ECE 385

DIGITAL SYSTEMS LABORATORY

LABORATORY MANUAL
Version 20.0

By

Janak Patel
Rakesh Kumar
Deming Chen
Zuofu Cheng

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 385

DIGITAL SYSTEMS LABORATORY

LABORATORY MANUAL
Version 20.0

By

Janak Patel
Rakesh Kumar
Deming Chen
Zuofu Cheng

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Acknowledgements

Lab 1, 2, 3, 5, 6 were originally developed by Professors Janak Patel and Rakesh Kumar and modified by Professor Deming Chen. Original versions of the lab manual were prepared by Huan-Ting (Joe) Meng. Lab 7, SoC materials, and document maintenance was prepared by Professor Zuofu Cheng. New materials in Lab 8 and the Introduction to NIOS II and Qsys in Lab 8 and 9 were prepared by Ying-Yu (Christine) Chen, Benjamin Delay, and Bilal Gabula. Work on Lab 9 Avalon-MM components were done by Edward Wang. Additional copyediting and improvements were made by Po-Han Huang and Yikuan Chen, as well as contributions from Mihir Iyer.

Table of Contents

University of Illinois ECE 385

DIGITAL SYSTEMS LABORATORY
INTRODUCTION

COURSE GOALS:

ECE 385 is a digital system design laboratory providing practical design experience and extending the design methods developed in the lecture course ECE 120/220. The laboratory exercises are developed to give students ability to design, build, and debug digital systems. The first three labs use standard TTL chips, wires and a proto board. The rest of the course uses CAD tools to synthesize logic described in SystemVerilog, which is then mapped to an FPGA.

COURSE EQUIPMENT:

Due to equipment limitations and to encourage cooperative work and discussion, students will be paired up for in-laboratory work. A standard lab kit is checked out to each lab pair, and they are jointly responsible for its contents. The lab kit may be taken home or stored in the lockers in the laboratory between lab periods. Lab kits must be checked in by each pair at the end of the semester. You will be charged for any missing components beyond those lost by normal wear and tear.

COURSE CONTENT:

The goal of this 385 course is to teach students design, build, and test/debug a digital system, which can be a 16-bit microprocessor, a dedicated logic core, or a simple system-on-a-chip (SoC). Students will learn about the modular design approach, which is the underlining principle of IP-based SoC design methodology. Students will start with TTL logic in the first three weeks to strengthen their skills of using physical logic elements to build relatively complex circuits such as a 4-bit serial processor. As the circuits will being built on the protoboard will quickly become more complex as compared to what they experienced in their previous semesters, the course will make a transition to SystemVerilog and RTL design methodology. The concept of modular design is carried over to SystemVerilog so students would have a concrete understanding of the connection between wired circuits on the protoboard and mapped circuits on the FPGA board, and the RTL design abstraction can be better captured through this process. Students will then start to gradually learn how to design and realize digital circuits using SystemVerilog, CAD

tools, and FPGA board. These labs would include an 8-bit serial processor, arithmetic units (various adders and multipliers), a 16-bit SLC-3 processor, USB input and VGA display controllers, and a simple SoC that connects a NIOS II embedded processor with a dedicated data-decryption accelerator IP core. Finally, students will have four weeks at the end of the semester to work on an open project of their own choice. This open final project will be graded by creativity, complexity, and functionality of the design. At the conceptual level, students will learn combinational and sequential logic, storage elements, I/O and display, timing analysis, design tradeoffs, synchronous and asynchronous design methods, data-path and controller, microprocessor design, software/hardware co-design, and embedded systems.

DOCUMENTATION AND PROCEDURES:

Until the final project, a new experiment is performed each week in the laboratory. Each experiment requires a pre-lab write-up and a post-lab write-up. Most require an in-lab demonstration. Students are encouraged to discuss general aspects of designs and other aspects of the course with each other. Each lab pair must, however, develop its own detailed circuit implementation and documentation. Except where specifically noted, only one copy of the logic diagrams, component layouts, post lab and the written report is required from each lab pair.

The pre-lab documentation is simply a presentable record of the basic design work which should be done simultaneously with the design, construction, and testing of the circuit. The pre-lab materials should reflect the process of designing, constructing, and testing the circuit.

The pre-lab material may consist of:

1) a written description of the operation of the circuit with functional block diagram
2) Karnaugh maps, Boolean equations, state diagrams, next-state equations
3) a logic diagram with all gates and ports labeled
4) a component layout
5) testing related documentation – such as procedures and results
6) log of major milestones in circuit design (source control commit logs)
7) answers to questions in the pre-lab, if any

The format and conventions presented in the general guide should be used. Each team should bring the pre-lab documentation as well as the design implementation at the beginning of

the lab period. Past experience indicates that it is highly unlikely for a group to obtain any demo points if they have not completed the pre-lab and the implementation beforehand. It is strongly recommended (after the first week) that you and your lab partner meet before coming to the lab to discuss the experiment, decide on a circuit to be built, and wire up that circuit or complete the SystemVerilog design before coming to the lab. This will leave your lab time completely free for debugging and modifying circuits, making observations and collecting data with additional assistance from your TA. If all goes well then, the entire lab period may not be needed. There are additional open lab times available beyond the scheduled time for students to work in the designated labs. However, given that the actual lab period is very short, it is very important to do as much work as possible outside the lab before coming to the scheduled lab session. In general, the amount of time students will need out commit outside of lab varies significantly and depends strongly on how well the students understood pre-requisite materials. **However, most students will expect to budget at least 9 additional hours per week,** given that ECE 385 is a 3-credit hour class.

When you have completed the experiment, ask the TA to observe a demonstration or to check out your circuit. Following the demonstration, each team writes up a post-lab consisting of the pre-lab materials augmented with additional test results, corrections of faulty designs, and answers to specific questions posted in the experiment handout. The post-lab write-up (lab report) is always due by 11:59 PM on the day of your next lab period. This deadline will be enforced.

GRADING:

Grade breakdown for each of the ten experiments is as follows:

	Report	Demonstration	Total
Experiment 1	15	0	15
Experiment 2	15	5	20
Experiment 3	15	5	20
Experiment 4	15	5	20
Experiment 5	15	5	20
Experiment 6	20	10	30
Experiment 7	15	5	20
Experiment 8	15	5	20
Experiment 9	20	10	30

Additionally, the final project will be graded out of 60 points: 30 for the report, 20 for the demo, and 10 for normalizing projects of varying difficulty. There will be two **in-class** exams which will be designed to test your understanding of the various concepts learnt in the lab **and**

lectures. Each will be worth 25 points and consist of 25 multiple choice questions for a total of 50 points. Finally, 15 points will be assigned by your lab partner in three separate performance evaluation reports (5 points each), bringing the total points available in the semester to 320. Reports will be graded and comments will be available on Compass so that students will get suggestions and advice to improve their report writing skills. The majority of your course grade will be based on the designs given in your pre-labs and your ability to detect, explain and redesign faulty circuits in your post-labs. The TA's evaluation of your lab technique will be considered when deciding borderline cases. It should be emphasized that the ability to clearly and precisely describe a design is a very important part of digital design and is weighted heavily in the grading.

STRONGLY RECOMMENDED:

1. Attend every lecture. Concepts covered in lectures will appear on the two exams (worth a total of 50 points). Lectures will also cover common pitfalls students have in the design of each lab. Attending the lectures **will save you time in the long run**.

2. Attend open lab hours. Open lab hours will have undergraduate TAs available to help you debug your designs.

3. Start early! Early in the week sections will benefit from optional Friday Q/A session to discuss upcoming lab.

4. Use the performance evaluation reports (the first two of which will be filled out jointly by both lab partners) to honestly assess what worked well and what needs improvement up to that point. **Take appropriate action from this.** For example, if it is clear one lab partner has missed a significant number of meetings, their performance evaluation should state so and along with remedial actions for the next portion of the course (different meeting times, change in partners, etc).

RECOMMENDED

1. 6" needle nose pliers for ease of insertion and removal of very short wires in protoboard.

2. Schematic capture software (e.g., EAGLE, SketchIT, Fritzing, Altium, Circuitlab). These tools are of varying complexity and capability, but it will benefit your long-term career to be familiar and comfortable with a schematic capture software.

3. SystemVerilog simulation and synthesis tools (for free download; see links on ECE 385 homepage).

4. Yale N. Patt and Sanjay J. Patel, Introduction to Computing Systems, McGraw-Hill, Boston, Second Edition, 2004 (ECE 120 Text Book).

SUGGESTED READING:

1. John F. Wakerly, Digital Design: Principles and Practices, Prentice Hall, New Jersey, (Third Edition), 2000.
2. Stuart Sutherland, SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling, Springer, New York, 2006.
3. James. O. Hamblen and M. J. Furman, Rapid Prototyping of Digital Systems – A Tutorial Approach, Kluwer, 2001.
4. Stuart Sutherland and Don Mills, Verilog and SystemVerilog Gotchas, Springer, 2007.

University of Illinois ECE 385

GENERAL GUIDE

I. LAB KITS

Equipment List

The lab kits checked out in a box to each group contain:

1) A protoboard and a 5V Power Supply.

2) No. 22 wire kit in assorted colors and lengths.

3) Discrete components (resistors, LEDS, SWITCHES).

4) About fifty assorted integrated circuit chips or packages.

5) Altera DE2-115 FPGA board, power supply, and programming cable

Integrated Circuit (IC) Packages

The IC's in the kit will be similar to the one shown in Figure 1. Since the pins on newly manufactured IC packages are bowed outward for use with automatic insertion equipment, it may be necessary to gently bend the pins so they will go into an individual socket easily. This is best done by pushing each row of pins inward on a flat surface.

Consider the IC chip shown in Figure 1. The number SN7400N on the chip indicates the following: The SN is a prefix used by Texas Instruments, Inc., the 74 indicates the chip is from the 74-hundred family of Transistor-Transistor Logic (TTL), the 00 indicates the exact type of circuit. The suffix N gives information on the packaging (e.g. N for Plastic DIP, J for Ceramic DIP, DIP stands for Dual In-line Package) as well as the connection between the internal logic circuits and the external pins. On some chips a few letters may appear between the family digits and the type digits; e.g., SN74LS00 and SN74S00. These letters are used to indicate that the chip differs electrically (but not logically) from standard TTL chips (e.g. LS stands for Low power Schottky, S for higher speed Schottky). The digits 7523 are a manufacturing date and indicate the IC package was made in the 23rd week of ’75 (an antique given the current iteration of this document!) The 54-hundred family provides the same functions as 74-hundred family, but with mil-spec extended operating temperature range and radiation hardening.

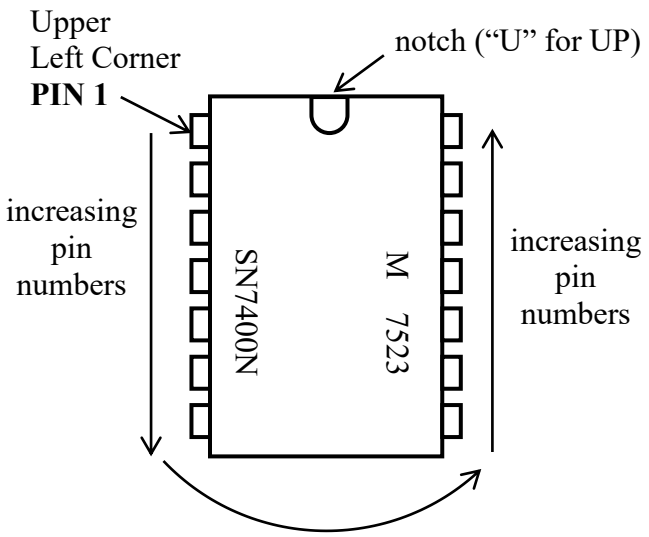
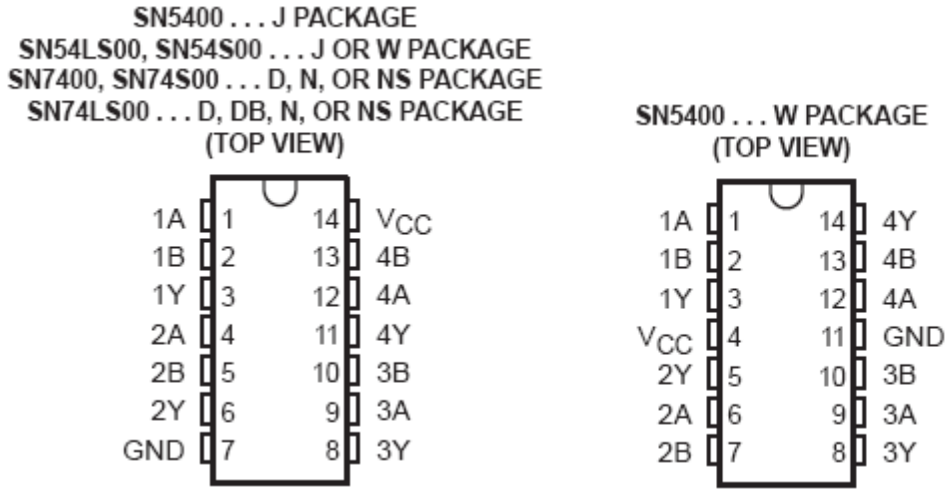


Figure 1. IC Package

54/74 FAMILIES OF COMPATIBLE TTL CIRCUITS



These devices contain four independent 2-input NAND gates. The devices perform the Boolean function $Y = \overline{A \bullet B}$ or $Y = \overline{A} + \overline{B}$ in positive logic.

Figure 2. Data Sheet for 54/7400

Using Data Sheets

A part of data sheet for the 7400 package is reproduced from Texas Instruments website (www.ti.com). The data sheet states that the chip contains four independent 2-input NAND gates, and gives Boolean function in variables A, B and Y. Each NAND is

identified on the pins using these same variables with a distinct prefix. For example, 3A, 3B and 3Y. The data sheet also indicates that pin 7 must be grounded and pin 14 must be connected to power (V_{cc}), which is +5 volts for all chips in the 74 hundred family. You will be able to find the data sheets for the chips we will be using on our course website (under Lab 1).

A standard lab kit of IC's consists of the following:

Quantity	Type	IC Description
5	SN7400	Quad 2-input NAND
4	SN7402	Quad 2-input NOR
2	SN7404	Hex Inverter
1	SN7406	Hex Inverter with Open Collector
2	SN7410	Triple 3-input NAND
2	SN7420	Double 4-input NAND
1	SN7427	Triple 3-input NOR
1	SN7474	Dual D Positive Edge Triggered Flip Flop
2	SN7485	4-bit Magnitude Comparator
3	SN7486	Quad 2-input Exclusive-OR
2	SN7493	4-bit Ripple Counter
2	SN7495	4-bit Parallel-Access Shift Registers
4	SN74LS107A	Dual JK Negative Edge Triggered Flip-Flops
2	SN74LS112	Dual JK Negative Edge Triggered Flip-Flops
1	SN74151	8 to 1 Multiplexer
2	SN74153	Dual 4 to 1 Multiplexer
3	SN74157	Quad 2 to 1 Multiplexer
1	SN74LS169A	4-bit Synchronous Up-Down Counter
2	SN74193	4-bit Synchronous Up-Down Counter
3	SN74LS194A	4-bit Bidirectional Universal Shift Register
2	SN74LS279	Quad S-R Latch

DISCRETE COMPONENTS AND TOOLS

Quantity	Type
1	10 Light emitting diode DIP (single package)
10	330 Ω resistors (orange-orange-brown)
8	1 KΩ resistors (brown-black-red **or** brown-black-black-brown)
1	8 SPST switch DIP (single package)
1	4 SPDT switch DIP (single package)

TTL Characteristics

Power: TTL IC's require a 5V power supply. Each IC package has two power pins, one to be connected to GND (0 V) and one to be connected to V_{cc} (+5 V). All IC's used in the laboratory should be mounted with the indentation or notch, which is located on one end of the IC package, positioned up (away from you). When an IC is positioned in this fashion (pin 1 in the upper left corner and the highest numbered pin in the upper right), most IC's will have their GND connection in the lower left corner and their V_{cc} connection in the upper right corner. There are a few exceptions, however, so always check the data sheet. In the lab kit, the 7474, 7475, 7483 and the 7493 have non-standard power and ground pin locations.

Logic Levels: Under the positive logic convention, low voltage inputs and outputs (from 0.0V to about 0.7 V) are interpreted as logical 0 and high voltage (from about 2 V to 5 V) as logical 1. To place logical 0 on an input, connect it to GND. To place logical 1 on an input, connect it to V_{cc} through a 1kΩ 1/4 W resistor. Although it is possible to connect a logical input directly to V_{cc}, having a 1kΩ resistor will prevent the destruction of the device if you misread the datasheet or miscount the pin number. One such resistor can generally be used to pull as many as eight inputs high. In addition, this pull-up resistor will limit input current during transients (e.g. rapid switching). A gate input which is disconnected is generally considered an unknown logic level. **Some** gates have weak internal pull-up resistors which will cause disconnected inputs to read as logical 1. However, if your circuit contains floating inputs, they are apt to be sensitive to noise**, therefore you should not rely on this behavior.** Unused

clock, mode, clear and preset pins are particularly sensitive and **must** be tied to the appropriate logic level for reliable operation.

Loading: Most TTL gate-outputs will drive a maximum of ten standard TTL inputs or ten "unit loads" (i.e., they have a maximum fan-out of 10). Connection to more than ten gate-inputs may overload the output and produce unpredictable results. A few non-standard TTL outputs are not able to drive as many as ten unit loads and a few non-standard TTL inputs are more than one unit load. In the lab kit, the non-standard chips are the following:

Non-standard inputs

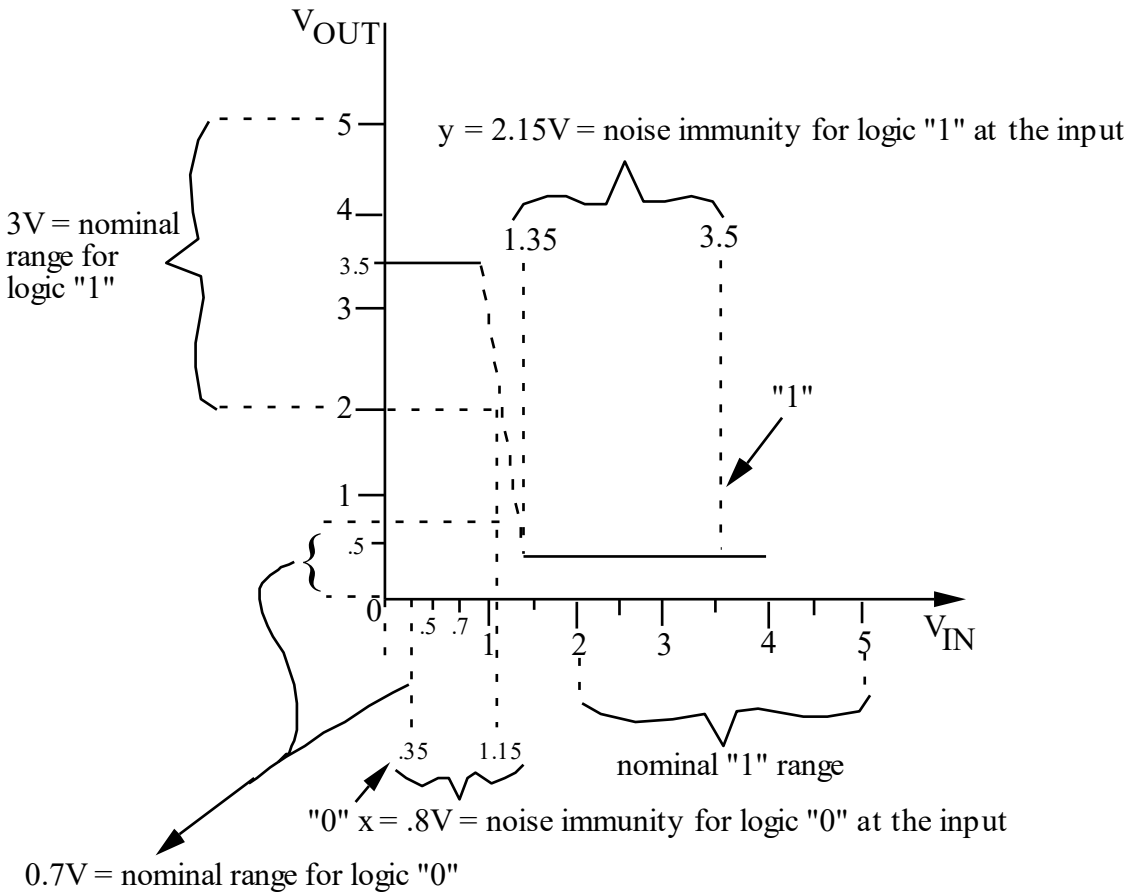
7474	Clear, Clock	2 loads
7475	D, G	2 loads, 4 loads
7583	A_1, A_3, B_1, B_3	4 loads
7493	A, B	2 loads
74LS107	Clear, Clock	2 loads
74LS112	All inputs	1/4 to 1/2 load

Non-standard outputs

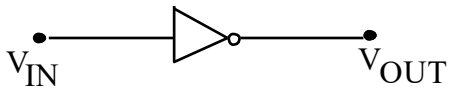
74LS112	Outputs drive only five unit loads

The Noise Immunity of a gate is defined as the maximum amplitude of a positive-going (noise) pulse added to the nominal logic "0" voltage level or a negative-going (noise) pulse added to the logic "1" voltage level at the input of a gate which does not cause the output of that gate to change its logic value. The nominal logic "0" and "1" voltage levels can be determined by connecting several inverter gates (e.g., 7404) in series and observe the voltage levels at the output of the last inverter when the input to the first inverter is at GND and at +5v. What is the advantage of a larger noise immunity? Why is the last inverter observed rather than simply the first? Given a graph of output voltage (VOUT) vs. input voltage (VIN) for an inverter, how would you calculate the noise immunity for the inverter? See the following figure.

