, which is a verbal wiring diagram without storage.

```
assign a=b & c | d;        /* "|" is a OR */
assign d = e & (~c);
```

Here the order of the statements does not matter. Changing e will change a.

Procedural which is used for circuits with storage, or as a convenient way to write conditional logic.

```
always @(posedge clk)   // Execute the next statement on every rising clock edge.
count <= count+1;
```

Procedural code is written like c code and assumes every assignment is stored in memory until over written. For synthesis, with flip-flop storage, this type of thinking generates too much storage. However people prefer procedural code because it is usually much easier to write, for example, **if** and **case** statements are only allowed in procedural code. As a result, the synthesizers have been constructed which can recognize certain styles of procedural code as actually combinational. They generate a flip-flop only for left-hand variables which truly need to be stored. However if you stray from this style, beware. Your synthesis will start to fill with superfluous latches.

This manual introduces the basic and most common Verilog behavioral and gate-level modelling constructs, as well as Verilog compiler directives and system functions. Full description of the language can be found in *Cadence Verilog-XL Reference Manual* and *Synopsys HDL Compiler for Verilog Reference Manual*. The latter emphasizes only those Verilog constructs that are supported for synthesis by the *Synopsys Design Compiler* synthesis tool.

In all examples, Verilog keyword are shown in **boldface**. Comments are shown in *italics*.

2. Lexical Tokens

Verilog source text files consists of the following lexical tokens:

2.1. White Space

White spaces separate words and can contain spaces, tabs, new-lines and form feeds. Thus a statement can extend over multiple lines without special continuation characters.

2.2. Comments

Comments can be specified in two ways (exactly the same way as in C/C++):

- Begin the comment with double slashes (//). All text between these characters and the end of the line will be ignored by the Verilog compiler.
- Enclose comments between the characters **/*** and ***/**. Using this method allows you to continue comments on more than one line. This is good for "commenting out" many lines code, or for very brief in-line comments.

Example 2 .1

```
a = c + d;              // this is a simple comment
/* however, this comment continues on more
   than one line */
assign y = temp_reg;
assign x=ABC   /* plus its compliment*/  +  ABC_
```

2.3. Numbers

Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal (See Sect. 6.1. for details on number notation).
Examples are 3'**b**001, *a 3-bit numbe*r, 5'd30, (*=5'b11110*), and 16'**h**5ED4, (*=16'd24276*)

2.4. Identifiers

Identifiers are user-defined words for variables, function names, module names, block names and instance names. Identifiers begin with a letter or underscore (Not with a number or $) and can include any number of letters, digits and underscores. Identifiers in Verilog are case-sensitive.

Syntax

allowed symbols
ABCDE . . . abcdef. . . 1234567890 _$
not allowed: anything else especially
- & # @

Example 2 .2

```
adder               // use underscores to make your
by_8_shifter        // identifiers more meaningful
_ABC_       /* is not the same as */    _abc_
Read_       // is often used for NOT Read
```

2.5. Operators

Operators are one, two and sometimes three characters used to perform operations on variables.
Examples include >, **+**, **~**, **&**, **!=**. Operators are described in detail in "Operators" on p. 6.

2.6. Verilog Keywords

These are words that have special meaning in Verilog. Some examples are **assign, case, while, wire, reg, and, or, nand,** and **module**. They should not be used as identifiers. Refer to *Cadence Verilog-XL Reference Manual* for a complete listing of Verilog keywords. A number of them will be introduced in this manual. Verilog keywords also includes Compiler Directives (Sect. 15.) and System Tasks and Functions (Sect. 16.).

3. Gate-Level Modelling

Primitive logic gates are part of the Verilog language. Two properties can be specified, *drive_strength* and *delay.*

Drive_strength specifies the strength at the gate outputs. The strongest output is a direct connection to a source, next comes a connection through a conducting transistor, then a resistive pull-up/down. The drive strength is usually not specified, in which case the strengths defaults to **strong1** and **strong0**. Refer to *Cadence Verilog-XL Reference Manual* for more details on strengths.

D*elays*: If no delay is specified, then the gate has no propagation delay; if two delays are specified, the first represent the rise delay, the second the fall delay; if only one delay is specified, then rise and fall are equal. Delays are ignored in synthesis. This method of specifying delay is a special case of "Parameterized Modules" on page 11. The parameters for the primitive gates have been predefined as delays.

3.1. Basic Gates

These implement the basic logic gates. They have one output and one or more inputs. In the gate instantiation syntax shown below, GATE stands for one of the keywords **and, nand, or, nor, xor, xnor**.

Syntax

```
GATE (drive_strength) # (delays)
instance_name1(output, input_1,
                    input_2,..., input_N),
instance_name2(outp,in1, in2,..., inN);
Delays is
      #(rise, fall)   or
      # rise_and_fall    or
      #(rise_and_fall)
```

Example 3 .1

```
and c1 (o, a, b, c, d);      // 4-input AND called c1 and
    c2 (p, f g);             // a 2-input AND called c2.
or #(4, 3) ig (o, a, b);     /* or gate called ig (instance name);
                                rise time = 4, fall time = 3  */
xor #(5) xor1 (a, b, c);     // a = b XOR c after 5 time units
xor (pull1, strong0) #5 (a,b,c);  /* Identical gate with pull-up
      strength pull1 and pull-down strength strong0. */
```

3.2. buf, not Gates

These implement buffers and inverters, respectively. They have one input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword **buf** or **not**

Syntax

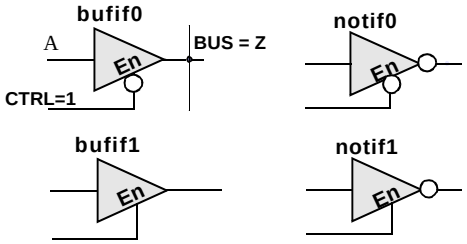
```
GATE (drive_strength) # (delays)
instance_name1(output_1, output_2,
                    ..., output_n, input),
instance_name2(out1, out2,  ..., outN, in);
```

Example 3 .2

```
not #(5) not_1 (a, c);       // a = NOT c after 5 time units
buf c1 (o, p, q, r, in);     // 5-output and 2-output buffers
    c2 (p, f g);
```

3.3. Three-State Gates; bufif1, bufif0, notif1, notif0

These implement 3-state buffers and inverters. They propagate z (3-state or high-impedance) if their control signal is deasserted. These can have three delay specifications: a rise time, a fall time, and a time to go into 3-state.



Example 3 .3

```
bufif0 #(5) not_1 (BUS, A, CTRL);   /* BUS =  A
                  5 time units after CTRL goes low. */
notif1 #(3,4,6) c1 (bus, a, b, cntr);     /* bus goes tri-state
                  6 time units after ctrl goes low. */
```

4. Data Types

4.1. Value Set

Verilog consists of only four basic values. Almost all Verilog data types store all these values:

0 (logic zero, or false condition)
1 (logic one, or true condition)
x (unknown logic value) x and z have limited use for synthesis.
z (high impedance state)

4.2. Wire

A **wire** represents a physical wire in a circuit and is used to connect gates or modules. The value of a **wire** can be read, but not assigned to, in a function or block. See "Functions" on p. 19, and "Procedures: Always and Initial Blocks" on p. 18. A **wire** does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module. Other specific types of wires include:

wand (wired-AND);:the value of a wand depend on logical AND of all the drivers connected to it.
wor (wired-OR);: the value of a wor depend on logical OR of all the drivers connected to it.
tri (three-state;): all drivers connected to a tri must be z, except one (which determines the value of the tri).

Syntax

wire [msb:lsb] wire_variable_list;
wand [msb:lsb] wand_variable_list;
wor [msb:lsb] wor_variable_list;
tri [msb:lsb] tri_variable_list;

Example 4 .1

```
wire c          // simple wire
wand d;
assign d = a;   // value of d is the logical AND of
assign d = b;   // a and b
wire [9:0] A;   // a cable (vector) of 10 wires.
```

4.3. Reg

A **reg** (register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks. See "Wire" on p. 4 above. A **reg** is a Verilog variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as *unsigned* numbers and no sign extension is done for what the user might have thought were two's complement numbers.