NOISE IMMUNITY



THE OVERALL NOISE IMMUNITY OF THE GATE IS THE SMALLEST OF THE RANGES X AND Y.



II. LAB STATIONS AND PROTOBOARD

Connections to switches, LEDs, hexadecimal displays and other external devices will be required during the lab sessions. This can be done conveniently by connecting #22 gauge solid wire between inputs and outputs of your circuit on the protoboard to specified holes in the 60-pin connectors mounted on the protoboard (see Figs. 4 and 5). At the lab, the protoboard is connected to the I/O boards of the lab station via two 60-line flat cables. Note that the connectors on the protoboard have latches to ensure a good connection to the ribbon cable.

A. THE PROTOBOARD

The protoboard (see Fig. 4) is a board on which all circuits will be built. It contains: i) socket strips used to mount IC's and interconnect them, ii) 3 BNC connectors for signal input from the pulse generator and signal outputs to the two oscilloscope traces, and iii) two 60-pin connector-plug pairs for easy interconnection to the LAB STATION.

The 5 volt power supply (provided inside the lab kit box) is current limited so that a short circuit will not damage the supply. However, a short circuit may damage the protoboard or destroy a chip (a common mistake is to reverse power and ground). Power can be accessed by connecting two cables to two binding posts on the protoboard, +5V (Red) and GND (Black).

The socket strips are used to mount and interconnect components. Five socket strips for components and six bus strips which are useful for bussing commonly used signals such as power, ground or clocks, are mounted vertically. Each strip is simply an array of holes with spring clips underneath them. Stripped #22 gauge solid wire, provided in the lab, must be used for interconnections in the Protoboard. **Be sure that the portion of the wire inserted is straight**. **If wire heavier than #20 gauge is forced into the holes, it will permanently damage the spring clips**. Any components with heavy gauge leads, e.g., large capacitors, resistors, crystal oscillators, etc. must be soldered into IC headers or soldered onto #22 solid hookup wire. In general, you will not need to do this in ECE 385. The spring clips in the Protoboard are connected as shown (Fig. 4).

When the ECE 385 I/O board is connected to the Protoboard you have access to:

i) 16 debounced switches

ii) 16 light emitting diodes (LED) with drivers to monitor the state of various lines in your circuit; and

iii) 4 hexadecimal displays that will display the following characters in response to the corresponding hexadecimal binary coded inputs 0 thru 9 and A, b, C, d, E and F.

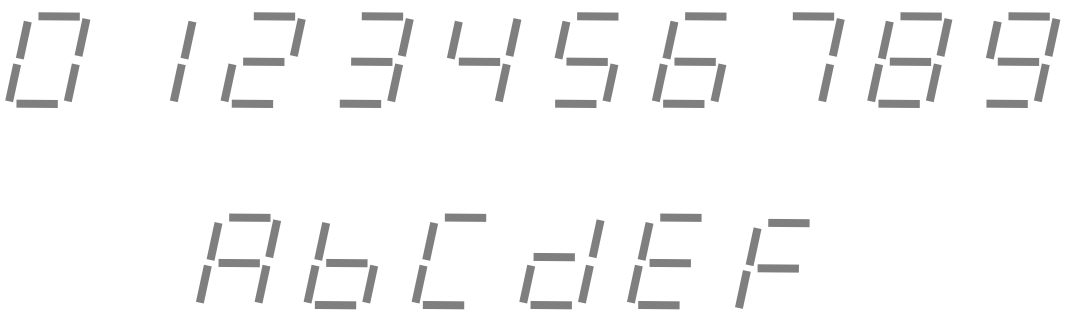


Figure 3

The connection points for the 60-pin connector can be seen in Fig. 5.

All I/O board switches are debounced.

The hexadecimal displays are driven by binary to hexadecimal decoder/drivers that generate a unique display symbol for each hexadecimal 4 bit binary coded input (0-9 and A-F). Blanking of D0 and D1 is possible by setting pins 41 and 43 high on the Protoboard.

The ICs used in your circuits must be labeled, in your logic diagram and component layout diagram, using the column letters A, B, C, D, and E and the row numbers 1 to 7 (see Figs. 4, 8, and 9).

Figure 4. ECE 385 Protoboard

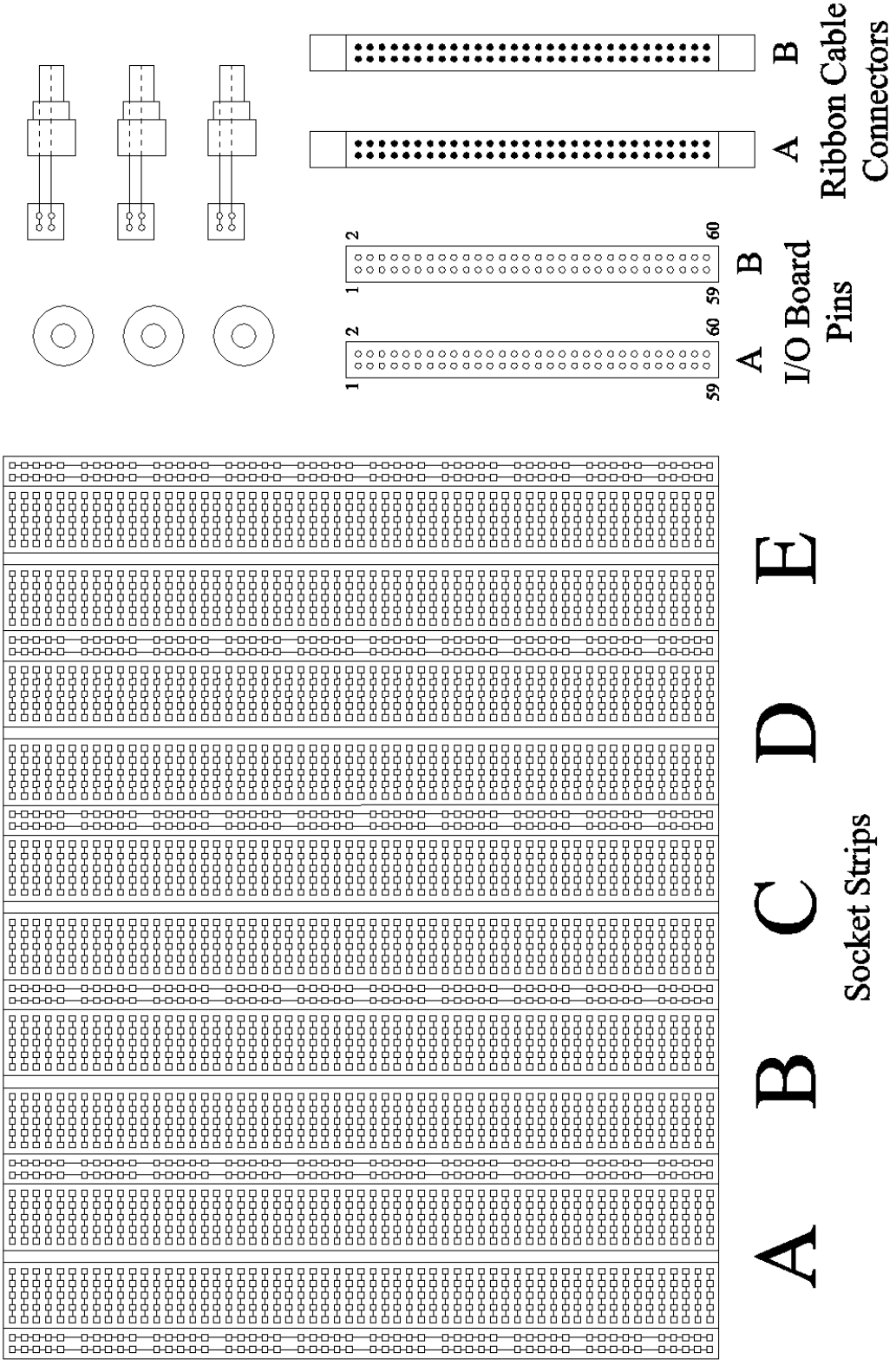
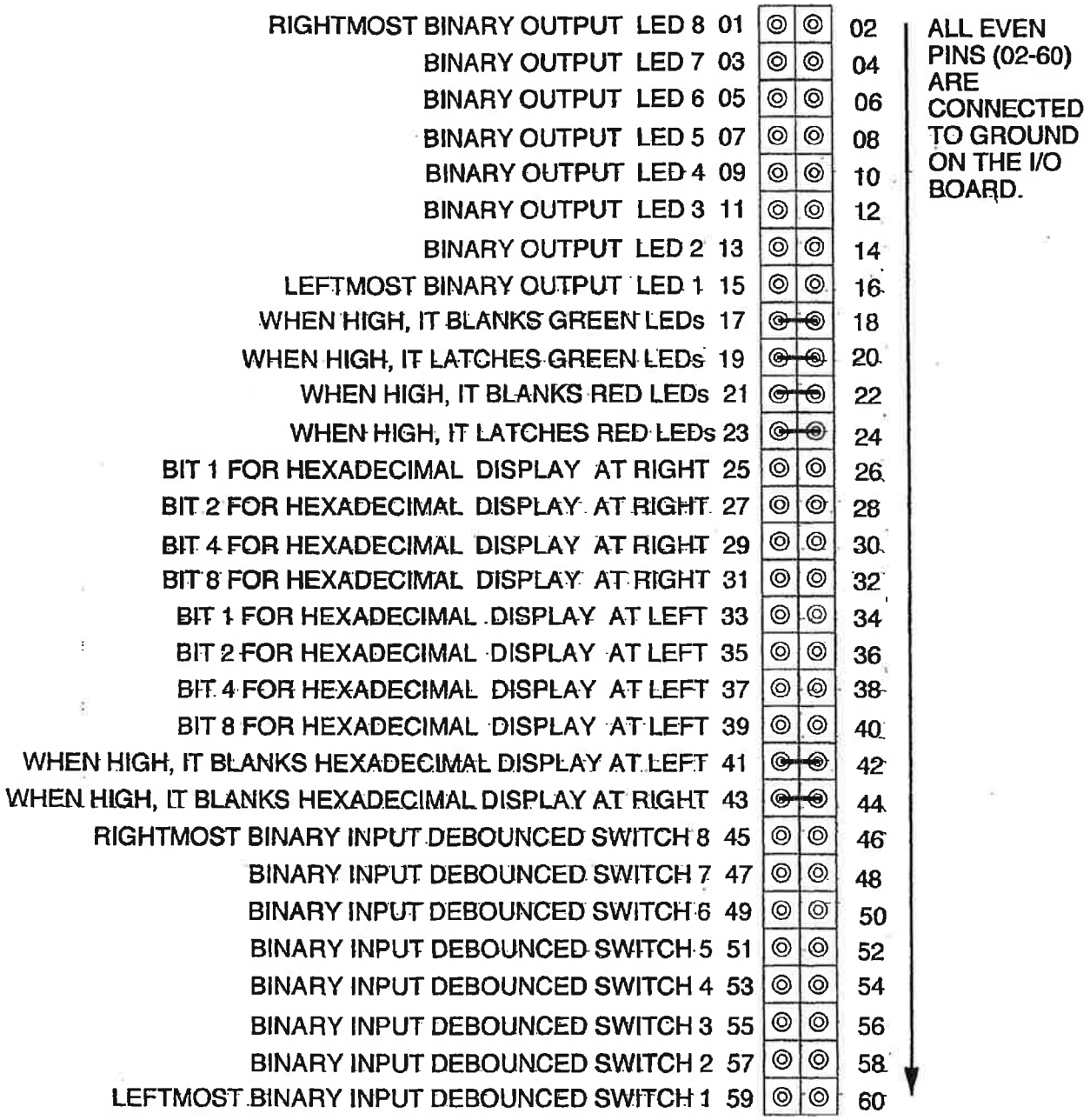


FIG . 5. 60 - PIN CONNECTOR ASSIGNMENT
FOR EACH SECTION OF THE ECE 249 I/O BOARD.



GG.12

FIG . 5. 60 - PIN CONNECTOR ASSIGNMENT
FOR EACH SECTION OF THE ECE 249 I/O BOARD.

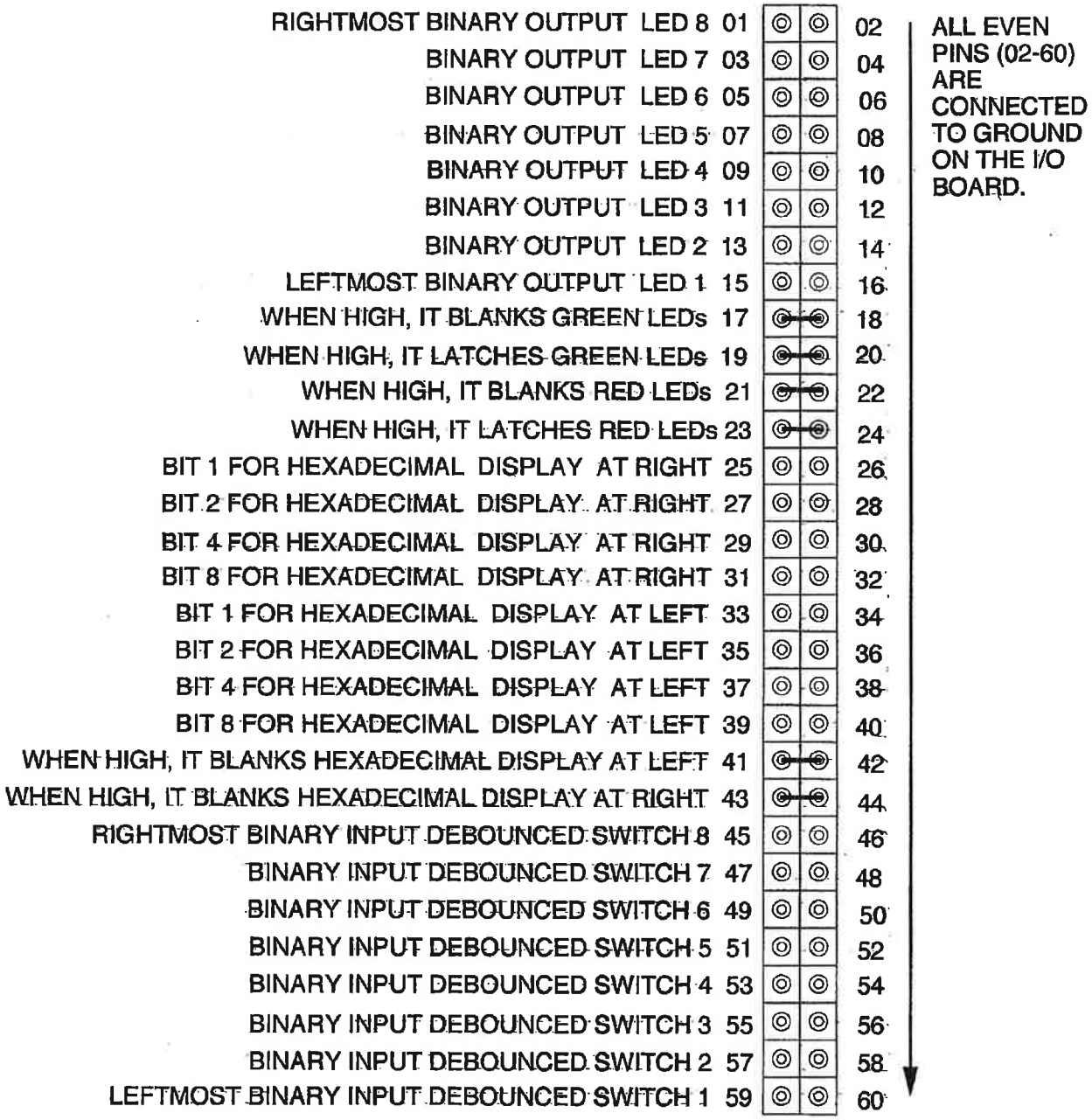
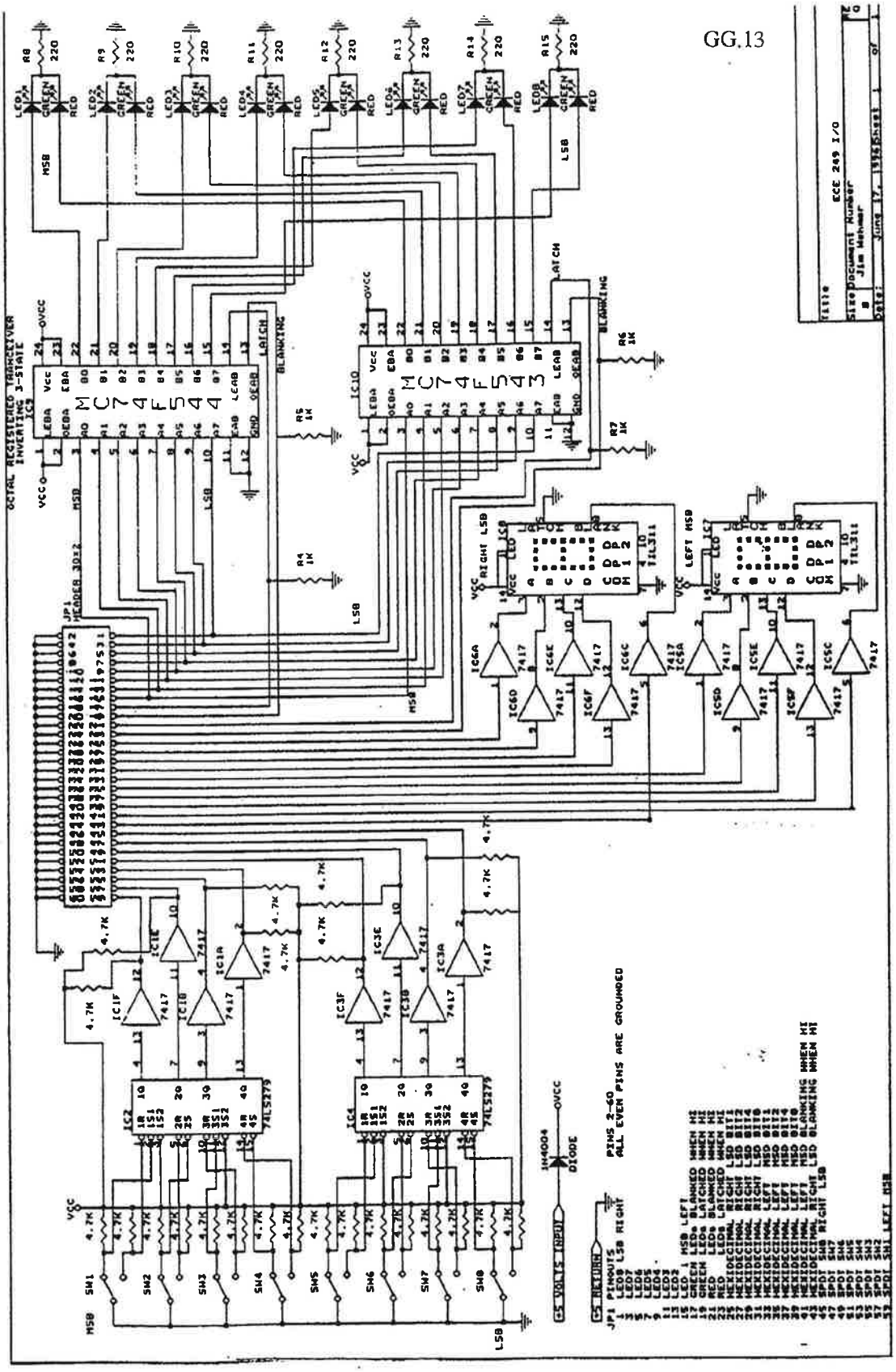


FIG. 6


OCTAL REGISTERED TRANSCEIVER
INVERTING 3-STATE
MC74F544
MC74F543
LATCH
BLANKING
MSB
LSB
RIGHT LSB
LEFT MSB
74LS279
7417
TIL311
1N4004
DIODE
5 VOLTS INPUT
PINS 2-60
ALL EVEN PINS ARE GROUNDED
ECE 249 I/O
Jim Wehner
June 17, 1996

GG.13

B. THE LAB STATION 16-BIT I/O BOARD

A brief description of the new I/O board and its features follow:

Input (to the user's hardware) is generated by 16 SPDT debounced switches arranged as a 16-bit word.

Output (from the user's hardware) is displayed by:

i. Sixteen two-color (red, green) light emitting diodes (LEDs) arranged as a 16-bit word.

ii. Four hexadecimal displays.

The 16-bit I/O board is organized in two identical sections of eight bits each with input (eight switches) and output (eight LEDs and two hexadecimal displays). See GG.11, 12.

Each section connects to the user's hardware through a 60-line flat cable. Thirty lines are used for signals (odd numbered) and thirty lines are used for ground (even numbered) so that there is a ground between any two signals throughout the length of the cable.

Normally, the LEDs represent a high (H) logic level with green on and a low (L) logic level with red on (e.g. logic in color mode). However, for students who have trouble distinguishing colors, either color may be blanked (turned off) for all eight bits of each section. When a high (H) is presented at line 17 (of the corresponding 60-line flat cable) the green will be blanked. Alternatively, when a high (H) is presented at pin 21, the red will be blanked. Therefore, either color may be used to represent a logic level with the complement represented by no light. Line 17 and/or line 21 should be driven by a buffer.

Moreover, either color may be latched (all eight bits of each section) when a high (H) is presented at line 19 to latch the green, or a high (H) is presented at line 23 to latch the red. Note that if both red and green are used to represent logic levels, both must be latched to preserve the data displayed. Line 19 and/or line 23 should be driven by a buffer. Be aware of accidentally invoking this behavior.

Either of the two hexadecimal displays of each section may also be blanked with a high (H) on line 41 to blank the display on the left or with a high (H) on line 43 to blank

the display on the right. Line 41 and/or line 43 must be grounded when blanking of the hexadecimal display is not required.

C. HP FUNCTION (WAVEFORM) GENERATOR

The HP 33120A function/arbitrary waveform generator uses direct digital-synthesis techniques to create a stable, accurate output signal for clean, low-distortion sine waves. It also gives you fast rise and fall-time square waves, and linear ramp waveforms down to 10 MHz.

In ECE 385 we will primarily use square waves to generate the clock signals, and other inputs, for our digital circuits. Hence, it is very important that the function generator is set properly before each use to ensure proper circuit function. Do not assume your generator is properly setup. Failure to setup your generator will result in circuit malfunction and may damage your chips or the generator. **Note that by default the function generator is designed to drive a bipolar (both positive and negative going) signal into a 50 Ω load. This may destroy TTL chips which are designed for 5 V TTL unipolar signals.**

Setup Procedure:

1. **Turn function generator on.**
2. **Select high impedance output.**
    - Press the blue **'Shift'** button then the **'Enter'** button.
    - You are now on **'A: MOD MENU'**
    - Hit the **'>'** button three (3) times.
    - You are now on the **'D: SYS MENU'**
    - Hit the '\/' button twice.
    - If the display says **'HIGH Z'**, skip to step 3.
    - If the display says **'50 OHM'**:
    - Press the **'>'** button once.
    - The display should say **'HIGH Z'**, if it does not repeat this step again.
    - Press the **'ENTER'** button.
    - The display will flash **'ENTERED'**
3. **Set a 0 to 5 Volt square wave signal.**
    - Select a square wave by pressing the '¯|_' button
    - Press the **'Ampl'** button.

- Turn knob to set **'5.000 VPP'**
- Press the **'Offset'** button.
- Turn knob to set **'+2.500 VDC'**

4. **Set signal frequency**
   - Press the **'Freq'** button.
   - Turn knob until desired frequency is set.
5. **Check the signal from the output port with the oscilloscope.**

HP 33120A Single Step Mode:

Single step mode is often useful in debugging clocked circuits, as it allows you to send single clock pulses instead of a continuous clock.

Setup Procedure:

- **Follow the steps shown above to set up HP generator for desired signal**
- **Set up burst mode**
  - Press the blue **'Shift'** button then the **'Sawtooth Wave'** button (Button #4).
  - The word **'Burst'** should appear in the middle bottom of the display
  - Press the blue **'Shift'** button, then press the **'<'** button once
  - You are now on the **'4: BURST CNT'**
  - Hit the '\/' button once.
  - The display will say **'00001 CYC'**
  - Turn the dial to set a value for **'CYC.'** This value is the number of pulses that will be sent each time you press the **'Single'** button; e.g. if you set the count to '00003,' when you press **'Single'** you will get a square wave that looks like: _____|‾‾|__|‾‾|__|‾‾|_____
  - Press the **'ENTER'** button.
  - The display will flash 'ENTERED'
- Press the **'Single'** button (Button #9) to begin single step mode.
- From now on, every time you press **'Single,'** the number of clock pulses specified by the CYC value will be sent.

- To exit single step mode, press the blue **'Shift'** button then the **'Sawtooth Wave'** button (Button #4)

D. ALTERA DE2-115 FPGA BOARD

Included with the lab kits at check-out time is one DE2-115 FPGA board, one 9V power supply for that board, and one USB programming cable (USB A to B). We will introduce the DE2-115 FPGA board in Lab 4.

III. LAB DOCUMENTATION

Overview

Precise and uniform documentation is an indispensable part of digital design. The documentation is simply a presentable record of your design procedure supplemented with a written description of the operation of the circuit. From your documentation, someone unfamiliar with the experiment should be able to understand what your circuit does and should be able to wire up and test your circuit without reference to any other handbooks or literature. Good documentation is essential for correct circuit-assembly and for circuit-examination by an instructor. **The documentation should reflect what was done in the lab and should not be considered an independent portion of the course.**

By default, a complete lab report should consist of an **Introduction** which states the purpose of the lab, **Documentation** for each circuit, **Answers** to the questions asked in the lab manual, and a **Conclusion** to wrap up the things learned.

Basic **Documentation** for a digital circuit designed should consist of

(1) A written description of the operation of the circuit and a block diagram

(2) Karnaugh maps, State Diagrams and Tables, and Boolean Equations

(3) A Logic Diagrams (preliminary AND-OR and final NAND implementations)

(4) Component Layout

Though the four parts have been listed in the correct order for a reasonable derivation of a completed design, the actual design process will probably proceed in a different order. For simple designs it is probably most convenient to draw a partial logic diagram, determine a component layout, complete the logic diagram, and finally supply some description of the circuit operation. More complicated designs will require completing some of the descriptive material (e.g., signal definitions, state transition diagrams) before a logic diagram can be started. Complicated designs will probably require moving back and forth between the design levels several times (e.g., the logic diagram may suggest a better set of signal definitions).

Written Description with Block Diagram

The ability to describe a circuit in a manner which is organized, precise, clear, and easily understood is a very important part of digital design. The amount and type of description needed will depend upon the complexity of the circuit. But almost always, a high-level functional block diagram is an integral part of any written description. The high-level block diagram should be the foundation of this section of your lab documentation but may not be complete by itself. Further elaboration (sub-diagrams or written descriptions) will likely be required.

Karnaugh Maps, State Transition Diagrams and Tables

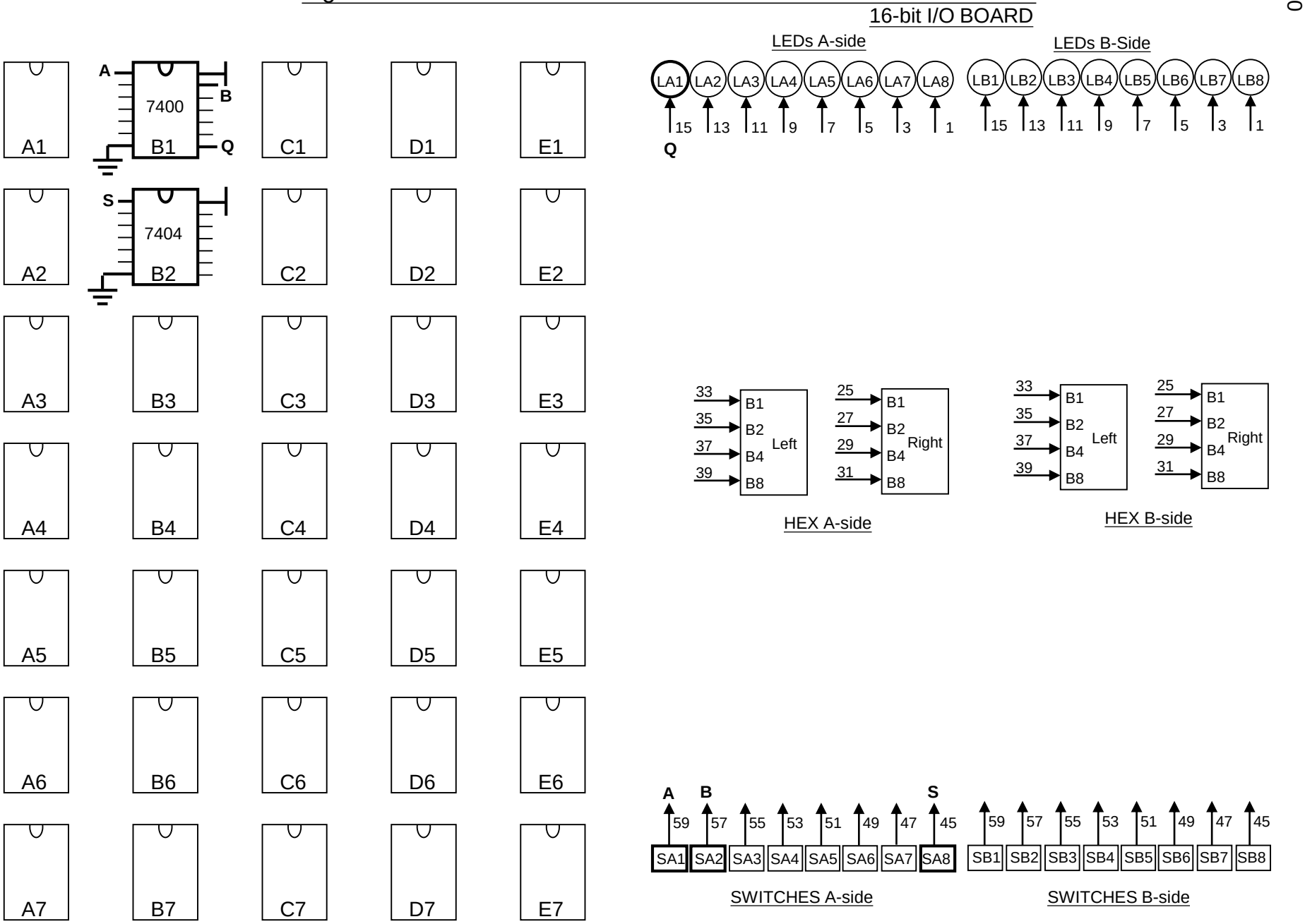
Here you present your derivation of logic using formal techniques learnt in ECE 120. Often there are several alternative solutions. You should present your rationale for choosing an implementation.

Component-layout

A component layout is used to indicate which types of IC will be needed, where they will be mounted, which pins require power connections, where the input signals originate, and where output signals are monitored. This information is needed for circuit construction and is essential for circuit debugging when one needs to find the physical location of a given logic signal (net) on the logic diagram.

A sample component-layout is given in Fig. 8. Each switch and LED to be used should be labeled with the symbolic name for the associated digital signal. The five IC socket strips are denoted A to E from left to right. IC packages are numbered 1, 2, 3,… from top to bottom. (In general, seven ICs will fit from top to bottom, but differing package sizes may change this.) Thus, C2 denotes the second IC down on the third socket-strip from the left. Each IC location is labeled with the position designation and the number of the IC to be inserted there. The component layout sheet should also show the power and ground connections for each chip, especially those with non-standard power and ground pin locations. Using the standard orientation (notch up) most chips will have the ground connection in the "lower left corner" and +5 volts in the "upper right corner." However, you should always check the data sheet for power and ground pin numbers since some chips are non-standard. Careful component-layout can greatly reduce wire lengths and greatly enhance the appearance and maintainability of a digital circuit.

Figure 8. SAMPLE COMPONENT LAYOUT AND I/O ASSIGNMENT



Logic Diagram

A sample logic diagram is shown in Fig. 9. Each device (gate, flip-flop, etc.) is drawn using the standard logic symbols. Each device is labeled according to its IC location and IC pin numbers. I/O signals are given symbolic titles, written next to their corresponding switches/LEDs/Hex displays. I/O ports are labeled according to their type (S, L, or H), their side of the I/O board (A or B), and their number (1 to 8, from left). Thus, the label SB6 signifies the third switch from the right of the I/O board (switch 6 of side B). Signal lines drawn crossing each other are interpreted as being unconnected unless a dot is drawn at their intersection. Should your design require multiple logic diagrams to properly represent, signal lines going to and coming from other diagrams are drawn to enter the diagram at the left and leave the diagram at the right, using standard port-in and port-out notation, and are labeled with the source/destination diagram name and signal name, separated by a slash. As your designs get more complex, logic diagrams can be component level (RTL or Register Transfer Language) rather than gate level. For example, in experiments post-Lab 1, a 2:1 multiplexor can be drawn as a block by itself.

2-TO-1 MULTIPLEXER

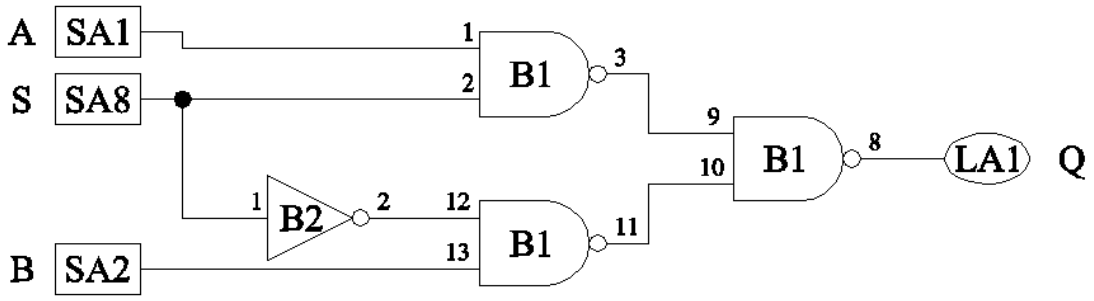


Figure 9: Sample logic diagram.

(Note: Figures 8 and 9 are corresponding diagrams for the same circuit.)

IV. DESIGN TECHNIQUES

General Considerations:

Introductory logic design courses such as ECE 120 choose to ignore certain aspects of circuit design to provide a clearer presentation to the student. Technological constraints imposed upon the designer are ignored to emphasize a strong theoretical base for further study. In the laboratory, actual device characteristics must be included in all designs.

In Transistor-Transistor Logic (TTL), a NAND gate can be implemented with fewer transistors than an AND gate. In addition, since any function can be implemented with NAND gates, their use can reduce overall package count and diversity in a design. The ECE 385 student lab kit contains no AND or OR gates; it contains NAND and NOR gates. Design with AND or OR gates is conceptually easier than design with NAND and NOR gates because the design follows intuitively from Boolean algebra equations. A simple technique for converting two level AND/OR circuits to NAND circuits is given later in these notes. You will have to design your circuits considering what actual devices are available in the ECE 385 student lab kit.

Another point to remember when designing with actual devices is that all unused inputs to the device must be tied to some logic level. The decision to tie an input to zero or one is made by checking the input's function on the device data sheet. To tie an input to a logic zero, simply connect it to system ground. To pull an input to a logic one, a ‘pull-up’ resistor (nominally 1kΩ for 74LSXXX) must be inserted between the input and the +5 volt power supply. Note that a single resistor may be used to hold more than one (up to ten) input at the logic one level.

Delays and Glitches:

An extremely important characteristic of TTL (and other technologies) that is often overlooked by inexperienced designers is that all logic gates have a nonzero propagation delay from input to output. These propagation delays are infinitesimal with respect to the human observer. However, the propagation delay of TTL gates is finite and may lead to the occurrence of glitches in a signal. A glitch is a momentary incorrect output value produced by a circuit during periods when the inputs are changing. Glitches may be detected during the design phase using timing diagrams. Glitches are one of the most important new concepts we will discuss in ECE 385 and have wide ranging implications for digital design in general.

The circuit of Figure 10 should always output a logic one at C as it implements the steady-state logic equation $C = A \oplus \bar{A} = 1$. However, due to the finite delays of the gates used, it may momentarily output a logic zero (a glitch) right after the input A changes. The inverter in the circuit of Figure 10 is from a 7404 chip. The 7404 has a typical propagation delay of 10ns, and a guaranteed maximum delay of 22ns. The exclusive-or gate (7486) has a typical delay of 14ns and a maximum of 30ns. Note that neither of these gates has a guaranteed minimum delay time; therefore, the assumed minimum delay is zero. When preparing a timing diagram always use the worst-case propagation delays, not the typical!

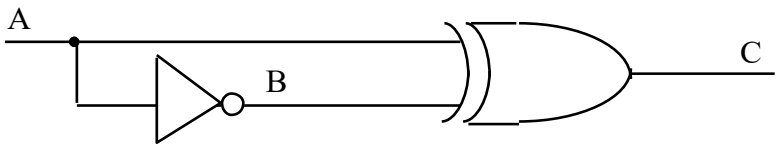


FIGURE 10

A timing diagram for the circuit of Figure 10 is given in Figure 11. At time T = 0, the input A changes from a zero to a one. The output of the inverter (B) is unknown from time T = 0 until time T = 22 ns. The value of B probably changes somewhere around 10ns (the typical delay time), but the output cannot be guaranteed until T = 22ns. Since B is an input to the XOR gate, its output cannot be guaranteed until 30ns (maximum propagation delay) after the value of B is known. The cross-hashed areas in the timing diagram are the times when the signal value is unknown. The XOR output, C, is unknown for 52 ns after the input A changes. During those 52 ns, the output may be at logic zero or logic one. If it becomes a logic zero at any instance during these 52ns, then the circuit is said to produce a glitch.

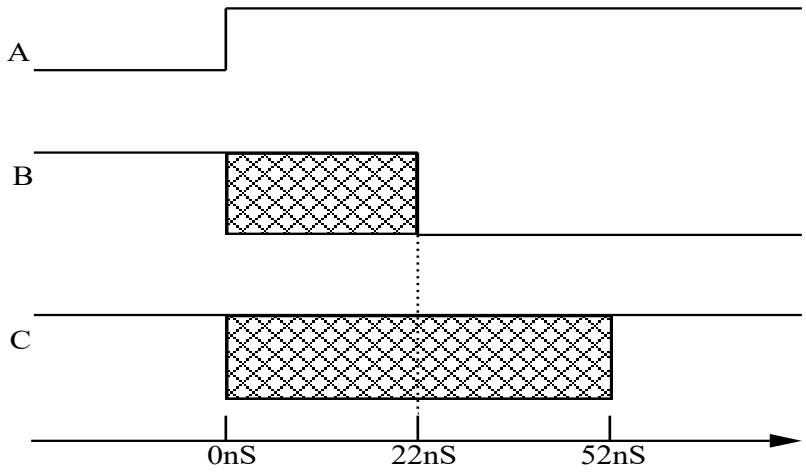


FIGURE 11

Use of NAND (NOR) Gates:

A Karnaugh map of a function is given in Figure 12. To implement this function directly in standard 7400 series TTL components would require three chips: a 7404 hex inverter, a 7408 quad 2-input AND, and a 7432 quad 2-input OR. A logic diagram of the circuit is shown in Figure 13.

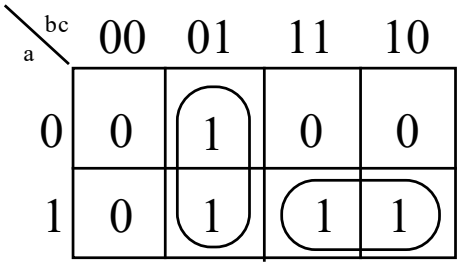


FIGURE 12. Karnaugh map of 2-to-1 Multiplexer Function

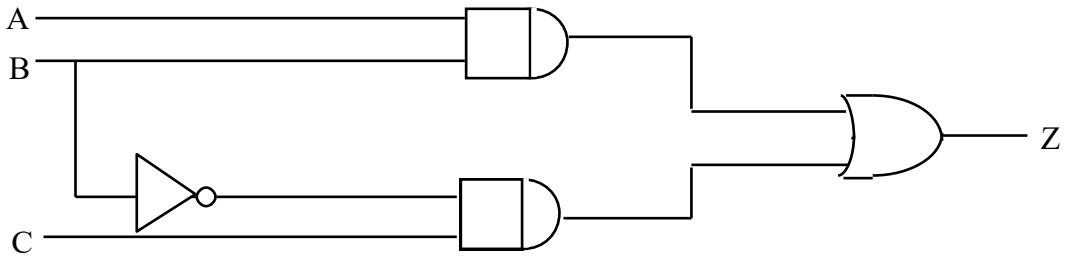


FIGURE 13. Sum of Products implemented with 2-level AND-OR Logic

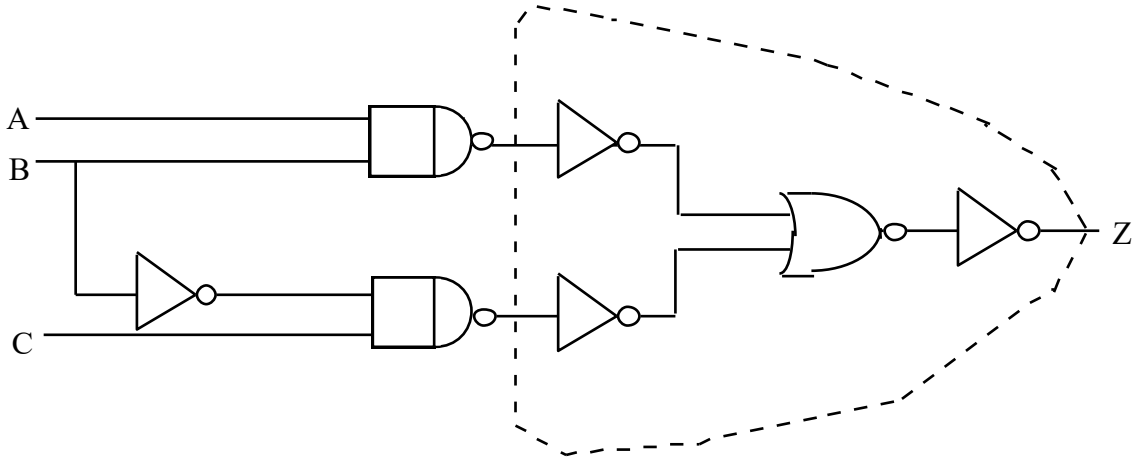


FIGURE 14. Transforming AND-OR to NAND-NAND

Since the ECE 385 lab kit does not contain 7408's or 7432's, it is necessary to modify the circuit to one that uses only NANDs, NORs, and inverters. A direct modification is shown in Figure 14. Each AND gate has been replaced by a NAND gate and an inverter, and each OR gate has been replaced by a NOR and an inverter. Notice the four circled gates in Figure 14: a NOR gate with inverted inputs and inverted output. These four gates can be replaced by a single NAND gate. The resulting circuit is shown in Figure 15. In general: **any two level AND/OR circuit can be replaced by a two level NAND/NAND circuit by** simply replacing each AND and OR gate with a NAND gate.