Syntax

reg [msb:lsb] reg_variable_list;

Example 4 .2

```
reg a;          // single 1-bit register variable
reg [7:0] tom;  // an 8-bit vector; a bank of 8 registers.
reg [5:0] b, c; // two 6-bit variables
```

4.4. Input, Output, Inout

These keywords declare input, output and bidirectional ports of a **module** or **task**. Input and inout ports are of type **wire**. An output port can be configured to be of type **wire**, **reg**, **wand**, **wor** or **tri**. The default is **wire**.

Syntax

input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
inout [msb:lsb] inout_port_list;

Example 4 .3

```
module sample(b, e, c, a);  //See "Module Instantiations" on p. 10
  input a;          // An input which defaults to wire.
  output b, e;      // Two outputs which default to wire
  output [1:0] c;   /* A two-it output. One must declare its
        type in a separate statement. */
  reg [1:0] c;      // The above c port is declared as reg.
```

4.5. Integer

Integers are general-purpose variables. For synthesois they are used mainly loops-indicies, parameters, and constants. See“Parameter” on p. 5. They are of implicitly of type **reg**. However they store data as signed numbers whereas explicitly declared **reg** types store them as unsigned. If they hold numbers which are not defined at compile time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.

Syntax

integer integer_variable_list;
... integer_constant ... ;

Example 4 .4

```
integer a;          // single 32-bit integer
assign b=63;        // 63 defaults to a 7-bit variable.
```

4.6. Supply0, Supply1

Supply0 and **supply1** define wires tied to logic 0 (ground) and logic 1 (power), respectively.

Syntax

supply0 logic_0_wires;
supply1 logic_1_wires;

Example 4 .5

```
supply0  my_gnd;     // equivalent to a wire assigned 0
supply1 a, b;
```

4.7. Time

Time is a 64-bit quantity that can be used in conjunction with the **$time** system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

Syntax

time time_variable_list;

Example 4 .6

```
time c;
c = $time;          // c = current simulation time
```

4.8. Parameter

A **parameter** defines a constant that can be set when you instantiate a **module**. This allows customization of a module during instantiation. See also “Parameterized Modules” on page 11.

Syntax

parameter par_1 = *value*,
par_2 = *value*, ...;
parameter [range] parm_3 = value

Example 4 .7

```
parameter  add = 2'b00, sub = 3'b111;
parameter n = 4;
        parameter n = 4;
parameter [3:0]  param2 = 4'b1010;
. . .
reg [n-1:0] harry;    /* A 4-bit register whose length is
                         set by parameter n above. */
always @(x)
   y = {{(add - sub){x}};  // The replication operator Sect. 5.8.
   if (x) begin
            state = param2[1]; else state = param2[2];
          end
```

5. Operators

5.1. Arithmetic Operators

These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.

Operators

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulus)

Example 5 .1

```
parameter n = 4;
reg[3:0] a, c, f, g, count;
f = a + c;
g = c - n;
 count = (count +1)%16;          //Can count 0 thru 15.
```

5.2. Relational Operators

Relational operators compare two operands and return a single bit 1or 0. These operators synthesize into comparators. Wire and reg variables are positive Thus (-3'b001) = = 3'b111 and (-3d001)>3d110. However for integers -1< 6.

Operators

< (less than)
<= (less than or equal to)
> (greater than)
>= (greater than or equal to)
== (equal to)
!= (not equal to)

Example 5 .2

```
if (x = = y)    e = 1;
else            e = 0;

// Compare in 2's compliment;  a>b
reg [3:0] a,b;
if (a[3]= = b[3])   a[2:0] > b[2:0];
else                b[3];
```

Equivalent Statement

```
e = (x == y);
```

5.3. Bit-wise Operators

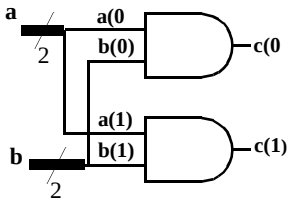
Bit-wise operators do a bit-by-bit comparison between two operands. However see"Reduction Operators" on p. 7.

Operators

~ (bitwise NOT)
& (bitwise AND)
| (bitwise OR)
^ (bitwise XOR)
~^ or ^~(bitwise XNOR)

Example 5 .3

```
module and2 (a, b, c);
  input [1:0] a, b;
  output [1:0] c;
  assign c = a & b;
endmodule
```



5.4. Logical Operators

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as "1". Logical operators are typically used in conditional (**if** ... **else**) statements since they work with expressions.

Operators

! (logical NOT)
&& (logical AND)
|| (logical OR)

Example 5 .4

```
wire[7:0] x, y, z;              // x, y and z are multibit variables.
reg a;
. . .
if ((x == y) && (z)) a = 1;  // a = 1 if x equals y, and z is nonzero.
  else  a = !x;                 // a =0 if x is anything but zero.
```

5.5. Reduction Operators

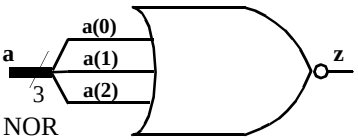
Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.

Operators

 (reduction AND)
| (reduction OR)
~ (reduction NAND)
~| (reduction NOR)
^ (reduction XOR)
~^ or ^~(reduction XNOR)

Example 5 .5

```
module chk_zero (a, z);
  input [2:0] a;
  output z;
  assign z = ~| a; // Reduction NOR
endmodule
```



5.6. Shift Operators

Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

Operators

<< (shift left)
>> (shift right)

Example 5 .6

```
assign c = a << 2;      /* c = a shifted left 2 bits;
                           vacant positions are filled with 0's */
```

5.7. Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

Operators

{ }(concatenation)

Example 5 .7

```
wire [1:0] a, b;    wire [2:0] x;      wire [3;0] y, Z;
assign x = {1'b0, a};  // x[2]=0,  x[1]=a[1],  x[0]=a[0]
assign y = {a, b};   /* y[3]=a[1],  y[2]=a[0],  y[1]=b[1],
y[0]=b[0] */

assign {cout, y} = x + Z;  // Concatenation of a result
```

5.8. Replication Operator

The replication operator makes multiple copies of an item.

Operators

{n{item}} (n fold replication of an item)

Example 5 .8

```
wire [1:0] a, b;    wire [4:0] x;
assign x = {2{1'b0}, a};  // Equivalent to  x = {0,0,a }
assign y = {2{a}, 3{b}};   //Equivalent to  y = {a,a,b,b}
```

For synthesis, Synopsis did not like a zero replication. For example:-

```
parameter n=5, m=5;
assign x= {(n-m){a}}
```

5.9. Conditional Operator: “?”

Conditional operator is like those in C/C++. They evaluate one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

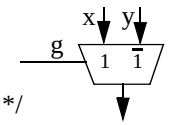
Operators

(cond) ? (result if cond true):
(result if cond false)

Example 5 .9

assign a = (g) ? x : y;

assign a = (inc = = 2) ? a+1 : a-1;
/* *if (inc), a = a+1,* else a = a-1 */



5.10. Operator Precedence

Table 6.1 shows the precedence of operators from highest to lowest. Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define order of precedence and improve the readability of your code.

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3’B101 and Y=3’B110, then X&Y=3’B100, X^Y=3’B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3’B101, 3’B110} = 6’B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
= =, !=	logical equality, logical inequality
= = =, != =	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&	logical AND. Treat all variables as False (zero) or True (nonzero). logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
||	
? :	conditional. x=(cond)? T : F;

Table 5.1: Verilog Operators Precedence

6. Operands

6.1. Literals

Literals are constant-valued operands that can be used in Verilog expressions. The two common Verilog literals are:
(a) String: A string literal is a one-dimensional array of characters enclosed in double quotes (“ “).
(b) Numeric: constant numbers specified in binary, octal, decimal or hexadecimal.