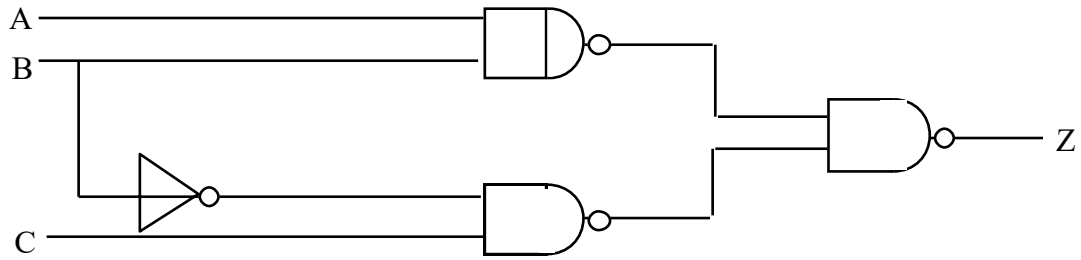


FIGURE 15. Final implementation using NAND-NAND

A slight modification of the circuit of Figure 15 is given in Figure 16. Notice that this circuit requires only one 7400, whereas the original design (Figure 13) required three chips to implement. In this example, the conversion to an all NAND implementation reduced the package count by two thirds.

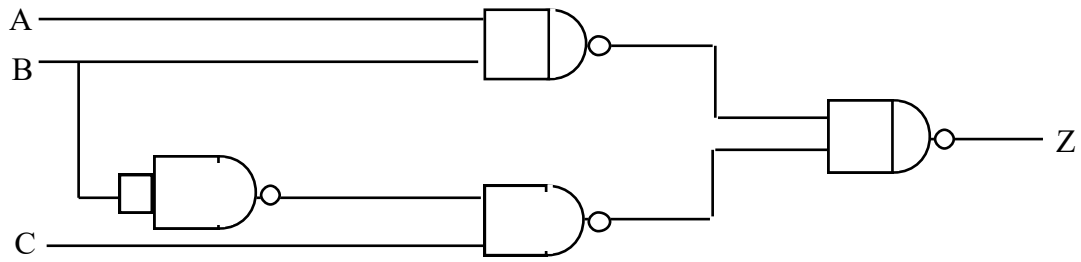


FIGURE 16. Single chip implementation using one 7400 Quad 2-input NANDs

V. LAB TECHNIQUES

Circuit Assembly

Circuits should be assembled in the following manner (with the power supply switched OFF):

(1) Carefully align and insert IC's in their designed positions on the protoboard socket-strips according to your component-layout. Some of the IC's provided have been pre-socketed for their protection and are to remain socketed. Each IC should be positioned with its indentation or notch up (away from you).

(2) Establish a GND-bus along the left sides of active socket-strips and a V_{cc} bus along the right side of active socket-strips (see Fig. 3). Make power connections to all the IC's using black wires for GND connections and red wires for V_{cc} connections. It is essential that power connections be made properly or damage to the ICs, protoboard, and the power supply will result. Check your work!

(3) Make other wiring connections and check them off on your logic-diagram as they are made. Only No. 22 wire and 1/4 W resistors should be used in making connections on the panels. Heavier wire will damage the spring clips in the socket causing faulty contacts in the future.

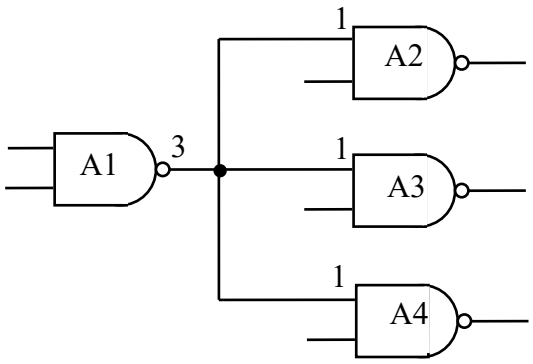
(4) Label (with symbolic names) all switches, pushbuttons, and indicator lamps used. Labels are made by printing on removable gummed labels available in the lab.

Wiring Techniques

Only No. 22 wire and 1/4 W resistors should be used in making electrical connections on the protoboard socket strips. The wire should have approximately 1/4" of insulation removed from the end. If more insulation is removed, the exposed metal wire will often contact an exposed neighboring wire, resulting in unreliable operation at best.

When inserting or removing wires, push or pull in a vertical direction (i.e., don't wiggle the wires or they will break off in the socket). If new wire is cut, cut it to a convenient length and strip 1/4" of insulation from each end.

Select wire colors by function where possible to facilitate organization. However, it is best to use red and only red for +5v wires; and black for GND. Where the output of a gate goes to several different places, debugging and circuit modification is much easier if you do not "daisy chain."



That is, one end of each of the three wires making the electrical connections between gates should go to the socket strip connection points at pin 3 of chip A1. Don't run a wire from 3A1 (pin 3 chip A1) to 1A2 (pin 1 chip A2), a second wire from 1A2 to 1A3, and a third wire from 1A3 to 1A4.

Where possible physically route the wires so that you can test an IC's pins and replace it if it is bad, i.e., do not run wires tightly over the top of IC's. Do not bundle wires tightly into long parallel runs. Signals would then cross-couple from one wire to another, causing noise spikes on a wire when the signal on a neighboring wire changes.

Debugging Techniques

When inserting an IC onto the protoboard socket strip, be certain the pins on the IC are properly aligned. If the pins are not properly aligned, they may fold under when inserted onto the socket strip. The IC will appear to be properly inserted, but the pins which folded under will not make electrical contact. To detect this problem when debugging circuits, always check input logic levels on the exposed IC pins themselves and check output logic levels at one of the connection points on the protoboard socket. Use the oscilloscope. If a

particular hole on the socket strip is found to make a faulty connection, a short stub of wire should be inserted in the hole and clipped off, so that the hole will not inadvertently be used in the future. Always be suspicious of signals which are outside of the nominal range of 5V TTL circuits and signals which appear noisy beyond the operating frequency of the circuit (typically 1 kHz).

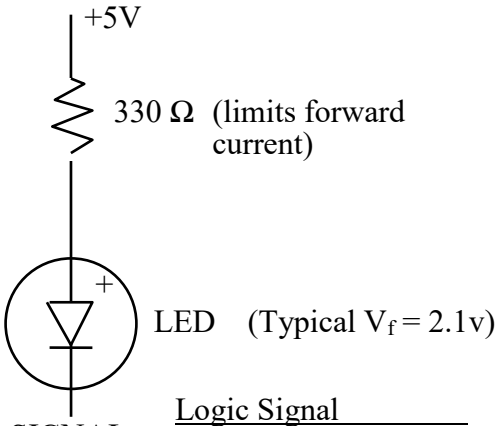
Recall that a floating input is logically unknown. All unused light inputs should be grounded, as well as all unused inputs to an IC chip should be grounded or pulled high with a pull-up resistor (1 kΩ, 1/4 watt).

VI. DEBUGGING OUTSIDE THE LABORATORY

Light Emitting Diodes (LED's)

An LED is a semiconductor diode designed to emit light when forward biased. Since it is a diode, voltage polarity must be observed in its use (reverse voltage of 0 to 5 volts will simply leave the LED off). The logic kits contain red LED's.

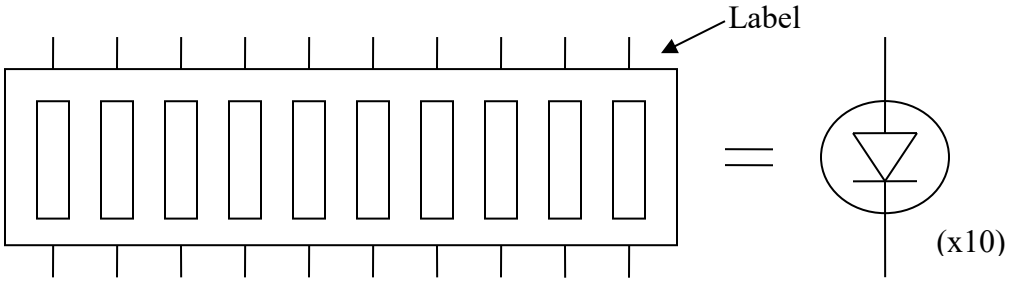
LED's are ideal as logic signal indicators because zero current shuts them off and forward current turns them on. A typical indicator circuit is shown below. The current limiting resistor is necessary to prevent the LED from clamping (affecting) the signal voltage.



Logic Signal	
0	ON
1	OFF

Normally people like light to be ON for logic “1” and OFF for logic “0”. In this case the complement of the function to be displayed is connected as SIGNAL. Each LED circuit sources ≈ 11 mA on low input. An inverter circuit like the 7404 can drive this circuit if its output is connected to SIGNAL. The 7404 can sink 16mA maximum. If more brightness is desired, appropriate resistors (e.g. 270Ω, but no smaller than 100Ω) and driving circuitry can be found or designed. For higher currents, an Open-Collector inverter like 7406 should be used in place of 7404. The 7406 can sink a maximum of 30mA and may drive this LED circuit and 20 TTL loads besides. It's easy to burn the driver IC, usually by making it sink more than 30 mA. If we have two or more LEDs to monitor several signals, why is it bad practice to share resistors?

LED Polarity

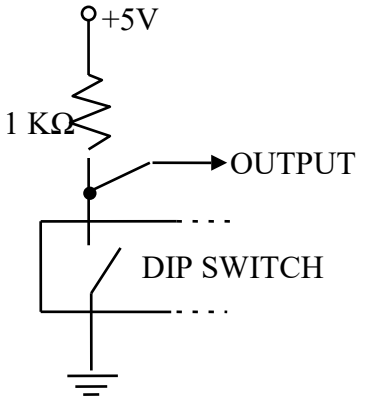


Note that the LEDs at the lab station use this same circuit with an inverter (74240) driving SIGNAL. The LEDs light when the connected signal (input of inverter) is **high** and turn off when the connected signal is **low**.

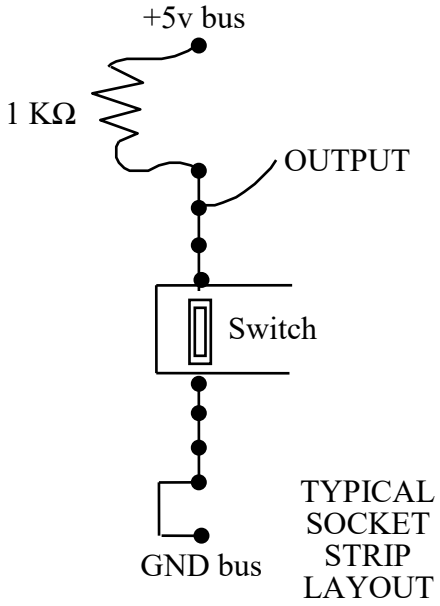
Toggle Switches (8-switch DIPs)

The DIP switch contains 8 single-pole single-throw switches mounted in a 16 pin DIP. The "ON" position corresponds to a closed switch and the "OFF" position an open switch. Some switches are marked CLOSED AND OPEN.

One use of the DIP switch will be to form a switch "register". A 1-bit slice of such a "register" is shown below (note that the pull up resistor is **required** to prevent the logic input from floating while the switch is open).



SWITCH POSITION	OUTPUT
ON-CLOSED	0 V
OFF-OPEN	+5 V



Note that the switches are numbered on each DIP package. It is convenient to refer to them and use them by number.

Switches may be toggled with a well-filed fingernail or a pencil point. Switches of this type (employing contact closures), bounce when they are closed or opened. The logic which receives switch outputs must be designed so that the bounces have no ill effect on its operation.

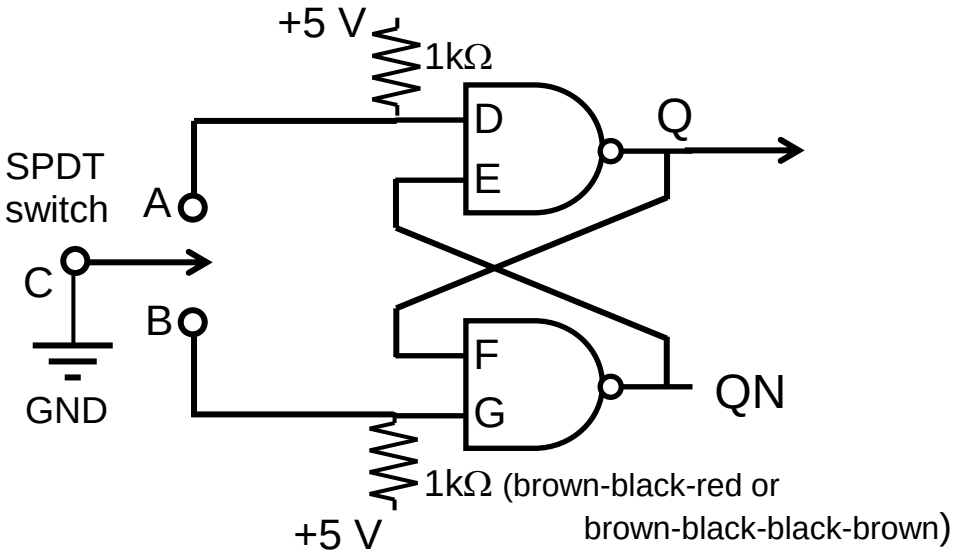
Note: If you choose to use these LEDs and switches, you need not reconnect to use the lab station LEDs and switches when you come to the lab. It may, however, be more convenient to do so.

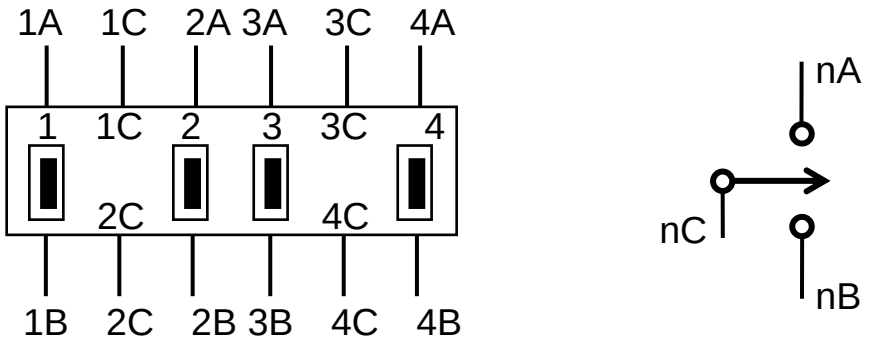
Contact Bounce:

The incredible speed of TTL causes some interesting problems when interfacing circuits to the slow world around them. For example, when a mechanical switch is closed, the electrical contacts inside the switch are violently slammed together. The force of the impact of the two contacts is often sufficient to cause the contacts to separate momentarily, and then come together. The momentary separation of the electrical contacts is called 'contact bounce.' When a switch is flipped, the contacts may bounce several times before finally coming to rest. The contacts will be stable within milliseconds of first contact: essentially zero time to the human observer, but a long period for a TTL circuit. If a switch is used to clock a counter circuit, the counter may advance several times per flip of the switch.

The 'de-bouncer' circuit shown in Figure 17 eliminates problems caused by switch contact bounce by insuring a clean transition from a logic zero to logic one (or logic one to logic zero) when the switch is thrown. The details of how and why the debouncer circuit works is left as a post-lab exercise. (Hint: The $\overline{S}$ - $\overline{R}$ cross-coupled NAND latch has its output stable when $\overline{S}$ = $\overline{R}$ =1)

Note the use of the pull-up resistors in the de-bouncer circuit. When the switch is in position 'A,' the input 'D' of the first NAND gate is tied directly to ground (a logic 0). With the switch in position 'B,' NAND gate input 'D' is not connected to ground but is pulled to a logic 1 by the pull-up resistor. Without the pull-up resistor, input 'D' would be left floating (unconnected) when the switch is in position 'B.' REMEMBER - All inputs must be held at some logic level!





QUAD SPDT Switch

Figure 17. A Debouncing Circuit for an SPDT Switch

ECE 385

EXPERIMENT #1

Introductory Experiment

I. OBJECTIVE

This experiment is intended primarily to be an introduction to the ECE 385 Laboratory and equipment. The equipment introduced in this experiment includes the student lab kit, the lab station I/O boards, the Agilent MSO6032A oscilloscope and the Hewlett Packard 33120A pulse generator.

II. INTRODUCTION

A discussion of the theory behind Experiment 1 can be found in the General Guide, parts IV (Design Techniques, GG.22) and VI (Debugging Outside the Lab, GG.29). If you have the opportunity, you should read these sections before coming to lab. If not, you should read these sections after lab. Though the lab portion of Experiment 1 may be completed without reading these sections, understanding of these concepts is critical to both completing the Experiment 1 report and succeeding in the course in general.

III. PRE-LAB

A. Complete the design of the circuit shown in the General Guide, Figure 16 (GG.25) (add pin numbers, chip placement, and I/O assignments). You will be able to find the pin assignments for the chips in the Data Sheets provided on the Lab 1 webpage on the course website. Document the circuit as a circuit logic diagram (with pin numbers assigned and ports interconnected) and in a layout sheet (blank sheets provided at the back of the lab manual.) Drive the three inputs A, B, and C from I/O board switches. **The I/O board switches are debounced – no modifications are necessary**. Also, display the three inputs and the output on LEDs. Note that the I/O boards contain LEDs and drivers; to display a signal, simply hook it directly to the appropriate I/O board ribbon cable connection. Not all groups may observe static hazards (why?) If you do not observe a static hazard, chain an odd number of inverters together in place of the single inverter from Figure 16 **or** add a small

capacitor to the output of the inverter until you observe a glitch. Why does the hazard appear when you do this?

B. Redesign the circuit of part A to eliminate all static-1 hazards (glitches) at the output. (For a discussion on glitches, see General Guide part IV – "Design Techniques", "Delays and Glitches" section GG.22-25). Again document the circuit as a separate logic diagram and in the same layout sheet (only one layout sheet per lab, but you should include all circuits.) Build the circuit, drive all inputs from I/O board switches, and display the output on an LED. This circuit should be built independently of the circuit from part A; the two may share inputs, but both should be built from scratch.

Demo Points Breakdown:

The first lab does not have any in-lab demo points. However, results will need to be contained within the lab report.

IV. LAB

1. Unit test each integrated circuit (chip) required for this lab. Since all the chips used in this lab are combinational logic, you may simply verify that each gate within the chip individually implements the truth table as expected. You should also verify that the output for each chip is within the expected noise margin (check the data-sheet for specifics). This means that there should be no ambiguous voltage levels (those which are not clearly logical 0 or logical 1). Ambiguous voltages indicate either a wiring problem (e.g. a bus conflict) or a faulty chip. This should be standard practice with all TTL labs.

2. Test the circuit of part A of the pre-lab by applying all possible input combinations using the three switches. Complete a truth table of the output as a function of the three inputs.

3. Disconnect the switch going to input B of the circuit in part A. After setting both inputs A and C high, drive input B with a 1 MHz, 0 to 5 volt square wave from the pulse generator. **WARNING: You should always check the output of the pulse generator on the oscilloscope before connecting it to the circuit. Doing this will greatly reduce the chance of your circuit being smoked by mistake.**

Display input B on one channel of the scope and the circuit output on the other channel. Note that displaying the input is necessary to provide a frame of reference for the output waveform. With the input displayed, it can be seen how the output of the circuit is changing in response to various signals. Save the waveforms (using a USB or Agilent's screen capturing software on the PC); pay particular attention to the relation between the two waves and the timing. Make sure that voltage and time scales are clearly indicated.

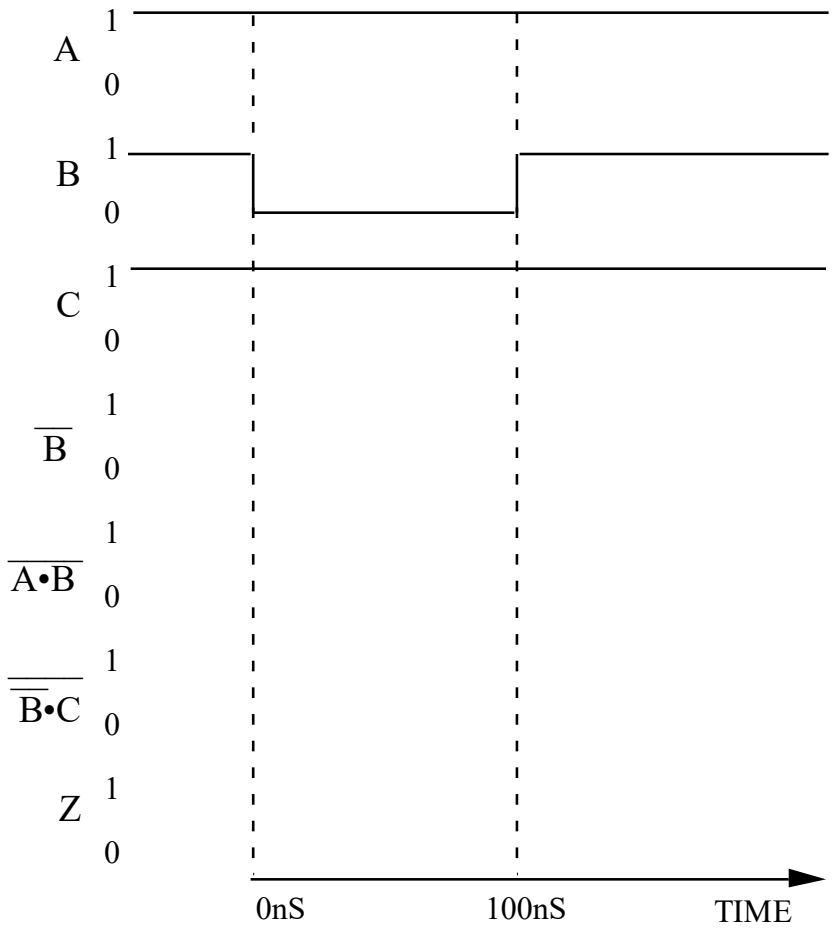
4. Using the three input switches, test the circuit in part B of the pre-lab. Complete a truth table of the output. Does it respond like the circuit of part A? Next, disconnect the switch from input B and hook the pulse generator to input B. With inputs A and C high, observe input B and the circuit output on the oscilloscope. Describe and save the output and explain any differences between it and the results obtained in part 2. Consider the following question and explain: for the circuit of part A of the pre-lab, at which edge (rising/falling) of the input B are we more likely to observe a glitch at the output?

V. POST-LAB

1.) Given that the guaranteed minimum propagation delay of a 7400 is 0ns and that its guaranteed maximum delay time is 20ns, complete the timing diagram below for the circuit of part A. (See GG.23 if you are not sure how to proceed.)

1.4

TIMING DIAGRAM



How long does it take the output Z to stabilize on the falling edge of B (in ns)? How long does it take on the rising edge (in ns)? Are there any potential glitches in the output, Z? If so, explain what makes these glitches occur.

2.) Explain how and why the debouncer circuit given in General Guide Figure 17 (GG.32) works. Specifically, what makes it behave like a switch and how the ill effect of mechanical contact bounces is eliminated?

VI. REPORT

For lab report, you should hand in the following:

- An introduction;
- Written description of the operation of circuits from both parts of the pre-lab;

- One (1) component layout sheet, with the package layout of all circuits (DO NOT draw the interconnections! Refer to GG.20 for the proper documentation);
- Circuit diagrams for all circuits (pre-lab part A and B are two separate circuits);
- Documentation from all parts of the lab. This includes but not limited to a truth table for the circuit of pre-lab part A, answers to the questions posed in the pre-lab, a truth table for the circuit of pre-lab part B, an oscilloscope printout from lab part 2, and an oscilloscope printout for lab part 3;
- Answers to all post-lab questions;
- Answers to questions from the General Guide (GG.6, GG.29);
- A conclusion regarding what worked and what didn't, with explanations of any possible causes and the potential remedies.

Note: This lab report is an **individual report**, not a group report. Each person in the lab must hand in his/her own lab report; lab partners can work together but should not hand in copies of the same work. This means that oscilloscope traces may be shared, but answers to the questions should be in each partner's own words. Combined lab reports will begin at Experiment #2.

VII. HOMEWORK

Design a portable I/O board using circuits like those described in the General Guide, section VI (pages GG.29-32). This board will be used for debugging circuits outside of the laboratory and should therefore be built as carefully as possible. The board should include at least two (2) debounced switches (the four SPDT switches are mounted in a Dual Inline Package (DIP)) and two (2) LEDs. Drive the LEDs using the 7404 or 7406 open collector hex inverter chip from the lab kit. The 7406 is preferred because you will unlikely use open collector outputs elsewhere in your designs. Remember to include a separate current limiting resistor for each of the LED circuits! The I/O board must fit on a single socket strip; equivalent to one of the five columns (A, B, C, D, or E) on the lab kit protoboard (see GG.10) as you will not be required to disassemble this portion of the protoboard in between labs.

Begin building the portable I/O board as soon as possible. It is easier to test the board using the facilities of the lab, so proceed as far as possible on Experiment #2 in the remaining lab time.

ECE 385

EXPERIMENT #2

Data Storage

I. OBJECTIVE

In this experiment, you will design and construct a simple 2-bit, four-word shift-register storage unit.

II. INTRODUCTION

Conceptually, random access memory (RAM) is a storage device arranged as a set of binary words that can be individually identified and accessed using unique addresses (see Figure 1).

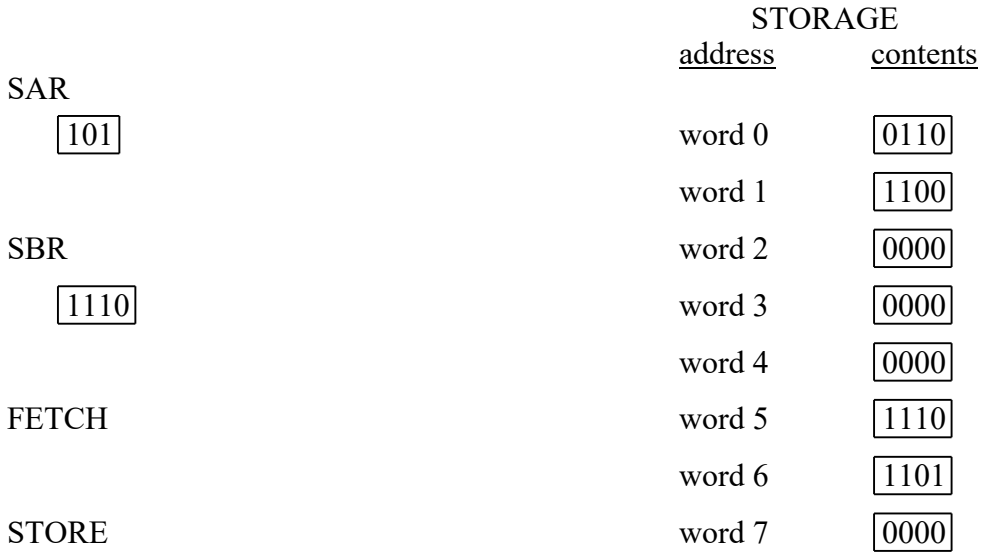


Figure 1: An Eight-word Storage Unit Using 4-Bit Words

To fetch a word from storage, the unique word address is placed in the Storage Address Register (SAR) and a FETCH signal is sent. The binary string or a content of the specified word appears in the Storage Buffer Register (SBR) a short time later (exactly how much later depends upon the technology used for the storage).

To store a word into storage the unique word address is placed in the SAR, the binary data to be stored is placed in the SBR, and a STORE signal is sent. The binary data in the SBR is stored in the word whose address is specified in the SAR. The previous contents of the word are destroyed by the STORE operation.

Cathode ray tubes, delay lines, and magnetic cores were once used for storage. In the 1970s, this was replaced by semiconductor RAMs, which are common now. You should be able to construct a storage unit from parallel-in/parallel-out shift registers, multiplexers, counters, and combinational logic.

One storage technique uses serial-in/serial-out (SISO) shift-registers shifting synchronously. A single 1024-bit SISO shift register could be used to provide 1024 words of storage, where each word is a single bit (e.g. a 1024x1 RAM). Note that with a 1024-bit SISO shift register, only the output of the rightmost flip-flop and only the input to the leftmost flip-flop are available. In theory, a shift register can be built at much lower cost compared to a RAM. This is because there are far fewer pins and interconnections in shift register than in RAM. In addition, the storage cell of a shift register could be very simple, a capacitor for example. The shift operation is simply moving charge from one capacitor to the neighboring capacitor. Such shift registers are called Charge Coupled Devices (CCDs). Today, CCDs are primarily used in imaging applications, such as in digital cameras. The charge in a cell slowly decays and therefore must be refreshed before it is lost. For this reason a SISO memory based on CCDs must be continuously shifted to keep the information from being lost.

Words larger than a single bit can be constructed by using more 1024-bit shift-registers clocked synchronously. Typically, 16 such SISO shift registers would be used to construct 1024 words of storage, where each word is 16 bits long. More generally, an n-bit, m-word shift-register storage consists of n m-bit shift-registers shifting together (see Figure 2).

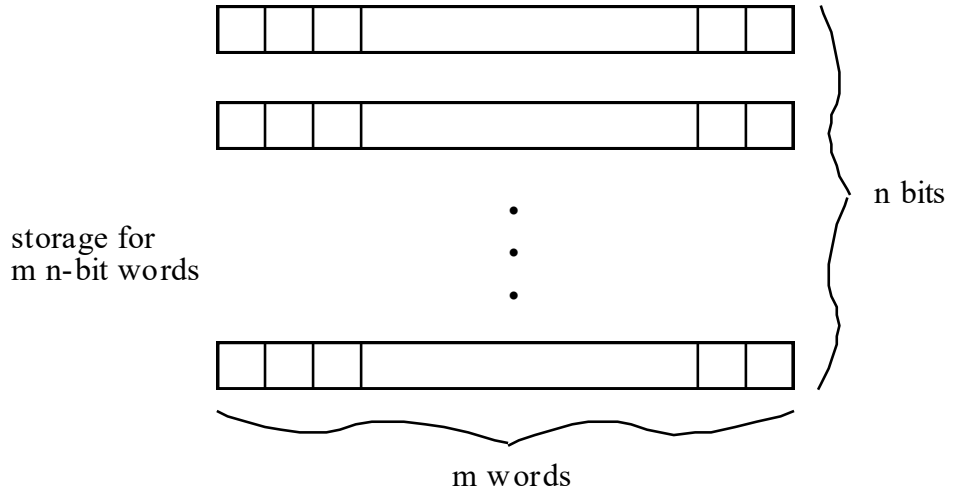


Figure 2: Configuration of a Shift-register Storage

As mentioned earlier, an alternative to the above storage devices are those devices that are built with "static" logic elements (SRAM). This is a setup where the storage device can retain data so long as a specified supply voltage is maintained. These SRAM chips are readily available from a number of manufacturers with varying features and parameters.

III. PRE-LAB

A. Design, document, and build a 2-bit four-word shift-register storage unit using two 74LS194 shift-registers without using the parallel load or parallel output capability. Of the 74LS194 data (non-control) inputs and outputs, you may use only the serial input and the rightmost (Qd) output. For the purposes of this experiment, imagine that the maximum and minimum clock period is specified as 1 millisecond for the 74LS194. The registers must be shifted on each clock pulse. The clock must run continuously – do **not** gate the clock (this is bad practice in digital design, why?) You **may** use clock enable or inhibit pins on the chips.

Signal Definitions:

LDSBR	When LDSBR is high, the SBR is loaded with the data word DIN1, DIN0.

FETCH	When FETCH is high, the value in the data word specified by the SAR is read into the SBR.
STORE	When STORE is high, the value in the SBR is stored into the word specified by the SAR.
SBR1, SBR0	The data word in the SBR; either the most recently fetched data word or a data word loaded from switches (note that when none of the LDSBR/FETCH/STORE switches is set, SBR should maintain the data in it)
SAR1, SAR0	The address, in the SAR, of a word in the storage
DIN1, DIN0	Data word to be loaded into SBR for storing into storage

Use flip-flops for the SBR. DIN1, DIN0, SAR1, and SAR0 should be obtained from switches. FETCH, STORE and LDSBR should also be obtained from switches. Display SBR1 and SBR0 on LEDs. You may also wish to include the display of other signals in your design for convenience when debugging your circuit. You can assume that only one of the FETCH/STORE/LDSBR switches will be set at any given time. Do not combine the FETCH/STORE switches!

To design the shift-register storage unit, we first need to look at the required specs. The most crucial requirement is for the shift registers to shift continuously, while using the serial input and output to store and fetch the data. We can break down our circuit operation into four operations: *load*, *read*, *write*, and *do nothing*. Let's first imagine the scenario where the circuit is turned on, but we are neither loading, reading nor writing. This is the most common state of the circuit, where no action is taken from the user – *do nothing*. Our requirements dictate that the shift registers must continuously shift, where any potentially stored data will be shifted out of the registers and into the void. To prevent losing any data, we will need to redirect the data shifting out of the registers back by connecting the serial output of the registers to their serial input, where the stored data will now be looping continuously in the shift registers. However, during a *write* operation, we do want to replace the old data with new data. To serve both purposes, a 2-to-1 multiplexer (MUX) can be placed at the serial input of the shift registers, taking either the new data or the old data depending on the current operation.

The rest of the circuit operation hinges upon the SBR, which serves two purposes: loading new data from DIN during a *load* operation and reading from the shift

register during a *read* operation. Note that the SBR must also behave as a register (that is, be able to synchronously maintain its previous contents). There are two ways to approach this. In the first, a simple D-flip flop may be used to implement the SBR. The input of the SBR takes in these three different choices by using a 3-to-1 MUX (or a 4-to-1 MUX with one input ignored), where one of the inputs is used to loop back when the SBR needs to maintain previous data. Alternatively, you may use a register chip which has a load enable to implement the SBR. This allows you to use a 2-to-1 MUX here instead.

But what is the ‘current operation’ at any given moment? Surely the desired operation is dictated by the user using the switches, but the inputs alone is not sufficient to tell each part of the circuit what to do. For example, if you would like to read from a specific address in the shift registers, you would first set the SAR to the specific address then you would hit the FETCH switch. But since the shift registers are constantly shifting data in and out of their serial ports, when exactly do you load the data into the SBR? How do you exactly tell what input the MUX should choose from? To solve the various problems associated with controlling and timing, it is generally not a good idea to use the inputs to directly control the various circuit components. Rather, it is almost always desired to have a centralized control logic that takes in all the inputs, process the request, and sends out various signals to control the circuit components. Figure 3 shows a general block diagram for the proposed circuit design. The most common form of a control logic is a state machine, which we will discuss in the next experiment. In this experiment, we will improvise a simpler control logic based on the requirements of our specific circuit.

First, notice that our shift registers are four word long, that is, each data will take exactly four shifts/clock cycles to loop back to its original location. We can exploit this property by employing a 2-bit counter (four distinct values) to keep track of the internal data address, then use a comparator to match the internal address with the SAR. Note that since the register is always shifting, it is meaningless to indicate "absolute" storage addresses. Rather, all addresses are "relative." If you wish to store data X in address Y, you can write the data into a random cell Z when the internal data address from the 2-bit counter matches the SAR. This (previously arbitrary) cell Z will now be associated with the address Y. Later, when we wish to fetch from address Y, we wait for the internal counter to

match the SAR again - that is when cell Z once again becomes available for reading or writing. Another interpretation that might be useful is that the counter always keeps track of the address associated with data to be shifted out from serial output/into serial input of the shift register array at the **up-coming clock edge**. Note that to control the MUXs, the 'select' signals generated by the control logic must take into account of the input switches and the comparator output (to indicate if we are currently looking at the correct address for reading/writing).

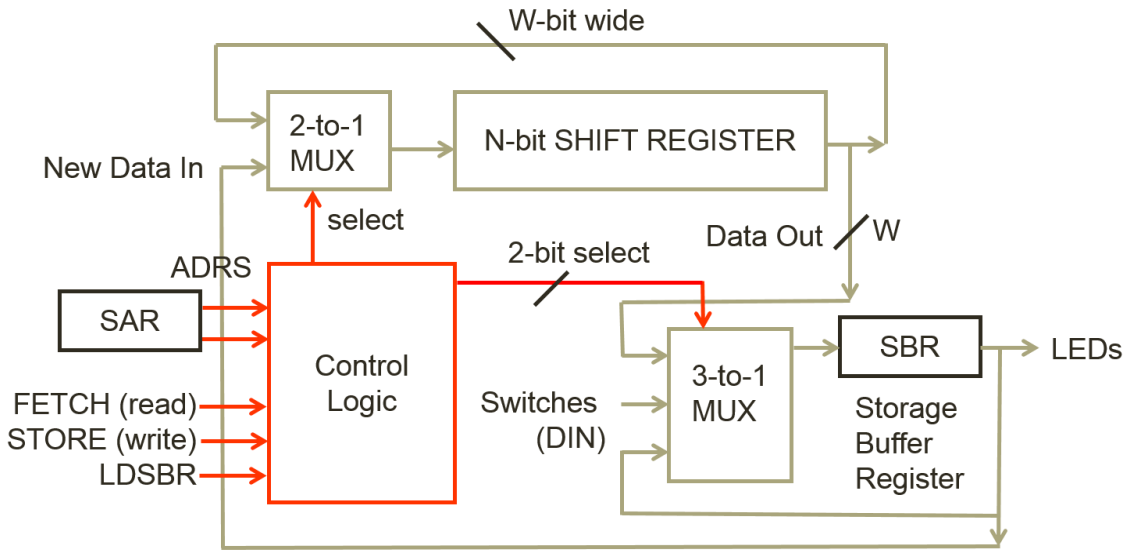


Figure 3: Block diagram of the shift-register storage unit.

Your pre-lab writeup should contain a written description of your circuit operation, a block diagram, operation of the controller, a logic diagram and layout documentation. Note that these materials only need to be turned in as part of your lab report, but you should keep in mind to generate them during the design process (rather than later).

HINT: Use the Pulse Generator to provide a basic clock. Continuously clock the shift-register and a counter that keeps track of which word is currently available. Use combinational circuitry (or 74LS85) to check for a match between the available word and the SAR.

B. Meet with your lab partner and wire up and test your design before coming to the lab. Use either the mini-switchbox circuit that you built at the end of Lab 1 (detailed in the General Guide) or attend an open lab session to test your circuit with the real

switchbox. Only the clock input needs to be de-bounced to strep through your circuit (why?).

Demo Points Breakdown:

1.0 point: When LDSBR is high, the data in DIN is loaded from the switches into SBR

1.0 point: When STORE is high, the contents of the SBR are stored into the location specified in SAR

3.0 points: When FETCH is high, the data word specified by the SAR is read into the SBR

Note: you may get 1 point of partial credit from these 3 for demonstrating that your shift register can shift right on each pulse of the clock. If you get the entire assignment working, you do not need to demonstrate this independently (since you may need to rewire a shift register to demo this).

IV. LAB

Finish testing and demonstrating your circuit to your TA. Correct the design if necessary and document your changes for the lab report.

Follow the Lab 2 demo information when debugging is completed.

V. POST-LAB

1) Your post-lab writeup (notes) should contain a corrected version of your pre-lab writeup and an explanation of any remaining problems in the operation of the circuits. This will aid the writing of your lab report.

2) Discuss with your lab partner and answer at least the following questions in your lab report:

- What are the performance implications of your shift register memory as compared to a standard SRAM of the same size?

- What are the implications of the different counters and shift register chips, what was your reasoning in choosing the parts you did?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below. Note that this is a group report, so only one partner will need to hand-in the report (decide who uploads the report and make sure they do it before the due date!)

1. Introduction
    a. Summarize what high-level function your circuit performs. How many words does your memory contain and what is the bit width of each word? The introduction should be approximately 3 - 5 sentences.
2. Operation of the memory circuit
    a. Describe how the addressing is implemented. When does the circuit commit a read or write from/to the input switches and output register respectively?
    b. Describe how a write operation is performed on the memory. Describe what switches you flip and in what order. Describe intuitively how the data flows through the circuit at each clock cycle.
    c. Describe how a read operation is performed on the memory. Describe what switches you flip and in what order. Describe intuitively how the data flows through the circuit at each clock cycle.
3. Written description and block diagram of memory circuit implementation
    a. High-level description
        i. Describe in words what components are necessary to perform the operations described in the written description of the memory circuit operation.
    b. Include a high-level block diagram like figure 3 in the lab manual. This diagram should contain components on the granularity of registers, muxes, and blocks and include few/no individual gates.
4. Control Unit
    a. Provide an intuitive written description of your control unit.
    b. Include a block diagram of your control unit. It is acceptable to turn in either:

        i. A single high-level block diagram, which contains the sub-components inside the control unit.
        ii. Two block diagrams, one like figure 3 in the lab manual and one containing only the inside of the control unit.
5. Design steps taken and detailed circuit schematic
    a. If you used k-maps or truth tables during design, include them here. (If you didn't need them, you don't need to include them).
    b. You do not need a state diagram for this lab.
    c. Written description of the design considerations taken (did you consider multiple implementations of the same circuit and the tradeoffs of each?)
    d. Detailed Circuit Schematic
6. This diagram should show all components used and their interconnections down to the logic gate level. Large schematics can be broken down into a hierarchy (for example, in the main diagram, the control logic can be shown as a box with inputs and outputs and a separate detailed diagram of the control logic can be placed below the main diagram). You may omit gate level representations of complex chips which you used (counters, etc.) and instead replace them with a block and all connections.
7. Component Layout Sheet
    a. Follow the guidelines in the general guide. Blank layout sheets as well as a sample completed layout sheet may be found in the general guide or course website. You may also use a computer tool for this portion.
8. Description of all bugs encountered, and corrective measures taken
9. Conclusion
    a. Summarize the lab in a few sentences
    b. Answers to all the post-lab questions, these may either be in a separate section or dispersed into the appropriate portions of the lab report.

ECE 385

EXPERIMENT #3

A Logic Processor

I. OBJECTIVE

In this experiment, you will design and build a bit-serial logic operation processor. The design will utilize two 4-bit shift registers, several multiplexers, and some type of counter. The circuit will be capable of calculating eight different functions and routing the results of those operations in four different ways. A finite state machine will be implemented to serve as the control unit of the circuit.

II. INTRODUCTION

In the experiment on storage, shift registers were used to store data by shifting a bit out of one end of the shift register and then shifting the same bit into the other end. However, when a write operation was performed, a new bit was shifted into the register. Taking this process one step further, we will modify, not one, but all the data bits in the register using a designated logical function. The circuit will provide the capability of bit-wise logical operations. We only want to perform the logical function once on each bit of the data, so we will also need to keep the data from circulating more than once through the circuit.