Number Syntax

n’Fddd..., where
n - integer representing number of bits
F - one of four possible base formats:
b (binary), **o** (octal), **d** (decimal),
h (hexadecimal). Default is **d**.
dddd - legal digits for the base format

Example 6 .1

```
“time is”// string literal
267     // 32-bit decimal number
2’b01 // 2-bit binary
20’hB36F// 20-bit hexadecimal number
‘o62    // 32-bit octal number
```

6.2. Wires, Regs, and Parameters

Wires, regs and parameters can also be used as operands in Verilog expressions. These data objects are described in more detail in Sect. 4. .

6.3. Bit-Selects “x[3]” and Part-Selects “x[5:3]”

Bit-selects and part-selects are a selection of a single bit and a group of bits, respectively, from a wire, reg or parameter vector using square brackets “[ ]”. Bit-selects and part-selects can be used as operands in expressions in much the same way that their parent data objects are used.

Syntax

variable_name[index]
variable_name[msb:lsb]

Example 6 .2

```
reg [7:0] a, b;
reg [3:0] ls;
reg c;
c = a[7] & b[7];        // bit-selects
ls = a[7:4] + b[3:0];   // part-selects
```

6.4. Function Calls

The return value of a function can be used directly in an expression without first assigning it to a register or wire variable. Simply place the function call as one of the operands. Make sure you know the bit width of the return value of the function call. Construction of functions is described in “Functions” on page 19

Syntax

function_name (argument_list)

Example 6 .3

```
assign a = b & c & chk_bc(c, b);// chk_bc is a function
. . ./*  Definition of the function  */
function  chk_bc;// function definition
  input c,b;
        chk_bc = b^c;
endfunction
```

7. Modules

7.1. Module Declaration

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specifies the i/o type (**input**, **output** or **inout**, see Sect. 4.4.) and width of each port. The default port width is 1 bit.

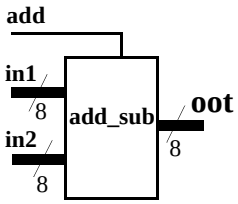
Then the port variables must be declared **wire**, **wand**,. . ., **reg** (See Sect. 4.). The default is **wire**. Typically inputs are **wire** since their data is latched outside the module. Outputs are type **reg** if their signals were stored inside an **always** or **initial** block (See Sect. 10.).

Syntax

module module_name (port_list);
  input [msb:lsb] input_port_list;
  output [msb:lsb] output_port_list;
  inout [msb:lsb] inout_port_list;
    ... *statements* ...
endmodule

Example 7 .1

```
module add_sub(add, in1, in2, oot);
  input add;              // defaults to wire
  input [7:0] in1, in2; wire in1, in2;
  output [7:0] oot;   reg oot;
        ... statements ...
endmodule
```



7.2. Continuous Assignment

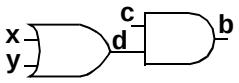
The continuous assignment is used to assign a value onto a wire in a module. It is the normal assignment outside of **always** or **initial** blocks (See Sect. 10.). Continuous assignment is done with an explicit **assign** statement or by assigning a value to a wire during its declaration. Note that continuous assignment statements are concurrent and are continuously executed during simulation. The order of assign statements does not matter. Any change in any of the right-hand-side inputs will immediately change a left-hand-side output.

Syntax

wire wire_variable = *value*;
assign wire_variable = *expression*;

Example 7 .2

```
wire [1:0] a = 2'b01;  // assigned on declaration
assign b = c & d;      // using assign statement
assign d = x | y;
/* The order of the assign statements
does not matter. */
```



7.3. Module Instantiations

Module declarations are templates from which one creates actual objects (instantiations). Modules are instantiated inside other modules, and each instantiation creates a unique object from the template. The exception is the top-level module which is its own instantiation.

The instantiated module's ports must be matched to those defined in the template. This is specified:
(i) by name, using a dot(.) " **.**template_port_name (name_of_wire_connected_to_port)".
or(ii) by position, placing the ports in exactly the same positions in the port lists of both the template and the instance.

Syntax for Instantiation

```
module_name
   instance_name_1 (port_connection_list),
   instance_name_2 (port_connection_list),
   ......
   instance_name_n (port_connection_list);
```

Example 7 .3

```
// MODULE DEFINITION

  module and4(a, b, c);
    input [3:0] a, b;
    output [3:0] c;
    assign c = a & b;
  endmodule

// MODULE INSTANTIATIONS
    wire [3:0] in1, in2;
    wire [3:0] o1, o2;
  /* C1 is an instance of module and4
     C1 ports referenced by position */
     and4   C1 (in1, in2, o1);
  /* C2 is another instance of and4.
  C2 ports are referenced to the
  declaration by name. */
     and4   C2 (.c(o2), .a(in1), .b(in2));
```

Modules may not be instantiated inside procedural blocks. See “Procedures: Always and Initial Blocks” on page 18.

7.4. Parameterized Modules

You can build modules that are parameterized and specify the value of the parameter at each instantiation of the module. See “Parameter” on page 5 for the use of parameters inside a module. Primitive gates have parameters which have been predefined as delays. See “Basic Gates” on page 3.

Syntax

```
module_name #(parameter_values)
   instance_name(port_connection_list);
```

Example 7 .4

```
// MODULE DEFINITION
  module shift_n (it, ot);        // used in module test_shift.
    input [7:0] it;    output [7:0] ot;
    parameter n = 2;‘              // default value of n is 2
    assign ot = (it << n);         // it  shifted left n times
  endmodule

//PARAMETERIZED INSTANTIATIONS
    wire [7:0] in1,  ot1, ot2, ot3;
    shift_n       shft2(in1,  ot1),      // shift by 2; default
    shift_n #(3) shft3(in1, ot2);   // shift by 3; override parameter 2.
    shift_n #(5) shft5(in1, ot3);   // shift by 5; override parameter 2.
```

Synthesis does not support the **defparam** keyword which is an alternate way of changing parameters.

c

8. Behavioral Modeling

Verilog has four levels of modelling:
1) The switch level which includes MOS transistors modelled as switches. This is not discussed here.
2) The gate level. See “Gate-Level Modelling” on p. 3
3) The Data-Flow level. See Example 7 .4 on page 11
4) The Behavioral or procedural level described below.
Verilog procedural statements are used to model a design at a higher level of abstraction than the other levels. They provide powerful ways of doing complex designs. However small changes n coding methods can cause large changes in the hardware generated. Procedural statements can only be used in procedures. Verilog procedures are described later in “Procedures: Always and Initial Blocks” on page 18,“Functions” on page 19, and “Tasks Not Synthesizable” on page 21.

8.1. Procedural Assignments

Procedural assignments are assignment statements used within Verilog procedures (**always** and **initial** blocks). Only **reg** variables and **integers** (and their bit/part-selects and concatenations) can be placed left of the “=” in procedures. The right hand side of the assignment is an expression which may use any of the operator types described in Sect. 5.

8.2. Delay in Assignment *(not for synthesis)*

In a *delayed assignment* Δt time units pass before the statement is executed and the left-hand assignment is made. With *intra-assignment delay,* the right side is evaluated immediately but there is a delay of Δt before the result is place in the left hand assignment. If another procedure changes a right-hand side signal during Δt, it does not effect the output. Delays are not supported by synthesis tools.

Syntax for Procedural Assignment
variable = expression
Delayed assignment
#Δt variable = expression;
Intra-assignment delay
variable = #Δt expression;

Example 8 .1

```
reg [6:0] sum;      reg h, ziltch;
sum[7] = b[7] ^ c[7]; // execute now.
ziltch = #15 ckz&h; /* ckz&a evaluated now; ziltch changed
                       after 15 time units. */
#10 hat = b&c;     /* 10 units after ziltch changes, b&c is
                       evaluated and hat changes. */
```

8.3. Blocking Assignments

Procedural (blocking) assignments (=) are done sequentially in the order the statements are written. A second assignment is not started until the preceding one is complete. See also Sect. 9.4.