The operations that this circuit will perform are similar to the bit-wise logical functions provided in machine level programming. For example, your circuit will be able to perform a bit-wise AND of the two operands that are stored in the two registers (RegA and RegB) and be able to place the result in either of the registers, leaving the other register unchanged in the process. A bit-wise AND means that each bit of RegA is logically ANDed with the corresponding bit in RegB. So, if RegA = $a_3a_2a_1a_0$ and RegB = $b_3b_2b_1b_0$ then the destination register (RegA or RegB) will hold (a_3b_3), (a_2b_2), (a_1b_1), (a_0b_0) after the computation is complete. This is equivalent the machine code instruction of the form:

```
AND R0, R1, R0     /*  R0 and R1 = > R0  */
```

The other functions are OR, XOR, NAND, NOR, XNOR, CLR, SET, and SWAP. The complete set of functions and their corresponding control inputs are tabulated below (Table 1).

The block diagram of the circuit you will design for this experiment is shown below (Fig. 1). It includes

1) a **register unit** that contains two 4-bit registers, which we will refer to as **RegA** and **RegB**,

2) a **computation unit** that executes the desired logical computation,

3) a **routing unit** that routes the signals back to the register unit after computation, and

4) a **control unit** that generates control inputs to the register unit.

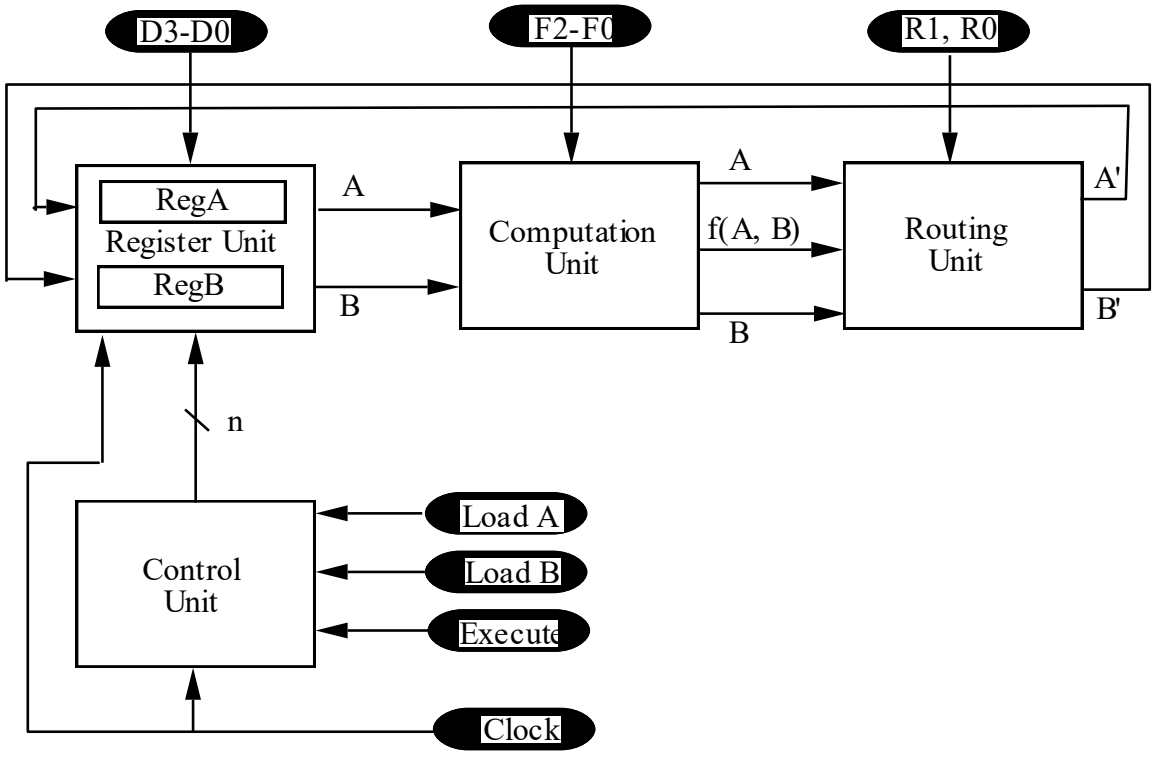


Figure 1: Block Diagram

Register Unit

The register unit will be made up of two 4-bit shift registers (7495A or 74LS194A) that will hold the values of Register A (RegA) and Register B (RegB). The contents of these registers should be displayed so that the contents before and after execution can be inspected. The control of these registers will be provided from the Control Unit while the serial input will be provided from the routing unit.

Computation Unit

The computation unit will accept as inputs the contents of RegA and RegB, and the function selection inputs F2, F1, F0. The unit will output the logical function f(A, B) specified by <F2, F1, F0> and will also output the A and B inputs unchanged. The three outputs will be fed to the Routing Unit.

Routing Unit

The routing unit will accept the A, B, and f(A, B) inputs and, based on the routing selection inputs R1, R0, will determine which signals to feed to the A' (new A) and B' (new B) outputs.

TABLE 1: Functions

Function Selection Inputs			Computation Unit Output
F2	**F1**	**F0**	**f(A, B)**
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R1	**R0**	**A***	**B***
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

3.4

Control Unit

The control unit will accept the following inputs: Load A, Load B, Execute, and the clock signal. The Load A and Load B inputs will perform parallel loads from the data input switches (D3-D0) into the A and B registers. Execute tells the control unit that the select switches and the register contents are ready for execution and that the control unit should begin the computation cycle. The control unit then shifts the register unit the required number of times and halts until the next execution is requested. Obviously, some type of mechanism to keep track of the shifts will be required. The clock input should be taken from the function generator to make the computation cycle appear to be instantaneous while also leaving the debugging capacity of single stepping.

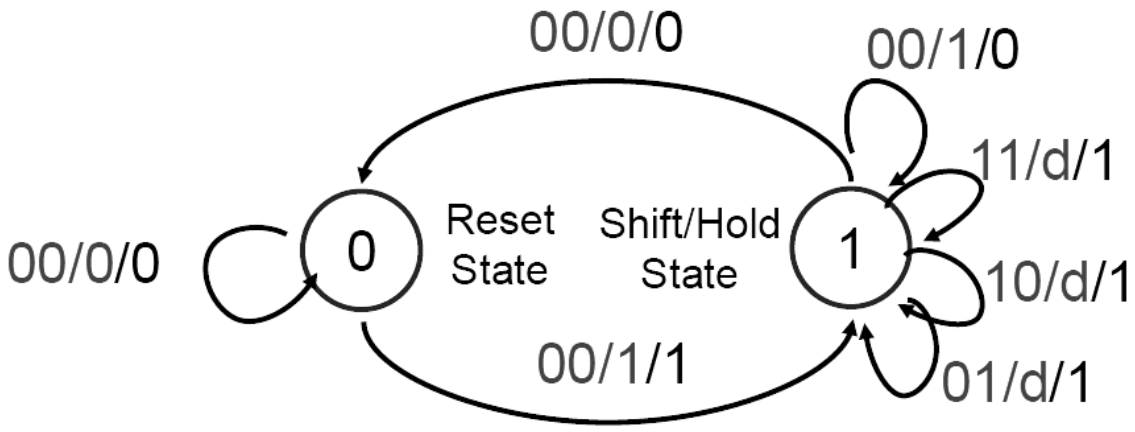


Figure 2: State Diagram

To accomplish this, a finite state machine is devised to control the operation of the register unit. There are two common state machine types: the Moore machine and the Mealy machine. During operation, both machines take in a set of inputs, transitions through a finite number of states, and output the relevant controls. The biggest difference between the two state machines is that the outputs of the Moore machine depend solely on the current state, each serving a specific output configuration, while the outputs of the Mealy machine depends on a combination of the current state and the current inputs. From this point of view, it is apparent that the Mealy machine will be able to achieve the same level of control by using fewer states than what's required by the Moore machine, which also makes the circuit implementation a little bit easier.

Table 1 shows an example of the Mealy machine for our control unit. The inputs of the Mealy machine are the ‘Execute’ switch, a single-bit state representation ‘Q’, and two-bit count ‘C1C0’. The ‘Execute’ switch dictates when the circuit should initiate the computation cycle. The single-bit state ‘Q’ split the Mealy machine into two states that serves distinct purposes – one is the reset/rest state, and the other one is the shift/halt state. And the count bits ‘C1C0’ are used to keep track of the number of shifts in the shift/halt state. Note that ‘C1C0’ represents a simplification, using a counter here reduces the number of states we must explicitly encode. The outputs of the mealy machine are the output signal ‘Reg. Shift’ (‘S’) which goes to the register unit, the next state ‘Q’, and the next count ‘C1C0’. Notice that the state descriptions are not very precise here, and this is the characteristics of the Mealy machine. Unlike the Moore machine where every state is directly linked to an operation and thus the purpose is clearly defined, the Mealy machine groups similar operations into a single state, then uses a combination of the current state and the current inputs to perform an operation.

To produce a state machine, you should follow the actual sequential operation of the circuit, where the state and the counts starts from (QC1C0=‘0000’), and the ‘Execute’ switch is held low (‘E’=’0’). As long as the ‘Execute’ switch remains low, the circuit stay in a rest state, where the register unit stays put and the next state and count also stay unchanged ($SQ^+C1^+C0^+$=‘0000’). However, at the immediate clock edge after the ‘Execute’ switch is flipped up (EQC1C0=‘1000’), the state machine moves to the shift/halt state, and sends out the signal to shift the registers and begins to increment the counter ($SQ^+C1^+C0^+$=‘1101’). The state machine in total should then carry out three additional shifts regardless of the condition of the ‘Execute’ switch. After the four shifts, the state machine will state in ($SQ^+C1^+C0^+$=‘0100’) if the ‘Execute’ switch remains high, or transitions back to ($SQ^+C1^+C0^+$=‘0000’) if the ‘Execute’ switch drops low. This completes one full cycle of bit-serial logic operation as the state machine comes back to where it had started and awaits for another full cycle of operation when the ‘Execute’ switch is flipped up again.

To transform the state machine into a physical circuit, we will next build a state transition table. Follow through the entire operation cycle to fill out as much of the state transition table as possible. You will then notice that not all combinations are valid. For example, if we are in the reset/rest state, it is not possible for our counts to hold any value other than ‘00’, and therefore we will never encounter, for example, (EQC1C0=‘0011’) during our circuit operation. If the combination is not valid, a ‘don’t care’ (‘d’) should be

placed in the outputs for easier implementation of the circuit (remember that it is ok to circle the K-map minterm over the ‘don’t cares’ without affecting the functionality of the circuit).

TABLE 1: Control unit state transition table using the Mealy state machine

Exec. Switch (‘E’)	Q	C1	C0	Reg. Shift (‘S’)	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

After the state transition table is made, we then proceed on producing the K-maps and circling the minterms. Since the transitioning of each of the four state machine components (S, Q, C1, C0) is dictated by four inputs (E, Q, C1, C0), you will need to convert the table to four K-maps and obtain the resulting circuits for each of the four components. Notice that while (Q, C1, C0) are internal components of the state machine, the only actual output ‘S’ will be used to control the shifting of the two registers. To adapt ‘S’ and ‘LoadA’, ‘LoadB’ to the two registers for the shift/load/halt operations, simple combinational logic will need to be devised.

Demo Points Breakdown:

1.0 point: Show correct loading of the A and B registers

1.0 point: Show the computation cycle is of the right length

1.0 point: Demonstrate the four routing operations

1.0 point: Demonstrate the eight function operations

1.0 point: Show the computation cycle completes even if the EXECUTE switch is returned to the low position mid-computation (requires a slow clock)

III. PRE-LAB

A. Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Sketch your circuit.

B. Explain how a modular design such as that presented above improves testability and cuts down development time.

C. Design, document and build the circuit described in Part II. Your circuit should be able to perform correctly all the functions listed. You may use either 7495A or 74LS194A chips for your shift registers. You will want to study each of the chips carefully before deciding on one or the other. Be sure to make your design as efficient as possible (there is more than one way to design this circuit).

A square wave from the Pulse Generator should be used as the basic system clock. Load A, Load B, Execute, D3-D0, R1, R0, and F2-F0 should be inputs from the switches. The control unit must be designed to perform the desired function once and only once each time the execute switch is flipped on. Results of the operation should be obtained even if the execute switch is flipped off in the middle of the computation cycle. You may only assume that the execute switch will remain high for at least one full clock period. Display the contents of Register A and Register B on LEDs. You may also want to include an LED that indicates when the computation cycle is complete for debugging purposes.

D. Work with your partner to wire-up the circuit.

IV. LAB

Follow the Lab 3 demo information on the course website.

V. POST-LAB

Document changes to your design and correct your Pre-Lab write-up, explaining any difficulties you had in debugging your circuit. Outline how the modular approach proposed in the pre-lab help you isolate design and wiring faults, be specific and give examples from your actual lab experience.

Make sure you report discusses the following:

Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1) Introduction
    a. Summarize what high-level function your circuit performs. What operations can the processor do? How many bits can it operate on? Etc. The introduction should be approximately 3 - 5 sentences.
    b. Answers to pre-lab questions.
2) Operation of the logic processor
    a. Describe the sequence of switches the user must flip to load data into the A and B registers.
    b. Describe the sequence of switches the user must flip to initiate a computation and routing operation.
3) Written description, block diagram and state machine diagram of logic processor

a. Written description: describe in words each block in the high-level diagram (a short paragraph for at least the register unit, computation unit, routing unit, and control unit).
b. Include a high-level block diagram. It's OK to use the one in the lab manual, provided it is modified as necessary to reflect what you implemented.
c. State Machine Diagram
    i. Explicitly state if you used a Mealy or Moore machine (or some hybrid)
    ii. Label each state (bubble)
        1. Give each state a meaningful name
        2. Specify the binary flip-flop values associated with each state.
        3. If you are using a Moore Machine, label the values of all meaningful outputs associated with each state.
    iii. Label each arc
        1. The combination of inputs which trigger the arc
        2. If you are using a Mealy Machine, label the values of all meaningful outputs associated with each arc.
        3. It is OK to label each bubble/arc with a single identifier and put the rest of the requested info in a table to reduce clutter.

4) Design steps taken and detailed circuit schematic diagram
    a. Written procedure of the design steps taken.
        i. If you used k-maps or truth tables during design, include them here. (If you didn't need them, you don't need to include them). K-maps are usually only helpful for creating the next-state logic in the control unit.
        ii. Written description of the design considerations taken (did you consider multiple implementations of the same circuit and the tradeoffs of each?)
    b. Detailed Circuit Schematic
        i. Draw a gate level schematic of your circuit. It is OK to use small standard blocks like MUXes, flip-flops, and shift registers, but custom blocks like your control unit must have a gate level schematic.
        ii. If the schematic becomes too large, components such as the control unit can be represented as black boxes on the top level schematic, and a detailed schematic of that component can be included below
5) Layout sheet
    a. Use the template in GG.20 or compose a similar layout sheet yourself.

   b. Items you DO NOT need to label in the component layout
      i. Vcc and GND pins
      ii. Unconnected pins (such as those on unused gates)
      iii. Mode pins tied to Vcc or GND (such as strobes and resets).
   c. Items you DO need to label
      i. Switch inputs
      ii. LED outputs
      iii. Intermediate logic signals (try to use the same naming conventions as your gate level and high-level diagrams.
      iv. Mode pins that are driven by a switch or another chip (such as strobes and resets).
6) Description of all bugs encountered, and corrective measures taken
7) Conclusion
   a. Summarize the lab in a few sentences
   b. Answer to all post-lab questions (they may be placed in conclusion or dispersed in more appropriate sections of the report).

ECE 385

EXPERIMENT #4

Introduction to SystemVerilog, FPGA, CAD, and 16-bit Adders

I. OBJECTIVE

In this experiment you will transition from breadboard TTL (transistor-transistor logic) elements to RTL (register-transfer level) design on an FPGA using SystemVerilog. You will come to understand the basic syntax and constructs of SystemVerilog, as well as acquire the basic skill required to operate Quartus Prime, a CAD tool for FPGA synthesis and simulation. Quartus Prime's performance analysis and optimization tools will be explored in the process of implementing three types of adders: a carry-ripple adder, a carry-lookahead adder, and a carry-select adder. This performance analysis and optimization will look at the various adders' area, power, and maximum operating frequencies.

II. INTRODUCTION

Please read the INTRODUCTION TO SYSTEMVERILOG AND TUTORIAL (IST. 1-26) and the INTRODUCTION TO QUARTUS PRIME (IQT. 1-40).

In addition to the standard synthesis and simulation capability, Quartus Prime provides a variety of compiler settings for the designer to tweak for the synthesis and compilation process. Depending on the settings the designer can gear the generated circuit to comply with some predefined constraints or performance criteria, such as the maximum operating frequency of the circuit, the maximum area of the circuit layout, or the maximum static or dynamic power consumed by the circuit.

During the synthesis and compilation process, Quartus Prime collects a variety of analysis data and display them in the generated Compilation Report. These data are important to the designer in the sense that the designer relies on these data to determine if his or her circuit has met the performance constraints. If the analysis result is far off from the performance criteria, the designer will most likely have to modify the circuit from the designing aspect of the circuit. On the other hand, if the analysis result is just slightly below the performance criteria, then the designer can use many of the built-in tools to optimize the circuit during the compilation process to meet the performance criteria.

Quartus Prime offers a variety of optimization tools, such as TimeQuest Timing Analyzer for the timing constraint, PowerPlay Power Analyzer for the power constraint, and a built-in placement fitter for the area constraint. Many of the optimization steps can be done by simply changing the various synthesis and compilation settings, as suggested by the Quartus Prime Optimization Advisors, some of the in-depth optimization and analysis can only be done by providing specific constraints to the analyzers.

In most industry practices, circuit implementation on FPGA is usually only a small portion of the entire design, where the circuit on FPGA will interface with external circuits through its inputs and outputs. These external circuits will have their own performance constraints which the FPGA circuit has to follow in order to be integrated. To incorporate these external constraints into the FPGA design, they are written into constraint files such as the Synopsys Design Constraint (SDC) format as input to the Quartus Prime Analyzers, where the analyzers will then be able to analyze and optimize the circuit based on the provided constraints.

To read more about the optimization process in Quartus Prime, please refer to Section III in Volume 2 of the Quartus Prime Standard Edition Handbook (currently v18.1, accessible on the Intel FPGA website) Chapter 10 gives a design optimization overview, Chapter 12-14 discuss timing, power and area optimization, respectively.

For this lab, we will consider the design of a binary adder. Binary adders are a key component of logic circuits. They are used not only in the arithmetic logic units (ALU) for data processing but are also used in other parts of a logic processor to calculate addresses and signal evaluations. An N-bit binary adder takes two binary numbers (A and B) of size N and a carry-in (C_{in}) as inputs, sum up the three values, and produces a sum (S) and a carry-out (C_{out}), as shown in Figure 1.

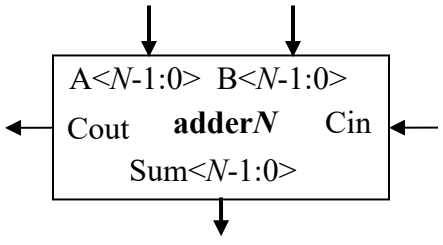


Figure 1: *N*-bit Binary Adder Block Diagram

Among the many different binary adder designs, the most straightforward one is the Carry-Ripple Adder (CRA). It is constructed using N full-adders. A full-adder is a single-bit version of the binary adder, where three binary bits (A, B and C_{in}) are inputted through a set of logic gates to produce a single-bit sum (S) and a single-bit carry-out (C_{out}), as shown in Figure 2. The N full-

adders are then linked together in series through the carry bits, forming an *N*-bit binary adder. When the binary inputs are provided, the full-adder of the least significant bit (LSB) will produce a sum (S_0) and a carry-out (C_1). The carry-out is fed to the carry-in of the second full-adder, which then produces a second sum (S_1) and a second carry-out (C_2). The process ripples through all *N* bits of the adder as shown in Figure 3, and settles when the full-adder of the most significant bit (MSB) outputs its sum (S_{N-1}) and carry-out (C_{out}).

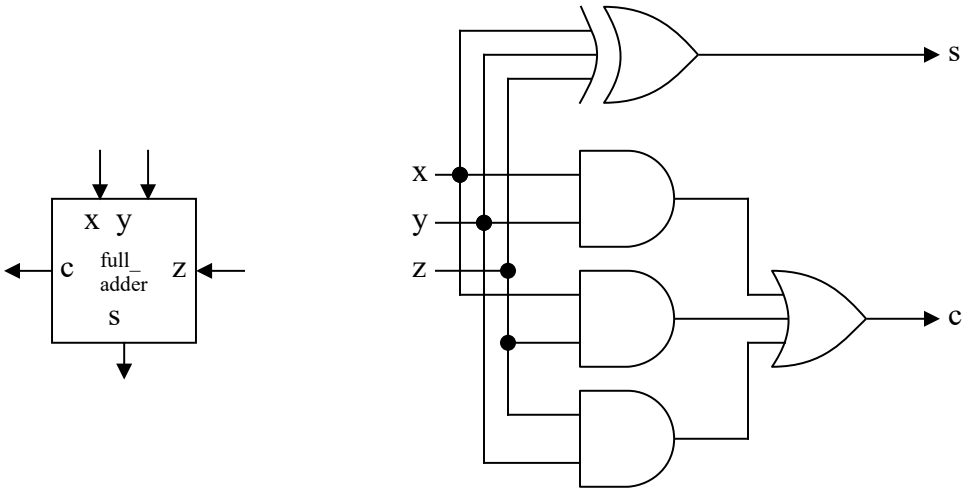


Figure 2: Full-Adder Block Diagram

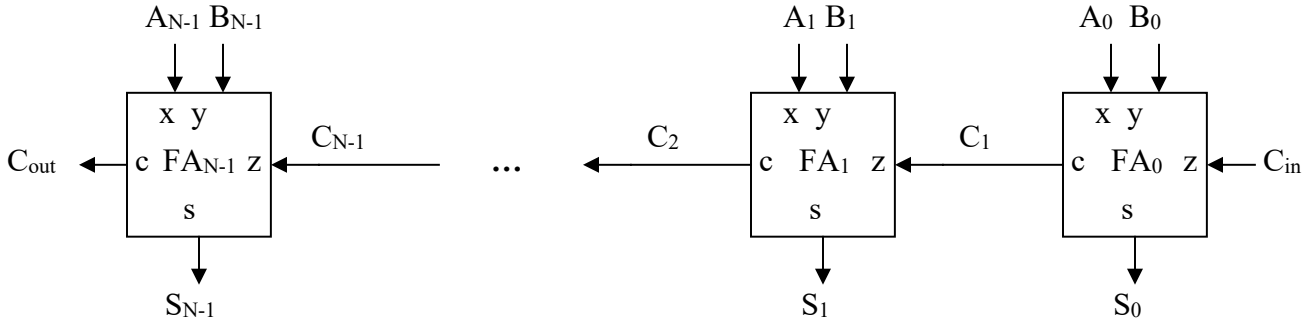


Figure 3: *N*-bit Carry-Ripple Adder Block Diagram

The CRA is simple in the design and straightforward to implement, but the long computation time is its drawback. Every full-adder has to wait for their lower-bit neighbor to produce a carry-out before it can correctly compute its sum and carry-out. This means that the propagation delay of the CRA increases with *N*. If one wishes to reduce the computation time, it is apparent that the computation of the carry-out bits has to be somehow parallelized. And this is precisely how a carry-lookahead adder operates.