Syntax

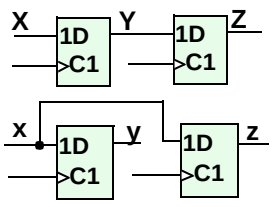
Blocking
variable = expression;
variable = #Δt expression;
grab inputs now, deliver ans. later.
#Δt variable = expression;
grab inputs later, deliver ans. later

Example 8 .2. For simulation

```
initial
  begin
     a=1;  b=2;  c=3;
  #5  a = b + c;      // wait for 5 units, and execute a= b + c =5.
     d = a;           // Time continues from last line, d=5 = b+c  at t=5.
```

Example 0 .1. For synthesis

```
always @( posedge clk)
  begin
     Z=Y;  Y=X; // shift register
     y=x;  z=y; //parallel ff.
```



8.4. Nonblocking (RTL) Assignments *(see below for synthesis)*

RTL (nonblocking) assignments (<=), which follow each other in the code, are done in parallel. The right hand side of nonblocking assignments is evaluated starting from the completion of the last blocking assignment or if none, the start of the procedure. The transfer to the left hand side is made according to the delays. A delay in a non-blocking statement will not delay the start of any subsequent statement blocking or non-blocking.
A good habit is to use "<=" if the same variable appears on both sides of the equal sign (Example 0 .1 on page 13).

For synthesis

- One must not mix "<=" or "=" in the same procedure.
- "<=" best mimics what physical flip-flops do; use it for "always @ (posedge clk ..) type procedures.
- "=" best corresponds to what c/c++ code would do; use it for combinational procedures.

Syntax

Non-Blocking

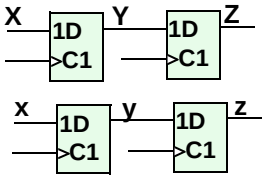
variable <= expression;
variable <= #Δ*t* expression;
#Δ*t* variable <= expression;

Example 0 .1. For simulation

```
initial
  begin
  #3   b <= a;        /* grab a at t=0 Deliver b at t=3.
  #6   x <= b + c;    // grab b+c  at t=0, wait and assign x  at  t=6.
                         x is unaffected by b's change. */
```

Example 0 .2. For synthesis

```
always @( posedge clk)
  begin
      Z<=Y;  Y<=X; // shift register
      y<=x;  z<=y; //also a shift register.
```



Example 8 .3. Use <= to transform a variable into itself.

```
reg G[7:0];
always @( posedge clk)
   G <= { G[6:0], G[7]}; // End around rotate 8-bit register.
```

The following example shows interactions between blocking and non-blocking for simulation. Do not mix the two types in one procedure for synthesis.

Syntax

Non-Blocking

variable <= expression;
variable <= #Δ*t* expression;
?#Δ*t* variable <=expression;

Blocking

variable = expression;
variable = #Δ*t* expression;
#Δ*t* variable = expression;

Example 8 .4 for simulation only

```
initial   begin
        a=1;  b=2;  c=3; x=4;
  #5   a = b + c;       // wait for 5 units, then grab b,c and execute a=2+3.
        d = a;          // Time continues from last line, d=5 = b+c  at t=5.
        x <= #6  b + c;// grab b+c  now at t=5, don't stop, make  x=5  at  t=11.
        b <= #2  a;     /* grab a at t=5 (end of last blocking statement).
           Deliver b=5 at t=7.  previous x is unaffected by b change. */
        y <= #1  b + c;// grab  b+c  at t=5, don't stop, make  x=5  at t=6.
  #3   z  =  b + c;     // grab b+c  at t=8 (#5+#3),  make  z=5  at t=8.
        w <= x          // make  w=4  at t=8. Starting at last blocking assignm.
```

8.5. begin ... end

begin ... end block statements are used to group several statements for use where one statement is syntactically allowed. Such places include functions, **always** and **initial** blocks, **if**, **case** and **for** statements. Blocks can optionally be named. See "disable" on page 15) and can include register, integer and parameter declarations.

Syntax

```
begin : block_name
   reg [msb:lsb] reg_variable_list;
   integer [msb:lsb] integer_list;
   parameter [msb:lsb] parameter_list;
            ... statements ...
end
```

Example 8 .5

```
function trivial_one; // The block name is "trivial_one."
   input a;
   begin: adder_blk;   // block named adder, with
      integer i;       // local integer i
      ... statements ...
   end
```

8.6. for Loops

Similar to for loops in C/C++, they are used to repeatedly execute a statement or block of statements. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```
for (count = value1;
      count </<=/>/>= value2;
      count = count +/- step)
   begin
   ... statements ...
   end
```

Example 8 .6

```
for (j = 0; j <= 7; j = j + 1)
   begin
      c[j] = a[j] & b[j];
      d[j] = a[j] | b[j];
   end
```

8.7. while Loops

The **while** loop repeatedly executes a statement or block of statements until the expression in the while statement evaluates to false. To avoid combinational feedback during synthesis, a while loop must be broken with an @(**posedge**/**negedge** clock) statement (Section 9.2). For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```
while (expression)
   begin
      ... statements ...
   end
```

Example 8 .7

```
while (!overflow) begin
   @(posedge clk);
   a = a + 1;
end
```

8.8. forever Loops

The forever statement executes an infinite loop of a statement or block of statements. To avoid combinational feedback during synthesis, a forever loop must be broken with an @(**posedge**/**negedge** clock) statement (Section 9.2). For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted. It is

Syntax

```
forever
   begin
      ... statements ...
   end
```

Example 8 .8

```
forever begin
   @(posedge clk);  // or use    a= #9 a+1;
   a = a + 1;
end
```

8.9. repeat Not Synthesizable

The repeat statement executes a statement or block of statements a fixed number of times.

Syntax

```
repeat (number_of_times)
   begin
      ... statements ...
   end
```

Example 8 .9

```
repeat (2) begin // after 50, a = 00,
   #50 a = 2'b00; // after 100, a = 01,
   #50 a = 2'b01; // after 150, a = 00,
end// after 200, a = 01
```

8.10. disable

Execution of a disable statement terminates a block and passes control to the next statement after the block. It is like the C *break* statement except it can terminate any loop, not just the one in which it appears.
Disable statements can only be used with named blocks.

Syntax

```
disable block_name;
```

Example 8 .10

```
begin: accumulate
 forever
   begin
      @(posedge clk);
      a = a + 1;
      if (a == 2'b0111) disable accumulate;
   end
end
```

8.11. if ... else if ... else

The **if** ... **else i**f ... **else** statements execute a statement or block of statements depending on the result of the expression following the if. If the conditional expressions in all the if's evaluate to false, then the statements in the else block, if present, are executed.
There can be as many **else if** statements as required, but only one **if** block and one **else** block. If there is one statement in a block, then the **begin** .. **end** statements may be omitted.
Both the **else if** and **else** statements are optional. However if all possibilities are not specifically covered, synthesis will generated extra latches.

Syntax

```
if (expression)
   begin
      ... statements ...
   end
else if (expression)
   begin
      ... statements ...
   end
... more else if blocks ...
else
   begin
      ... statements ...
   end
```

Example 8 .11

```
if (alu_func == 2'b00)
   aluout = a + b;
else if (alu_func == 2'b01)
   aluout = a - b;
else if (alu_func == 2'b10)
   aluout = a & b;
else // alu_func == 2'b11
   aluout = a | b;

if (a == b)          // This if with no else will generate
   begin             // a latch for x and ot. This is so they
      x = 1;         // will hold there old value if (a != b).
      ot = 4'b1111;
   end
```

8.12. case

The **case** statement allows a multipath branch based on comparing the *expression* with a list of *case choices.*
Statements in the **default** block executes when none of the *case choice* comparisons are true (similar to the else block in the if ... else if ... else). If no comparisons , including delault, are true, synthesizers will generate unwanted latches. Good practice says to make a habit of puting in a default whether you need it or not.
If the defaults are dont cares, define them as 'x' and the logic minimizer will treat them as don't cares.
Case choices may be a simple constant or expression, or a comma-separated list of same.