Instead of waiting on the actual carry-in values, Carry-Lookahead Adder (CLA) uses the concept of *generating* (G) and *propagating* (P) logic. The concept is that every bit of the CLA makes predictions using its immediate available inputs (A and B), and predicts what its carry-out would be for any value of its carry-in. A carry-out is *generated* (G) if and only if both available inputs (A and B) are 1, regardless of the carry-in. The equation is $G(A,B) = A \cdot B$. On the other hand, a carry-out has the possibility of being *propagated* (P) if either A or B is 1, which is written as $P(A,B) = A \oplus B$. With P and G defined, the Boolean expression for the carry-out C_{i+1} giving a potential C_i is then $C_{i+1} = G_i + (P_i \cdot C_i)$. Notice that C_{i+1} can be expressed in terms of C_i which in turn can be expressed in terms of C_{i-1}. However, if C_{i+1} still depends on C_i, it will behave like a ripple adder without giving any gain in speed. Therefore, to avoid the slow rippling of the carry bits, the expression of C_{i+1} should be expanded and computed directly from P_i s, G_i s. For example,

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$
$$\ldots$$

In this way, the computation time of the CLA is much faster than that of the CRA, resulting in a higher operating frequency. The downside of the CLA is its additional logic gates, which increases both the area and power consumption of the adder.

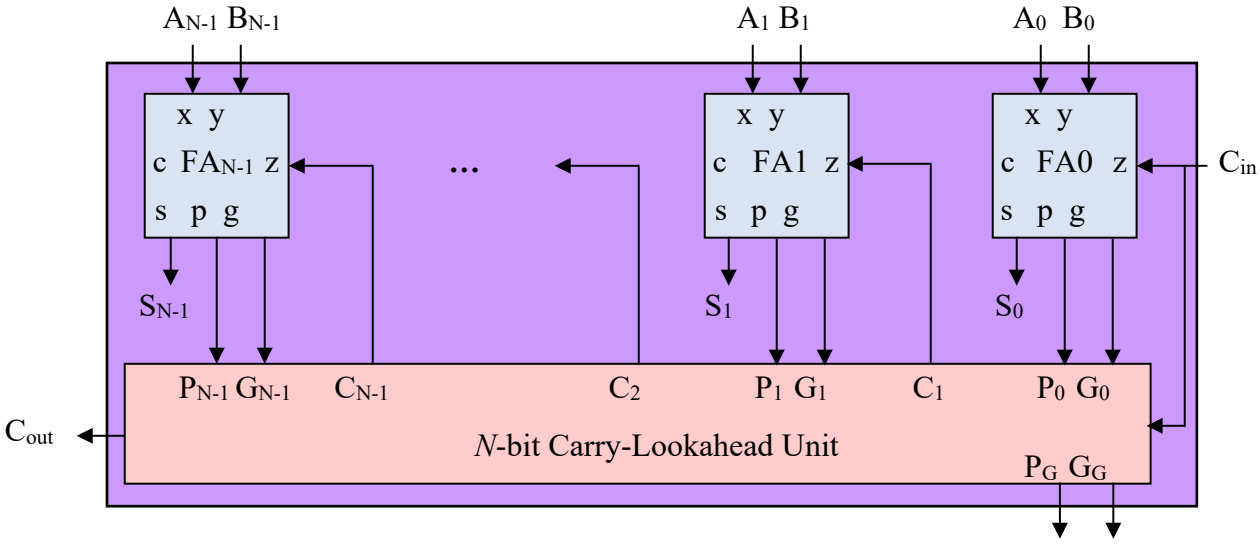


Figure 4: *N*-bit Carry-Lookahead Adder Block Diagram

To build an arbitrarily long *N*-bit CLA, one might be tempted to directly follow the above 'flat' approach. However, from the explicit expansion of C_i, you can find that the number of gates

involved for an increasing N will soon grow too large for the CLA to be practical. And thus, it is a common practice to first construct 4-bit CLAs, then use them to create a larger CLA in a hierarchical fashion. In this lab, the CLA should be implemented in 4x4-bit instead of 16-bit.

In the 4x4-bit hierarchical CLA design, the 16-bit inputs A and B are divided into groups of 4 bits. First, each group of 4 bits go through a 4-bit CLA, which is illustrated by Figure 4 with N=4. Note that the 4-bit CLA generates two additional output signals, the group propagate (P_G) and the group generate (G_G), with their logics being:

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

We will denote the P_Gs and G_Gs from these four 4-bit CLAs as P_{G0}, P_{G4}, P_{G8}, P_{G12}, and G_{G0}, G_{G4}, G_{G8}, G_{G12} from this point on.

Next, a tempting design is to cascade the four 4-bit CLAs by connecting the C_{out} from the previous 4-bit CLA to the C_{in} of the next 4-bit CLA, but in this way we will be trapped by the slow rippling of these carry bits again. Therefore, instead of using the C_{out} from the previous 4-bit CLA, we should generate the C_{in}s of the 4-bit CLAs using the P_Gs and G_Gs, as shown by the formulas below,

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$
...

Does this look familiar to you? Observe that this is the same as how we generated the carry bits within a 4-bit CLA. Therefore, we can directly take a copy of the 4-bit Carry-Lookahead Unit (CLU, red block in Figure 4) in the 4-bit CLA, but instead of the inputs coming from full adders, this time the inputs are the P_Gs and G_Gs from the 4-bit CLAs at the upper level. Figure 5 illustrates the resulting 4x4-bit hierarchical CLA.

This explains why this design is called *hierarchical*. If we add another layer to the hierarchy and use four 4x4-bit hierarchical CLAs and another 4-bit CLU, we can make a 4x4x4-bit hierarchical CLA, namely a 64-bit adder, without any issue of the slow rippling of the carry bits!

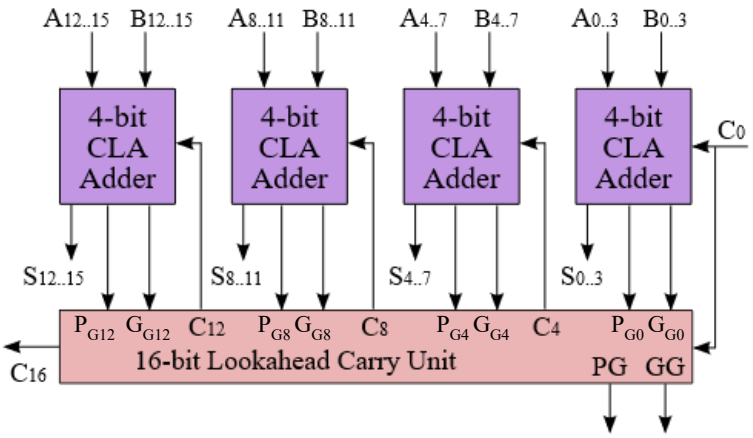


Figure 5: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

Carry-Select Adder (CSA) features another way to speed up the carry computation. It consists of two full adders (or CRAs if multiple bits are grouped) and a multiplexor. One adder computes the sum and carry-out based on the assumption that the carry-in is 0, and the other assumes that the carry-in is 1. In this way, both possible outcomes are pre-computed. Once the real carry-in arrives, the corresponding sum and carry-out is selected to be delivered to the next stage. By paying the price of almost twice the numbers of adders, we gain some speedup *(how exactly do we gain this speedup – we will discuss this in lecture, but you should make sure you understand and explain in your own words for your lab report!)*

In this lab, you are going to design a 16-bit CSA with 4x4-bit hierarchical structure as illustrated by Figure 5. For each group of 4-bit inputs, we use two CRAs to calculate two versions of the results, one with carry-in bit assumed to be 0 and the other to be 1. Note that the lowest significant group requires only one CRA, since its carry-in bit is directly available. Therefore, eventually the 16-bit CSA will contain seven 4-bit CRAs.

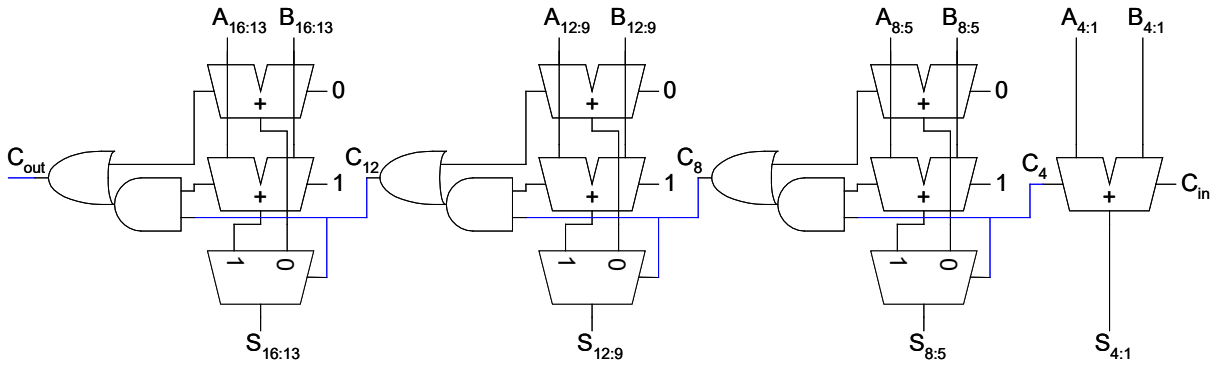


Figure 5: 16-bit Carry-Select Adder Block Diagram

Your circuits should have the following inputs and outputs:

Inputs

Clk, Reset, Load_B, Run – logic
SW – logic [15:0]

Outputs

CO – logic
Sum – logic[15:0]
Ahex0, Ahex1, Ahex2, Ahex3, Bhex0, Bhex1, Bhex2, Bhex3 – logic [6:0]

Internal Registers

A – logic [15:0]
B – logic [15:0]

SW[15:0] should come from on-board switches and its value should be displayed on Ahex0, Ahex1, Ahex2, and Ahex3 as a four-digit hex number. When Load_B is pressed, the registers B[15:0] should load the values of SW[15:0] to serve as *B*, one of the numbers to be added, and B[15:0] should be displayed on Bhex0, Bhex1, Bhex2, and Bhex3. At other times, the registers A[15:0] constantly load the values of SW[15:0] which serve as *A*, the other number to be added. The value of Sum[15:0] should be displayed on red LEDs (LEDR[15:0]), and CO should be displayed on LEDG[8] to indicate overflow. When Run is pressed, Sum[15:0] and CO should be updated with the result of adding SW[15:0] (*A*) and the old B[15:0] (*B*). Reset should clear all the registers. To achieve optimal speed and resource usage balance, the CLA and CSA will also need to be built in a hierarchical fashion. In this lab, they should be implemented in 4x4-bit instead of 16-bit. Also, for this lab, Cin may be assumed to be 0.

A test platform is required to demo your adders as there are not enough switches on the DE2-115 board. This platform is provided in the included Lab 4 files on the website, and it should be clear where to place your code for the three types of adders you will design. Registers A and B store the operands to be added, depending on whether Load_B is pressed (register A is continuously loaded from the switches on every cycle). Upon pressing the 'Run' button, the state machine will load the resulting sum (A+B) into a 16-bit output register to display. The load and run operation will be executed only once when the Load_B or run button is pressed each time, respectively. The circuit should be able to run multiple times without resetting the circuit before each operation.

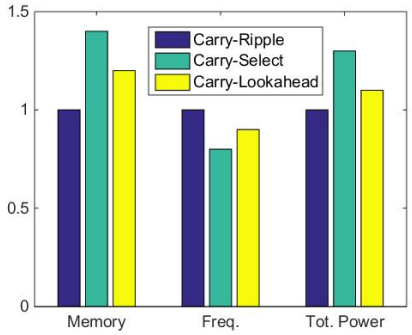
III. PRE-LAB

A. Complete the bit-serial logic processor exercise from the Introduction to SystemVerilog and Tutorial (IQT. 1-40). Include a copy of the generated diagram from Quartus of the 8-bit logic processor and the simulation waveform (with annotations) in your Lab 4 lab report.

B. Design, document, and implement a 16-bit carry-ripple adder, a 16-bit carry-lookahead adder, and a 16-bit carry-select adder in SystemVerilog. Use the provided code (from the website) as a testing framework.

C. Document design analysis for the three adders in the table below. Plot out the data from the table for comparison studies. Normalize the data across the three adders with the carry-ripple adder. When normalizing, choose data from one the carry-ripple adder as the baseline, and then divide the other two with the baseline number. Say, you got 20 from carry-ripple, 21 from carry-select, and 23 from carry-lookahead, the numbers after normalization becomes 20/20=1.0, 21/20=1.05, 23/20=1.15, respectively. The resulting plot should resemble the one below (the plot below does not use real data).

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)			
Frequency			
Total Power			



You will need to bring the following to the lab:

1. Your code for the 8-bit processor in a Quartus Prime project, ready to synthesize. You can bring the code to the lab using a USB storage device, source repository, any other method.

2. Your code for the 3, 16-bit adders with a project ready to synthesize and test on the FPGA board, be prepared to show your TA each adder's code to verify they are indeed performing according to design.

3. A block diagram for the 8-bit processor (or a project file which will generate the block diagram) to verify that you have completed the tutorial.

Demo Points Breakdown:

1.0 point: Functional simulation completed successfully for the 8-bit serial processor (annotations necessary)

1.0 point: RTL block diagram of the 8-bit logic processor extended from 4-bits. This can be automatically generated using Quartus.

1.0 point: Correct operation of the Carry-Ripple Adder on the DE2 board

1.0 point: Correct operation of the Carry-Lookahead Adder on the DE2 board using a 4x4 **hierarchical design** (TA's will look at code)

1.0 point: Correct operation of the Carry-Select Adder on the DE2 board using a **4x4 hierarchical design** (TA's will look at code)

IV. LAB

Follow the Lab 4 demo information on the course website.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Run	PIN_R24	On-Board Push Button (KEY3)
LoadB	PIN_M21	On-Board Push Button (KEY1)
Reset	PIN_M23	On-Board Push Button (KEY0)
SW[0]	PIN_AB28	On-board slider switch (SW0)
SW[1]	PIN_AC28	On-board slider switch (SW1)
SW[2]	PIN_AC27	On-board slider switch (SW2)
SW[3]	PIN_AD27	On-board slider switch (SW3)
SW[4]	PIN_AB27	On-board slider switch (SW4)
SW[5]	PIN_AC26	On-board slider switch (SW5)
SW[6]	PIN_AD26	On-board slider switch (SW6)
SW[7]	PIN_AB26	On-board slider switch (SW7)
SW[8]	PIN_AC25	On-board slider switch (SW8)
SW[9]	PIN_AB25	On-board slider switch (SW9)
SW[10]	PIN_AC24	On-board slider switch (SW10)
SW[11]	PIN_AB24	On-board slider switch (SW11)
SW[12]	PIN_AB23	On-board slider switch (SW12)
SW[13]	PIN_AA24	On-board slider switch (SW13)
SW[14]	PIN_AA23	On-board slider switch (SW14)
SW[15]	PIN_AA22	On-board slider switch (SW15)
Sum[0]	PIN_G19	On-Board LED (LEDR0)
Sum[1]	PIN_F19	On-Board LED (LEDR1)
Sum[2]	PIN_E19	On-Board LED (LEDR2)
Sum[3]	PIN_F21	On-Board LED (LEDR3)
Sum[4]	PIN_F18	On-Board LED (LEDR4)
Sum[5]	PIN_E18	On-Board LED (LEDR5)

4.10

Sum[6]	PIN_J19	On-Board LED (LEDR6)
Sum[7]	PIN_H19	On-Board LED (LEDR7)
Sum[8]	PIN_J17	On-Board LED (LEDR8)
Sum[9]	PIN_G17	On-Board LED (LEDR9)
Sum[10]	PIN_J15	On-Board LED (LEDR10)
Sum[11]	PIN_H16	On-Board LED (LEDR11)
Sum[12]	PIN_J16	On-Board LED (LEDR12)
Sum[13]	PIN_H17	On-Board LED (LEDR13)
Sum[14]	PIN_F15	On-Board LED (LEDR14)
Sum[15]	PIN_G15	On-Board LED (LEDR15)
Ahex0[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
Ahex0[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
Ahex0[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
Ahex0[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
Ahex0[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
Ahex0[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
Ahex0[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
Ahex1[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
Ahex1[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
Ahex1[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])
Ahex1[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
Ahex1[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
Ahex1[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
Ahex1[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])
Ahex2[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
Ahex2[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
Ahex2[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
Ahex2[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
Ahex2[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
Ahex2[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
Ahex2[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
Ahex3[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
Ahex3[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
Ahex3[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
Ahex3[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
Ahex3[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
Ahex3[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
Ahex3[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
Bhex0[0]	PIN_AB19	On-Board seven-segment display segment (HEX4[0])
Bhex0[1]	PIN_AA19	On-Board seven-segment display segment (HEX4[1])
Bhex0[2]	PIN_AG21	On-Board seven-segment display segment (HEX4[2])
Bhex0[3]	PIN_AH21	On-Board seven-segment display segment (HEX4[3])
Bhex0[4]	PIN_AE19	On-Board seven-segment display segment (HEX4[4])
Bhex0[5]	PIN_AF19	On-Board seven-segment display segment (HEX4[5])
Bhex0[6]	PIN_AE18	On-Board seven-segment display segment (HEX4[6])
Bhex1[0]	PIN_AD18	On-Board seven-segment display segment (HEX5[0])
Bhex1[1]	PIN_AC18	On-Board seven-segment display segment (HEX5[1])
Bhex1[2]	PIN_AB18	On-Board seven-segment display segment (HEX5[2])
Bhex1[3]	PIN_AH19	On-Board seven-segment display segment (HEX5[3])
Bhex1[4]	PIN_AG19	On-Board seven-segment display segment (HEX5[4])
Bhex1[5]	PIN_AF18	On-Board seven-segment display segment (HEX5[5])
Bhex1[6]	PIN_AH18	On-Board seven-segment display segment (HEX5[6])
Bhex2[0]	PIN_AA17	On-Board seven-segment display segment (HEX6[0])
Bhex2[1]	PIN_AB16	On-Board seven-segment display segment (HEX6[1])

Bhex2[2]	PIN_AA16	On-Board seven-segment display segment (HEX6[2])
Bhex2[3]	PIN_AB17	On-Board seven-segment display segment (HEX6[3])
Bhex2[4]	PIN_AB15	On-Board seven-segment display segment (HEX6[4])
Bhex2[5]	PIN_AA15	On-Board seven-segment display segment (HEX6[5])
Bhex2[6]	PIN_AC17	On-Board seven-segment display segment (HEX6[6])
Bhex3[0]	PIN_AD17	On-Board seven-segment display segment (HEX7[0])
Bhex3[1]	PIN_AE17	On-Board seven-segment display segment (HEX7[1])
Bhex3[2]	PIN_AG17	On-Board seven-segment display segment (HEX7[2])
Bhex3[3]	PIN_AH17	On-Board seven-segment display segment (HEX7[3])
Bhex3[4]	PIN_AF17	On-Board seven-segment display segment (HEX7[4])
Bhex3[5]	PIN_AG18	On-Board seven-segment display segment (HEX7[5])
Bhex3[6]	PIN_AA14	On-Board seven-segment display segment (HEX7[6])
CO	PIN_F17	On-Board LED (LEDG8)

V. POST-LAB

1) Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?

2) In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)

3) For the adders, refer to the **Design Resources and Statistics** in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every SystemVerilog circuit.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes sense? Are they complying with the theoretical design

expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder? Which design consumes more power than the other as you expected, why?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Summarize the high-level function performed by the serial logic processor and the three adders
2. Part 1 - Serial Logic Processor
    a. Include a block diagram can be adapted from lab manual or the top-level schematic generated by the RTL viewer. Please only include the top-level design if using the RTL viewer.
    b. Include a short description should include what was done with the provided code to extend it from 4 bits to 8 bits
    c. Include a simulation of the processor that has notes that give information such as what operation is being performed, where the result was stored, etc.
3. Part 2 - Adders
    a. Ripple Carry Adder
        i. Written description of the architecture of the adder
        ii. Block diagram.
    b. Carry Lookahead Adder
        i. Written description of the architecture of the adder
        ii. Describe how the P and G logic are used
        iii. Describe how you created the hierarchical 4x4 adder
        iv. Block diagram
        v. Block diagram inside a single CLA (4-bits)
        vi. Block diagram of how each CLA was chained together
    c. Carry Select Adder
        i. Written description of the architecture of the adder
        ii. Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later. Make sure you understand this!

   iii. Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic.
   d. Describe at a high level the area, complexity, and performance tradeoffs between the adders.
   e. Document the performance of each adder by creating a graph as specified in Prelab part C (page 4.6 in the manual).
4. Answers to the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.
5. Conclusion
   a. Describe any bugs and countermeasures taken during this lab.
   b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.
   c. Any additional summary you want to include

ECE 385

EXPERIMENT #5

An 8-Bit Multiplier in SystemVerilog

I. OBJECTIVE

In this experiment, you will design a multiplier in SystemVerilog for two 8-bit 2's compliment numbers and then run that multiplier on the DE2 FPGA board.