Syntax

```
case (expression)
   case_choice1:
      begin
         ... statements ...
      end
   case_choice2:
      begin
         ... statements ...
      end
... more case choices blocks ...
   default:
      begin
         ... statements ...
      end
endcase
```

Example 0 .1

```
case (alu_ctr)
   2'b00:  aluout = a + b;
   2'b01:  aluout = a - b;
   2'b10:  aluout = a & b;
   default: aluout = 1'bx; // Treated as don't cares for
endcase                    // minimum logic generation.
```

Example 0 .2

```
case (x, y, z)
   2'b00:  aluout = a + b; //case if x or y or z is 2'b00.
   2'b01:  aluout = a - b;
   2'b10:  aluout = a & b;
   default: aluout = a | b;
endcase
```

8.13. casex

In **casex**(a) the case choices constant "a" may contain z, x or ? which are used as don't cares for comparison. With **case** the corresponding simulation variable would have to match a tri-state, unknown, or either signal. In short, **case** uses x to compare with an unknown signal. **Casex** uses x as a don't care which can be used to minimize logic.

Syntax

same as for **case** statement (Section 8.10)

Example 8 .12

```
casex (a)
   2'b1x: msb = 1;      //  msb = 1 if a = 10 or a = 11
      // If this were case(a) then only a=1x  would match.
   default: msb = 0;
endcase
```

8.14. casez

Casez is the same as casex except only ? and z (not x) are used in the case choice constants as don't cares. **Casez** is favored over **casex** since in simulation, an inadvertent x signal, will not be matched by a 0 or 1 in the case choice.

Syntax

same as for **case** statement (Section 8.10)

Example 8 .13

```
casez (d)
   3'b1??: b = 2'b11;  //  b = 11 if d = 100 or greater
   3'b01?: b = 2'b10;  //  b = 10 if d = 010 or 011
   default: b = 2'b00;
endcase
```

9. Timing Controls

9.1. Delay Control Not Synthesizable

This specifies the delay time units before a statement is executed during simulation. A delay time of zero can also be specified to force the statement to the end of the list of statements to be evaluated at the current simulation time.

Syntax

#*delay* statement;

Example 9 .1

```
#5 a = b + c;       // evaluated and assigned after 5 time units
#0 a = b + c;       // very last statement to be evaluated
```

9.2. Event Control, @

This causes a statement or **begin-end** block to be executed only after specified events occur. An event is a change in a variable. and the change may be: a positive edge, a negative edge, or either (a level change), and is specified by the keyword **posedge**, **negedge**, or no keyword respectively. Several events can be combined with the **or** keyword. Event specification begins with the character **@**and are usually used in **always** statements. See page 18.

For synthesis one cannot combine level and edge changes in the same list.
For flip-flop and register synthesis the standard list contains only a clock and an optional reset.
For synthesis to give combinational logic, the list must specify only level changes and must contain all the variables appearing in the right-hand-side of statements in the block.

Syntax

@ (**posedge** variable **or**
negedge variable) statement;

@ (variable **or** variable . . .) statement;

Example 9 .2

```
always
@(posedge clk or negedge rst)
        if (rst) Q=0; else Q=D;  // Definition for a D flip-flop.

@(a or b or e);             // re-evaluate if a or b or e changes.
  sum = a + b + e;  // Will synthesize to a combinational adder.
```

9.3. Wait Statement Not Synthesizable

The **wait** statement makes the simulator wait to execute the statement(s) following the wait until the specified condition evaluates to true. Not supported for synthesis.

Syntax

wait (condition_expression) statement;

Example 9 .3

```
  wait (!c) a = b;    // wait until c=0, then assign b to a
```

9.4. Intra-Assignment Delay Not Synthesizable

This delay #Δ is placed after the equal sign. The left-hand assignment is delayed by the specified time units, but the right-hand side of the assignment is evaluated before the delay instead of after the delay. This is important when a variable may be changed in a concurrent procedure. See also “Delay in Assignment (not for synthesis)” on page 12.

Syntax

variable = #Δ*t* expression;

Example 9 .4

```
assign a=1; assign b=0;
  always @(posedge clk)
      b = #5 a;             // a = b after 5 time units.
  always @(posedge clk)
      c = #5 b;             /* b was grabbed in this parallel proce-
dure before the first procedure changed it. */
```

10. Procedures: Always and Initial Blocks

10.1. Always Block

The always block is the primary construct in RTL modeling. Like the continuous assignment, it is a concurrent statement that is continuously executed during simulation. This also means that all always blocks in a module execute simultaneously. This is very unlike conventional programming languages, in which all statements execute sequentially. The always block can be used to imply latches, flip-flops or combinational logic. If the statements in the always block are enclosed within **begin ... end**, the statements are executed sequentially. If enclosed within the **fork ... join**, they are executed concurrently (simulation only).

The always block is triggered to execute by the level, positive edge or negative edge of one or more signals (separate signals by the keyword **or**). A double-edge trigger is implied if you include a signal in the event list of the always statement. The single edge-triggers are specified by **posedge** and **negedge** keywords.

Procedures can be named. In simulation one can **disable** named blocks. For synthesis it is mainly used as a comment.

Syntax 1

```
always @(event_1 or event_2 or ...)
  begin
    ... statements ...
  end
```

Example 10 .1

```
always @(a or b) // level-triggered; if a or b changes levels
always @(posedge clk);  // edge-triggered: on +ve edge of clk
```

see previous sections for complete examples

Syntax 2

```
always @(event_1 or event_2 or ...)
  begin: name_for_block
    ... statements ...
  end
```

10.2. Initial Block

The initial block is like the always block except that it is executed only once at the beginning of the simulation. It is typically used to initialize variables and specify signal waveforms during simulation. Initial blocks are not supported for synthesis.

Syntax

```
initial
  begin
    ... statements ...
  end
```

Example 10 .2

```
inital
 begin
   clr = 0;        // variables initialized at
   clk = 1;        // beginning of the simulation
 end

inital              // specify simulation waveforms
 begin
   a = 2'b00;      // at time = 0, a = 00
   #50 a = 2'b01;  // at time = 50, a = 01
   #50 a = 2'b10;  // at time = 100, a = 10
 end
```

11. Functions

Functions are declared within a module, and can be called from continuous assignments, always blocks or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked.
Functions describe combinational logic, and by do not generate latches. Thus an if without an else will simulate as though it had a latch but synthesize without one. This is a particularly bad case of synthesis not following the simulation. It is a good idea to code functions so they would not generate latches if the code were used in a procedure.
Functions are a good way to reuse procedural code, since modules cannot be invoked from a procedure.

11.1. Function Declaration

A function declaration specifies the name of the function, the width of the function return value, the function input arguments, the variables (reg) used within the function, and the function local parameters and integers.

Syntax, Function Declaration

```
function [msb:lsb] function_name;
   input [msb:lsb] input_arguments;
   reg [msb:lsb] reg_variable_list;
   parameter [msb:lsb] parameter_list;
   integer [msb:lsb] integer_list;
            ... statements ...
endfunction
```

Example 11 .1

```
function [7:0] my_func;   // function return 8-bit value
   input [7:0] i;
   reg [4:0] temp;
   integer n;
     temp= i[7:4] | ( i[3:0]);
     my_func = {temp, i[[1:0]};
endfunction
```

11.2. Function Return Value

When you declare a function, a variable is also implicitly declared with the same name as the function name, and with the width specified for the function name (The default width is 1-bit). This variable is “my_func” in Example 11 .1 on page 19. At least one statement in the function must assign the function return value to this variable.

11.3. Function Call

As mentioned in Sect. 6.4. , a function call is an operand in an expression. A function call must specify in its terminal list all the input parameters.

11.4. Function Rules

The following are some of the general rules for functions:
- Functions must contain at least one input argument.
- Functions cannot contain an inout or output declaration.
- Functions cannot contain time controlled statements (#, @, wait).
- Functions cannot enable tasks.
- Functions must contain a statement that assigns the return value to the implicit function name register.

11.5. Function Example

A Function has only one output. If more than one return value is required, the outputs should be concatenated into one vector before assigning it to the function name. The calling module program can then extract (unbundle) the individual outputs from the concatenated form. Example 11.2 shows how this is done, and also illustrates the general use and syntax of functions in Verilog modeling.

Syntax

function_name = expression

Example 11 .2

```
module simple_processor (instruction, outp);
 input [31:0] instruction;
 output [7:0] outp;
 reg [7:0] outp;;    // so it can be assigned in always block
 reg func;
 reg [7:0] opr1, opr2;

 function [16:0] decode_add (instr) // returns 1 1-bit plus 2 8-bits
   input [31:0] instr;
   reg add_func;
   reg [7:0] opcode, opr1, opr2;
   begin
     opcode = instr[31:24];
     opr1 = instr[7:0];
     case (opcode)
       8'b10001000: begin          // add two operands
         add_func = 1;
         opr2 = instr[15:8];
       end
       8'b10001001: begin          // subtract two operands
         add_func = 0;
         opr2 = instr[15:8];
       end
       8'b10001010: begin          // increment operand
         add_func = 1;
         opr2 = 8'b00000001;
       end
       default: begin;             // decrement operand
         add_func = 0;
         opr2 = 8'b00000001;
       end
     endcase
     decode_add = {add_func, opr2, opr1}; // concatenated into 17-bits
   end
 endfunction
// ----------------------------------------    ---------------------------------
 always @(instruction) begin
   {func, op2, op1} = decode_add (instruction); // outputs unbundled
   if (func == 1)
     outp = op1 + op2;
   else
     outp = op1 - op2;
 end
endmodule
```

12. Tasks Not Synthesizable

A task is similar to a function, but unlike a function it has both input and output ports. Therefore tasks do not return values. Tasks are similar to procedures in most programming languages. The syntax and statements allowed in tasks are those specified for functions (Sections 11).

Syntax

```
task task_name;
   input [msb:lsb] input_port_list;
   output [msb:lsb] output_port_list;
   reg [msb:lsb] reg_variable_list;
   parameter [msb:lsb] parameter_list;
   integer [msb:lsb] integer_list;
            ... statements ...
endtask
```

Example 12 .1

```
module alu (func, a, b, c);
 input [1:0] func;
 input [3:0] a, b;
 output [3:0] c;
 reg [3:0] c;          // so it can be assigned in always block

 task my_and;
   input[3:0] a, b;
   output [3:0] andout;
   integer i;
   begin
      for (i = 3; i >= 0; i = i - 1)
         andout[i] = a[i] & b[i];
   end
 endtask

 always @(func or a or b) begin
   case (func)
      2'b00: my_and (a, b, c);
      2'b01: c = a | b;
      2'b10: c = a - b;
      default: c = a + b;
   endcase
 end
endmodule
```

13. Component Inference

13.1. Latches

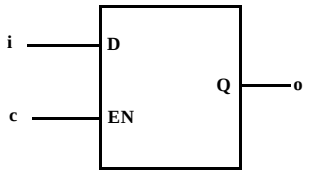
A latch is inferred (put into the synthesized circuit) if a variable is not assigned to in the else branch of an if ... else if ... else statement. A latch is also inferred in a case statement if a variable is assigned to in only some of the possible case choice branches. Assigning a variable in the default branch avoids the latch. In general, a latch is inferred in if ... else if ... else and case statements if a variable, or one of its bits, is only assigned to in only some of the possible branches.
To improve code readability, use the if statement to synthesize a latch because it is difficult to explicitly specify the latch enable signal when using the case statement.

Syntax

See Sections 8.9 and 8.10 for
if ... else if ... else and **case** statements

Example 13 .1

```
always @(c, i);
  begin;
    if (c == 1)
      o = i;
  end
```



13.1. Edge-Triggered Registers, Flip-flops, Counters

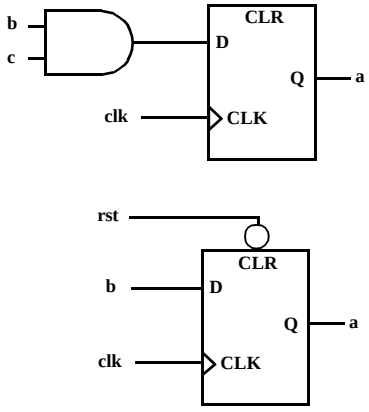
A register (flip-flop) is inferred by using posedge or negedge clause for the clock in the event list of an always block. To add an asynchronous reset, include a second posedge/negedge for the reset and use the if (reset) ... else statement. Note that when you use the negedge for the reset (active low reset), the if condition is (**!**reset).

Syntax

```
always @(posedge clk or
            posedge reset_1 or
            negedge reset_2)
  begin
    if (reset_1)  begin
                  ... reset assignments
                  end
    else if (!reset_2) begin
                  ... reset assignments
                  end
    else          begin
                  ...register assignments
                  end
  end
```

Example 0 .1

```
always @(posedge clk);
  begin;
    a <= b & c;
  end

always @(posedge clk or
            negedge rst);
  begin;
     if (! rst)  a< = 0;
     else        a <= b;
  end
```



Example 0 .2 An Enabled Counter

```
reg [7:0] count;
wire enable;
  always @(posedge clk or posedge rst) // Do not include enable.
    begin;
      if (rst) count<=0;
      else if  (enable) count <= count+1;
    end;                    // 8 flip-flops will be generated.
```

13.2. Multiplexers

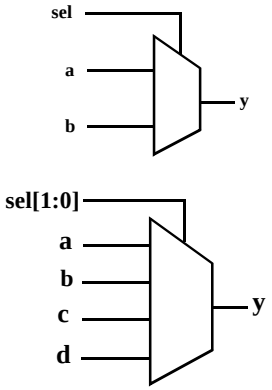
A multiplexer is inferred by assigning a variable to different variables/values in each branch of an if or case statement. You can avoid specifying each and every possible branch by using the else and default branches. Note that a latch will be inferred if a variable is not assigned to for all the possible branch conditions.
To improve readability of your code, use the case statement to model large multiplexers.

Syntax

See Sections 8.9 and 8.10 for
if ... else if ... else and **case** statements

Example 13 .2

```
if (sel == 1)
   y = a;
else
   y = b;

case (sel)
   2'b00: y = a;
   2'b01: y = b;
   2'b10: y = c;
   default: y = d;
endcase
```



13.3. Adders/Subtracters

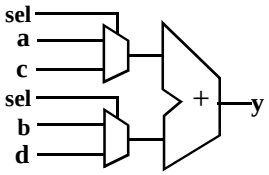
The +/- operators infer an adder/subtracter whose width depend on the width of the larger operand.

Syntax

See Section 7 for operators

Example 13 .3

```
if (sel == 1)
   y = a + b;
else
   y = c +  d;
```



13.4. Tri-State Buffers

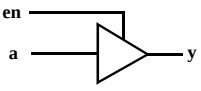
A tristate buffer is inferred if a variable is conditionally assigned a value of z using an if, case or conditional operator.

Syntax

See Sections 8.9 and 8.10 for
if ... else if ... else and **case** statements

Example 13.5

```
if (en == 1)
   y = a;
else
   y = 1'bz;
```



13.5. Other Component Inferences

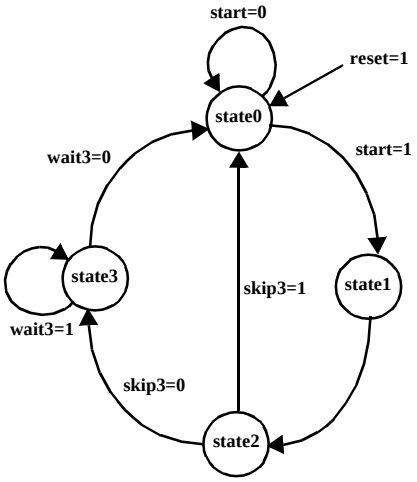
Most logic gates are inferred by the use of their corresponding operators. Alternatively a gate or component may be explicitly instantiated by using the primitive gates (**and, or, nor, inv** ...) provided in the Verilog language.

14. Finite State Machines. For synthesis

When modeling finite state machines, it is recommended to separate the sequential current-state logic from the combinational next-state and output logic.

State Diagram

for lack of space the outputs are not shown on the state diagram, but are:
in state0: Zot = 000,
in state1: Zot = 101,
in state2: Zot = 111,
in state3: Zot = 001.



Using Macros for state definition
As an alternative for-
parameter state0=0, state1=1,
state2=2, state3=3;
one can use macros. For example after the definition below 2'd0 will be textually substituted whenever `state0 is used.

```
`define state0 2'd0
`define state1 2'd1
`define state2 2'd
`define state3 2'd3;
```

When using macro definitions one must put a back quote in front. For example:

```
case (state)
      `state0:   Zot = 3'b000;
      `state1:   Zot = 3'b101;
      `state2:   Zot = 3'b111;
      `state3:   Zot = 3'b001;
```

Example 14 .1

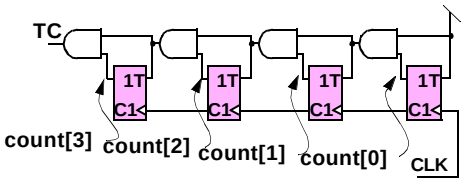
```
module my_fsm (clk, rst, start, skip3, wait3, Zot);
 input clk, rst, start, skip3, wait3;
 output [2:0] Zot;  // Zot is declared reg so that it can
 reg [2:0] Zot;     // be assigned in an always block.
 parameter state0=0, state1=1, state2=2, state3=3;
 reg [1:0] state, nxt_st;

 always @ (state or start or skip3 or wait3)
   begin : next_state_logic //Name of always procedure.
     case (state)
       state0: begin
                    if (start) nxt_st = state1;
                    else nxt_st = state0;
                    end
       state1: begin
                    nxt_st = state2;
                    end
       state2: begin
                    if (skip3) nxt_st = state0;
                    else nxt_st = state3;
                    end
       state3: begin
                    if (wait3) nxt_st = state3;
                    else nxt_st = state0;
                    end
       default: nxt_st = state0;
     endcase       // default is optional since all 4 cases are
   end             // covered specifically. Good practice says uses it.

 always @(posedge clk or posedge rst)
   begin : register_generation
     if (rst) state = state0;
     else state    = nxt_st;
   end

 always @(state) begin : output_logic
   case (state)
     state0:   Zot = 3'b000;
     state1:   Zot = 3'b101;
     state2:   Zot = 3'b111;
     state3:   Zot = 3'b001;
     default: Zot = 3'b000;// default avoids latches
   endcase
 end
endmodule
```

14.1.
14.2. Counters

Counters are a simple type of finite-state machine where separation of the flip-flop generation code and the next-state generation code is not worth the effort. In such code, use the nonblocking "<=" assignment operator.

Binary Counter

Using toggle flip-flops



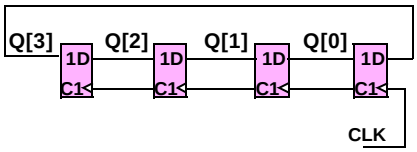
Example 14 .2

```
reg  [3:0]  count;   wire TC;  // Terminal count (Carry out)
  always @(posedge clk or posedge rset)
   begin
     if (rset) count <= 0;
     else count <= count+1;
   end
assign TC = & count;    // See "Reduction Operators" on page 7
```

14.3. Shift Registers

Shift registers are also best done completely in the flip-flop generation code. Use the nonblocking "<=" assignment operator so the operators "<< N" shifts left N bits. The operator ">>N" shifts right N bits. See also Example 8 .3 on page 13.

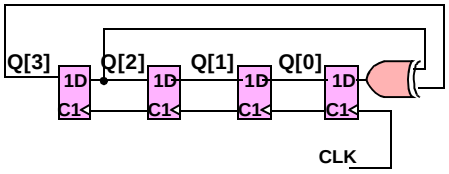
Shift Register



Example 14 .3

```
reg  [3:0]  Q;
  always @(posedge clk or posedge rset)
   begin
     if (rset) Q <= 0;
     else     begin
         Q <=Q << 1; // Left shift 1 position
        Q[0] <= Q[3]; /* Nonblocking means the old Q[3] is sent
to Q[0]. Not the revised Q[3] from the previous line.
   end
```

Linear-Feedback Shift Register



Example 14 .4

```
reg  [3:0]  Q;
  always @(posedge clk or posedge rset)
   begin
     if (rset) Q <= 0;
     else     begin
         Q <= {Q[2:1]: Q[3]^Q[2]; /* The concatenation operators
      "{...}" form the new Q from elements of the old Q. */
     end
   end
```

15. Compiler Directives

Compiler directives are special commands, beginning with ‘, that affect the operation of the Verilog simulator. The Synopsys Verilog HDL Compiler/Design Compiler and many other synthesis tools parse and ignore compiler directives, and hence can be included even in synthesizable models. Refer to *Cadence Verilog-XL Reference Manual* for a complete listing of these directives. A few are briefly described here.

15.1. Time Scale

`timescale specifies the time unit and time precision. A time unit of 10 ns means a time expressed as say #2.3 will have a delay of 23.0 ns. Time precision specifies how delay values are to be rounded off during simulation. Valid time units include s, ms, µs, ns, ps, fs.
Only 1, 10 or 100 are valid integers for specifying time units or precision. It also determines the displayed time units in display commands like **$display**

Syntax

`timescale time_unit / time_precision;

Example 15 .1

```
`timescale 1 ns/1 ps     // unit =1ns, precision=1/1000ns
`timescale 1 ns /100 ps   // time unit = 1ns; precision = 1/10ns;
```

15.2. Macro Definitions

A macro is an identifier that represents a string of text. Macros are defined with the directive **`define**, and are invoked with the quoted macro name as shown in the example.

Syntax

`define macro_name text_string;
. . . `macro_name . . .

Example 15 .2

```
`define add_lsb a[7:0] + b[7:0]
assign 0 = 'add_lsb;   // assign o = a[7:0] + b[7:0];
```

15.3. Include Directive

Include is used to include the contents of a text file at the point in the current file where the include directive is. The include directive is similar to the C/C++ include directive.

Syntax

`include file_name;

Example 15 .3

```
module x;
  'include “dclr.v”; // contents of file “dclr,v” are put here
```

16. System Tasks and Functions

These are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign (**$**). The Synopsys Verilog HDL Compiler/Design Compiler and many other synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models. Refer to *Cadence Verilog-XL Reference Manual* for a complete listing of system functions. A few are briefly described here.
System tasks that extract data, like **$monitor** need to be in an **initia**l or **always** block.

16.1. $display, $strobe, $monitor

These commands have the same syntax, and display their values as text on the screen during simulation. They are much less convenient than waveform display tools like *cwaves*® or *Signalscan*®. $**display** and $**strobe** display once every time they are executed, whereas $**monitor** displays every time one of its parameters changes. The difference between $**display** and $**strobe** is that $**strobe** displays the parameters at the very end of the current simulation time unit. The format string is like that in C/C++, and may contain format characters. Format characters include %**d** (decimal), %**h** (hexadecimal), %**b** (binary), %**c** (character), %**s** (string) and %**t** (time). Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax

$display (“format_string”,
par_1, par_2, ...);

Example 16 .1

```
 initial  begin
$displayh (b, d);  // displayed in hexadecimal
$monitor (“at time=%t, d=%h”, $time, a);
        end
```

16.2. $time, $stime, $realtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively. Their use is illustrated in Examples 4.6 and 13.1.

16.3. $reset, $stop, $finish

$reset resets the simulation back to time 0; **$stop** halts the simulator and puts it in the interactive mode where the user can enter commands; **$finish** exits the simulator back to the operating system.

16.4. $deposit

$deposit sets a net to a particular value.

Syntax

$deposit (net_name, *value*);

Example 16 .2

```
$deposit (b, 1’b0);
$deposit (outp, 4’b001x);// outp is a 4-bit bus
```

16.5. $scope, $showscope

$scope**(hierarchy_name) sets the current hierarchical scope to hierarchy_name. **$showscopes(n) lists all modules, tasks and block names in (and below, if n is set to 1) the current scope.

16.6. $list

$list (hierarchical_name) lists line-numbered source code of the named module, task, function or named-block.

16.7. $random

$random generates a random integer every time it is called. If the sequence is to be repeatable, the first time one invokes random give it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

Syntax

xzz = **$random**[(integer)];

Example 16 .3

```
reg [3:0] xyz;
initial  begin
    xyz= $random (7);  // Seed the generator so number
     // sequence will repeat if simulation is restarted.
    forever xyz = #20 $random;
   // The 4 lsb bits of the random integers will transfer into the
   // xyz. Thus xyz will be a random integer  0 ≤ xyz ≤ 15.
```

16.8. $dumpfile, $dumpvar, $dumpon, $dumpoff, $dumpall

These can dump variable changes to a simulation viewer like ***cwaves***®. The dump files are capable of dumping all the variables in a simulation. This is convenient for debugging, but can be very slow.

Syntax

$dumpfile(“filename.dmp”)
$dumpvar dumps all variables in the design.
$dumpvar(1, top**)** dumps all the variables in module top and below, but not modules instantiated in top.
$dumpvar(2, top**)** dumps all the variables in module top and 1 level below.
$dumpvar(n, top**)** dumps all the variables in module top and n-1 levels below.
$dumpvar(0, top**)** dumps all the variables in module top and all level below.
$dumpon initiates the dump.
$dumpoff stop dumping.

Example 16 .4

```
//  Test Bench
module testbench:
 reg a, b;   wire c;
  initial begin;
     $dumpfile(“cwave_data.dmp”);
     $dumpvar  //Dump all the variables
// Alternately instead of $dumpvar, one could use
     $dumpvar(1, top)  //Dump variables in the top module.
// Ready to turn on the dump.
     $dumpon
          a=1; b=0;
          topmodule top(a, b, c);
  end
```

16.9. $shm_probe, $shm_open

These are special commands for the *Simulation History Manager* for Cadence *cwaves*® only. They will save variable changes for later display.

Syntax

$shm_open (“cwave_dump.dm”)
$shm_probe (var1,var2, var3);
/* Dump all changes in the above 3 variables. */
$shm_probe(a, b, inst1.var1, inst1.var2);
/* Use the qualifier inst1. to look inside the hierarchy. Here inside module instance “inst1” the variables var1 and var2 will be dumped.*/

Example 16 .5

```
//  Test Bench
module testbench:
 reg a, b;   wire c;
  initial begin;
     $shm_open(“cwave_data.dmp”);
     $shm_probe(a, b, c)

/* See also the testbench example in “Test Benches” on p. 29
```

17. Test Benches

A test bench supplies the signals and dumps the outputs to simulate a Verilog design (module(s)). It invokes the design under test, generates the simulation input vectors, and implements the system tasks to view/format the results of the simulation. It is never synthesized so it can use all Verilog commands.

To view the waveforms when using Cadence Verilog XL Simulator, use the Cadence-specific Simulation History Manager (SHM) tasks of $**shm_open** to open the file to store the waveforms, and $**shm_probe** to specify the variables to be included in the waveforms list. You can then use the Cadence **cwaves** waveform viewer by typing **cwaves &** at the UNIX prompt.

Syntax

$shm_open(filename);
$shm_probe(var1, var2, ...)

Note also

var=$**random**
wait(condition) statement

Example 17 .1

```
‘timescale 1 ns /100 ps   // time unit = 1ns; precision = 1/10 ns;
module my_fsm_tb; // Test Bench of FSM Design of Example 14.1
  /* ports of the design under test are variables in the test bench */
  reg clk, rst, start, skip3, wait3;
  wire  Button;

     /**** DESIGN TO SIMULATE (my_fsm) INSTANTIATION ****/
  my_fsm    dut1 (clk, rst, start, skip3, wait3, Button);

       /**** SECTION TO DISPLAY VARIABLES ****/
  initial begin
    $shm_open(“sim.db”);     //Open the SHM database file
    /* Specify the variables to be included  in the waveforms to be
        viewed by Cadence cwaves */
    $shm_probe(clk, reset, start);
     // Use the qualifier dut1. to look at variables inside the instance dut1.
    $shm_probe(skip3, wait3, Button, dut1.state, dut1.nxt_st);
  end

       /****   RESET   AND    CLOCK    SECTION  ****/
  initial begin
    clk = 0;   rst=0;
    #1  rst = 1;     // The delay gives rst a posedge for sure.
    #200 rst = 0;  // Deactivate reset after two clock cycles +1 ns*/
  end
  always #50 clk = ~clk;  // 10 MHz clock (50*1 ns*2) with 50% duty-cycle

   /****  SPECIFY THE INPUT WAVEFORMS skip3 & wait3  ****/
  initial begin
    skip3 = 0;  wait3 = 0;   // at time 0, wait3=0, skip3=0
    #1;                 // Delay to keep inputs from changing on clock edge.
    #600 skip3 = 1; // at time 601, wait3=0, skip3=1
    #400 wait3 = 1; // at time 1001, wait3=1, skip3=0
    skip3= 0;
    #400 skip3 = 1; // at time 1401, wait3=1, skip3=1
    wait(Button)  skip3 = 0; // Wait until Button=1, then make skip3 zero.
    wait3 = $random;  //Generate a random number, transfer lsb into wait3
    $finish;            // stop simulation. Without this it will not stop.
  end
endmodule
```

17.1. Synchronous Test Bench

In synchronous designs, one changes the data during certain clock cycles. In the previous test bench one had to keep counting delays to be sure the data came in the right cycle. With a synchronous test bench the input data is stored in a vector or array and one part injected in each clock cycle. The Verilog array is not defined in these notes. Synchronous test benches are essential for cycle based simulators which do not use any delays smaller than a clock cycle.

Things to note:

data[8:1]=8'b1010_1101;
The underscore visually separates the bits. It acts like a comment.

if (I==9) **$finish**;
When the data is used up, finish

x<=data[I]***;*** I<=I+1;
When synthesizing to flip-flops as in an In an @(posedge... procedure, always use nonblocking. Without that you will be racing with the flip-flops in the other modules.

Example 17 .2

```
// Synchronous test bench
module SynchTstBch:
  reg [8:1] data;
  reg x,clk;
  integer I;

  initial begin
      data[8:1]=8'b1010_1101; // Underscore spaces bits.
      I=1;
      x=0;
      clk=0;
      forever #5 clk=~clk;
 end
  /*** Send in a new value of x every clock cycle***/
  always @(posedge clk)
       begin
           if (I==9) $finish;
      #1;      // Keeps data from changing on clock edge.
      x<=data[I];
      I<=I+1;
    end
  topmod top1(clk, x);

endmodule
```

APPENDIX F
REFERENCE MANUALS

The Digilent Basys Board Reference Manual is available at
http://www.digilentinc.com/data/products/basys/basys_e_rm.pdf

The Digilent Basys2 Board Reference Manual is available at
http://www.digilentinc.com/Data/Products/BASYS2/Basys2_rm.pdf