II. INTRODUCTION

You will use a simple add-shift algorithm to multiply two numbers. The algorithm is very similar to the pencil-and-paper method of multiplication except the final step for 2's Complement numbers depends on the sign bit. Consider the following example to calculate 8-bit 00000111 (7, Multiplicand) x 11000101 (-59, Multiplier)

```
  00000111          7 (multiplicand)
x 11000101       x (-)59 (multiplier)
 00000111        (-)413
+00000000x
+00000111xx
+00000000xxx
+00000000xxxx
+00000000xxxxx
+00000111xxxxxx
-00000111xxxxxxx  Subtract (or Add 2's comp of 00000111)
1111111001100011 (2's comp of result=0000000110011101=413)
```

Let us see how to perform multiplication using the add-shift method that you will use to multiply the contents of register B and switches S, leaving the result in registers AB:

Initial Values: X = 0, A = 00000000, B = 11000101 (achieved using ClearA_LoadB signal), S = 00000111, M is the least significant bit of the multiplier (Register B).

Function	X	A	*B*	M	Comments for the next step
Clear A, LoadB	0	0000 0000	*11000101*	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	0	0000 0111	*11000101*	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1 *1100010*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	11 *110001*	1	Add S to A since M = 1.
ADD	0	0000 1000	11 *110001*	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0100	011 *11000*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0010	0011 *1100*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	00011 *110*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0000	100011 *11*	1	Add S to A since M = 1
ADD	0	0000 0111	100011 *11*	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1100011 *1*	1	Subtract S from A since 8th bit M = 1.
SUB	1	1111 1100	1100011 *1*	1	Shift XAB after SUB complete
SHIFT	1	1111 1110	01100011	1	8th shift done. Stop. 16-bit Product in AB.

In the ADD state, the values of A and S are first sign-extended to 9 bits, and then summed together. The 9-bit results (not including the Cout) are then stored into XA. In the SHIFT state, the entire 17 bits of XAB is arithmetically right-shifted by one bit.

When M = 0, an ADD does not need to be performed. In that case, the ADD cycle can be omitted or a zero can be added to A. In addition, since we are using a 2's complement representation, we need to consider negative numbers. If A is negative, then XA will contain the correct partial sum and the sign will be preserved since the shift operation will perform an arithmetic shift on XAB. If B is negative (the most significant bit = 1), then M will be 1 after the seventh shift (see the example above). In that case a subtract operation is performed since the 8th bit of B has negative weight with 2's complement representation.

The 9-bit Adder/Subtractor should be designed using Full Adder primitives that you create. In other words, do not use the available SystemVerilog arithmetic operations "+" (add) and "-" (subtract) for this experiment. In future, you may use these operations in your designs.

You should design your control unit such that it executes one multiply operation when the Run press button is pressed. You can use symbolic states for the state machine in the controller for this experiment. You will need to have a Reset input that will reset the controller in the initial/start state. An **incomplete** block diagram of the circuit is shown in Figure 1:

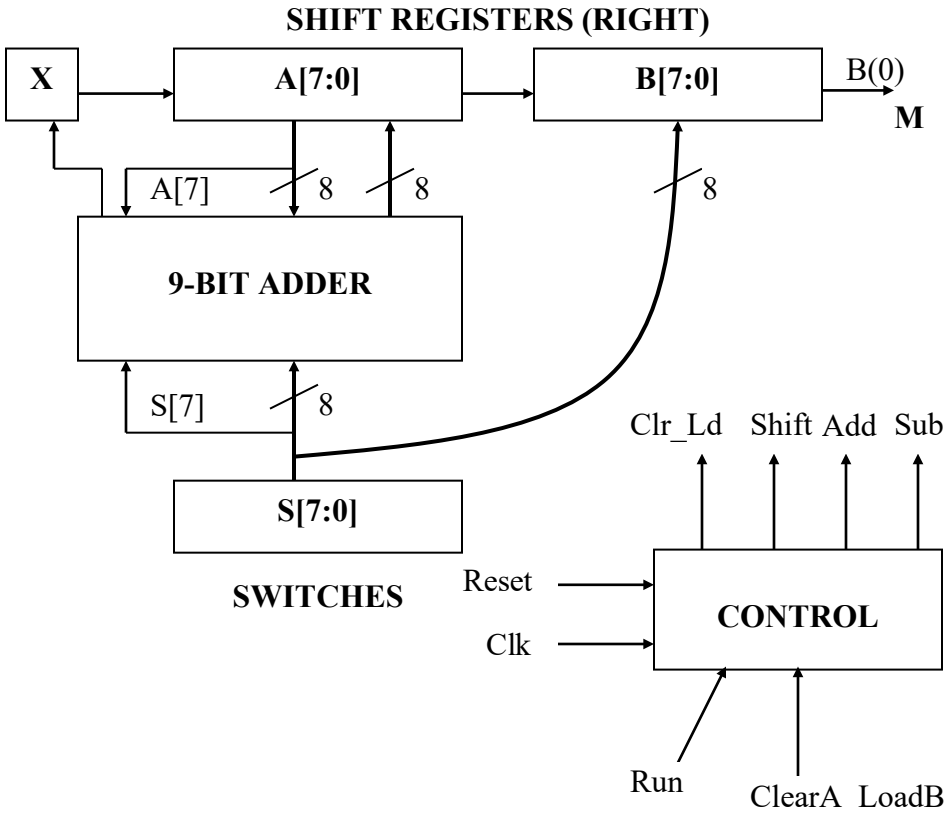


Figure 1: Incomplete Block Diagram

Your circuit should have the following inputs and outputs:

Inputs

S – logic [7:0]

Clk, Reset, Run, ClearA_LoadB – logic

Outputs

AhexU, AhexL, BhexU, BhexL – logic [6:0]

Aval, Bval – logic [7:0]

X –logic

To perform a multiplication, you will first load the multiplier to Register B by setting the switches (S) to represent the multiplier and pressing the ClearA_LoadB button. ClearA_LoadB button should also clears the X and A registers. Then you will set the switches (S) to represent

the multiplicand and press the Run button. ClearA_LoadB should be released before Run button is pushed. Once the Run signal triggers the multiplication, the circuit should complete the multiply operation regardless of the status of Run signal. The circuit should stop once the multiplication is done and the correct result should be displayed by outputting AB on the hex displays. Another multiply operation can be triggered by releasing the Run button and pressing it again.

Your circuit should support consecutive multiplications to receive full demo points. Note that neither ClearA_LoadB nor Reset buttons will be pressed between consecutive presses of the Run button, so your circuit needs to clear X and A before the next multiplication execution starts to get the correct results.

Demo Points Breakdown:

1.0 point: Functional simulation completed successfully

1.0 point: Correct operation of the *Clear_A_Load_B* function on the DE2 board

1.0 point: Correct operation of the *Multiplication* function on the DE2 board. (++, +-, -+, --)

1.0 point: Correct operation of *consecutive* multiplications on the DE2 board. (like -1 × -1 × …)

1.0 point: Execution cycle responds correctly (exactly one execution per press of the "Run" button)

III. PRE-LAB

A. Rework the 8-bit multiplication example presented in the table form at the beginning of this assignment. Use Multiplier B = 7, and Multiplicand S = -59. Note that this is different than the case when B = -59 and S = 7.

B. Design, document, and implement the 8-bit multiplier in SystemVerilog.

You will need to bring the following to the lab:

1. Your code for the 8-bit multiplier. You can bring the code to the lab using a USB storage device, FTP, or any other method.

2. Block diagram of your design, with components, ports, and interconnections labeled.

3. A simulation of your design showing at least one full multiplication. You should set the radix of the Switches, Aval, and Bval signals to signed decimal for readability. (Radix is set by right-clicking on a signal and selecting *Properties*.)

IV. LAB

Follow the Lab 5 demo information on the course website.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Run	PIN_R24	On-Board Push Button (KEY3)
ClearA_LoadB	PIN_N21	On-Board Push Button (KEY2)
Reset	PIN_M23	On-Board Push Button (KEY0)
S[0]	PIN_AB28	On-board slider switch (SW0)
S[1]	PIN_AC28	On-board slider switch (SW1)
S[2]	PIN_AC27	On-board slider switch (SW2)
S[3]	PIN_AD27	On-board slider switch (SW3)
S[4]	PIN_AB27	On-board slider switch (SW4)
S[5]	PIN_AC26	On-board slider switch (SW5)
S[6]	PIN_AD26	On-board slider switch (SW6)
S[7]	PIN_AB26	On-board slider switch (SW7)
AhexL[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
AhexL[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
AhexL[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
AhexL[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
AhexL[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
AhexL[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
AhexL[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
AhexU[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
AhexU[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
AhexU[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
AhexU[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
AhexU[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
AhexU[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
AhexU[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
BhexL[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
BhexL[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
BhexL[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
BhexL[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
BhexL[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
BhexL[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
BhexL[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
BhexU[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
BhexU[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
BhexU[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])
BhexU[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
BhexU[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
BhexU[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
BhexU[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])

X	PIN_F17	On-Board LED (LEDG8)

(Assignments for Aval and Bval are intentionally omitted. These outputs are included primarily for simulation, where reading the hex display outputs is not practical. Similarly, reading the outputs of your circuit in binary is not as practical as reading them in hex. You are free to reuse the assignments from Lab 4 for these signals, if you wish.)

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Come up with a few ideas on how you might optimize your design to decrease the total gate count and/or to increase maximum frequency by changing your code for the design.

2) Make sure your lab report answers at least the following questions:

- What is the purpose of the X register. When does the X register get set/cleared?
- What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?
- What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Summarize the basic functionality of the multiplier circuit
2. Pre-lab question
    a. Rework the multiplication example on page 5.2 of the lab manual, as in compute 00000111 * 11000101 in a table similar to the example
3. Written description and diagrams of multiplier circuit
    a. Summary of operation
        i. Explain in words how operands are loaded, how the multiplier computes its result, how the result is stored, etc.
    b. Top Level Block Diagram
        i. This can be generated from the RTL viewer. Please only include the top-level diagram and not the RTL view of every module.
    c. Written Description of .sv Modules
        i. List all modules used in a format shown in the appendix of this document
        ii. You may insert expanded RTL diagrams of each individual module here if it is legible
    d. State Diagram for Control Unit
        i. This can be done in a program like Visio, but if the Quartus state diagram generator is used, you must label the states and transitions. By default, the tool does not generate a very legible state diagram.
4. Annotated pre-lab simulation waveforms
    a. Must show 4 operations where operands have signs (+*+), (+*-), (-*+) and (-*-)
    b. Waveform must have notes that clearly show the operands as well as the result, etc.
5. Answers to post-lab questions
    a. Fill in the table shown in 5.6 with your design's statistics
    b. Answer all the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.
6. Conclusion
    a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it

b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

VII. APPENDIX

Module descriptions are an important part of the reports in ECE 385, and since this is the first significant FPGA lab, a brief example of how to write a module description is shown below.

Let's say you needed two 16-bit registers to store operands A and B in an adder that computes the sum of A and B. Here is example code of a 16-bit register with asynchronous reset and synchronous load that can be used for that purpose.

```
module reg16 (input [15:0] Din, input Clk, Load, Reset,
output logic [15:0] Dout);

always_ff @(posedge Clk or posedge Reset)
     begin
          if(Reset)
               Dout <= 16'h0000;
          else if(Load)
               Dout <= Din; //If load=1, perform parallel load
     on clock edge
     end
endmodule
```

And here is how a section of the report would describe it:

Module: `reg16.sv`

Inputs: `[15:0] Din, Clk, Load, Reset`

Outputs: `[15:0] Dout`

Description: This is a positive-edge triggered 16-bit register with asynchronous reset and synchronous load. When `Load` is high, data is loaded from `Din` into the register on the positive edge of `Clk`.

Purpose: This module is used to create the registers that store operands A and B in the adder circuit.

Simple modules can have a description and purpose that are just a sentence or two each, but more complicated modules require more detailed descriptions.

ECE 385

EXPERIMENT #6

Simple Computer SLC-3.2 in SystemVerilog

I. OBJECTIVE

In this experiment, you will design a simple microprocessor using SystemVerilog. It will be a subset of the LC-3 ISA, a 16-bit processor with 16-bit Program Counter (PC), 16-bit instructions, and 16-bit registers. *(For LC-3, see Patt and Patel (ECE 120 textbook).*

II. INTRODUCTION

There are three main components to the design of a processor. The central processing unit (CPU), the memory that stores instructions and data, and the input/output interface that communicates with external devices. You will be provided with the interface between the CPU and the memory (memory read and write functions). The computer will first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and then fetch again. See Figure 1.

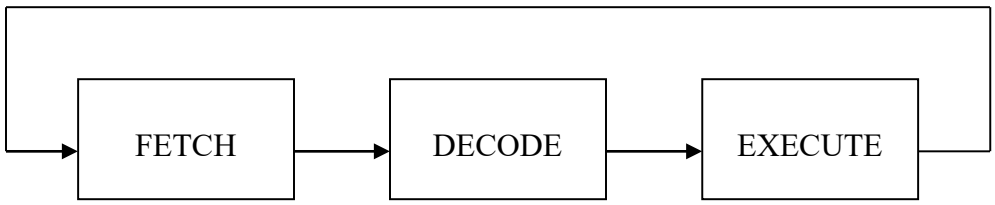


Figure 1

The CPU will contain a PC, a Instruction Register (IR), a Memory Address Register (MAR), a Memory Data Register (MDR), a Instruction Sequencer/Decoder, a status register (nzp), a 8x16 general-purpose register file, and an Arithmetic Logic Unit (ALU). All registers and instructions are 16-bits wide. The ALU will operate on 16-bit inputs. The Instruction Sequencer/Decoder will be responsible for providing proper control signals to the other components of the processor. It will contain a state machine that will provide the signals that will control the sequence of operations (fetch →decode → execute → fetch next) in the processor.

The simple computer will perform various operations based on the opcodes. An opcode specifies the operation to be performed. Specific opcodes and operations are shown in Table 1. The 4-bit opcode is specified by IR[15:12]; the remaining twelve bits contain data relevant that instruction.

In the table below, R(X) specifies a register in the register file, addressed by the three-bit address X. SEXT(X) indicates the 2's compliment sign extension of the operand X to 16 bits. nzp is the status register mentioned above. It is a three-bit value that states whether the resulting value loaded to the register file is negative, zero, or positive. This must be updated whenever an instruction performs a write to the register file (except JSR). For all instructions, PC ← PC + 1 is implicit, unless PC is stated to get some other value. In the table, right-hand-side "PC" indicates the value of the PC register after it was incremented immediately following fetch.

Instruction	Instruction(15 downto 0)							Operation
ADD	0001	DR		SR1	0	00	SR2	R(DR) ← R(SR1) + R(SR2)
ADDi	0001	DR		SR	1	imm5		R(DR) ← R(SR) + SEXT(imm5)
AND	0101	DR		SR1	0	00	SR2	R(DR) ← R(SR1) AND R(SR2)
ANDi	0101	DR		SR	1	imm5		R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001	DR		SR	111111			R(DR) ← NOT R(SR)
BR	0000	n	z	p	PCoffset9			if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100	000		BaseR	000000			PC ← R(BaseR)
JSR	0100	1	PCoffset11					R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110	DR		BaseR	offset6			R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111	SR		BaseR	offset6			M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12						LEDs ← ledVect12; Wait on Continue

Table 1: The SLC-3.2 ISA

The IR will provide the Instruction Sequencer/Decoder with the instruction to be executed. The IR will also provide the datapath with any other necessary data. As mentioned earlier, the Instruction Sequencer/Decoder will need to generate the control signals to execute the instructions in proper order. The Instruction Sequencer/Decoder will also specify the operation to

the ALU (e.g. add, etc.). Note that each operation will take multiple cycles and the Instruction Sequencer/Decoder will need to provide signals appropriately at each cycle.

On a reset, the Instruction Sequencer/Decoder should reset to the starting "halted" state, and wait for Run to go high. The PC should be reset to zero upon a reset, where it should proceed on incrementing itself when Run is pressed for fetching the instructions line by line. The first three lines of instructions will be used to load the PC with the value on the slider switches, which indicates the starting address of the instruction(s) of interest (in the form of test programs for the demo), and the program should begin executing instructions starting at the PC. Your computer must be able to return to the halted state any time a reset signal arrives.

Instruction Summary

ADD	Adds the contents of SR1 and SR2, and stores the result to DR. Sets the status register.
ADDi	Add Immediate. Adds the contents of SR to the sign-extended value imm5, and stores the result to DR. Sets the status register.
AND	ANDs the contents of SR1 with SR2, and stores the result to DR. Sets the status register.
ANDi	And Immediate. ANDs the contents of SR with the sign-extended value imm5, and stores the result to DR. Sets the status register.
NOT	Negates SR and stores the result to DR. Sets the status register.
BR	Branch. If any of the condition codes match the condition stored in the status register, takes the branch; otherwise, continues execution. (An unconditional jump can be specified by setting NZP to 111.) Branch location is determined by adding the sign-extended PCoffset9 to the PC.
JMP	Jump. Copies memory address from BaseR to PC.
JSR	Jump to Subroutine. Stores current PC to R(7), adds sign-extended PCoffset11 to PC.
LDR	Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT(offset6)). Sets the status register.
STR	Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
PAUSE	Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be "cleared" per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.

Here are the operations in more detail:

Fetch:

MAR ← PC; MAR = memory address to read the instruction from

MDR ← M(MAR); MDR = Instruction read from memory (note that M(MAR) specifies the data at address MAR in memory).

IR ← MDR; IR = Instruction to decode
PC ← (PC + 1)

Decode:
Instruction Sequencer/Decoder ← IR

Execute:
Perform the operation based on the signals from the Instruction Sequencer/Decoder and write the result to the destination register or memory.

Fetch, Load, and Store Operations:
For Fetch, Load (LDR), and Store (STR) operations you will need to set the memory signals (see Memory Interface below) appropriately for each state of the fetch/load/store sequence. Also, notice that the RAM we use does not have an R signal indicating that a read/write operation is ready. Instead, for any states reading from or writing to RAM, we stay at those states for several clock cycles to ensure that a memory read/write operation is complete.

FETCH:
state1: MAR ← PC
state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
state3: IR ← MDR;
PC ← PC+1; -- "+1" inserts an incrementer/counter instead of an adder.
Go to the next state.

LOAD:
state1: MAR ← (BaseR + SEXT(offset6)) from ALU
state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
state3: R(DR) ← MDR;

STORE:
state1: MAR ← (BaseR + SEXT(offset6)) from ALU; MDR ← R(SR)
state2: M(MAR) ← MDR; -- *assert Write Command on the RAM*

Memory Interface
The DE2 board is equipped with one 2 MB (1Mx16) SRAM. You will need to provide a memory address in MAR, data to be written in MDR (in the case of Store), and the Read and Write signals. The interface for these memory chips is as follows:

Data	Bidirectional 16-bit data bus.
ADDR	20-bit Address bus (in LC-3, the address space is only 16-bit wide, so the addresses are zero-extended to 20 bits)
CE	Chip Enable. When active, allows read/write operations. Active low.
UB	Upper Byte enable. Allows read/write operations on I/O<15:8>. Active low.
LB	Lower Byte enable. Allows read/write operations on I/O<7:0>. Active low.

OE Output Enable. When active, RAM chips will drive output on the selected address. Active low.
WE Write Enable. When active, orders writes to selected address. Active low. Has priority over OE.

Note that "Data" is declared as an **inout** port type. This means that it is a bidirectional data bus. In general, any port that attempts to both read and write to a bus needs to be declared as an inout type. A port can write to the bus through an output port. When not writing to the bus, you should assign the output a high-impedance value ("ZZZZZZZZZZZZZZZZ"). Otherwise, when reading from memory, both your circuit and the SRAM will try to drive the line at the same time, with unpredictable results (possibly including damage to one or both chips). This part has been provided to you in the given file named "tristate.sv".

I/O Specifications

I/O for this CPU is memory-mapped. I/O devices are connected to the memory signals, with a special buffer inserted on the memory data bus. When a memory access occurs at an I/O device address, the I/O device detects this, and sends a signal to the buffer to deactivate the memory, and instead use the I/O device's data for the response. You will be provided with a SystemVerilog entity that encapsulates this functionality into a single module for you to insert between your processor and the external memory (this is part of what the mem2io module does) (see figures 2 & 3)

Mem2IO

This manages all I/O with the DE2 physical I/O devices, namely, the switches and 7-segment displays. See Table 2. Note that the two devices share the same memory address. This is acceptable, because one of the devices (the switches) is purely input, while the other (the hex displays) is purely output.

Physical I/O Device	Type	Memory Address	"Memory Contents"
DE2 Board Hex Display	Output	0xFFFF	Hex Display Data
DE2 Board Switches	Input	0xFFFF	Switches(15:0)

Table 2: Physical I/O Device List

You will need to create a top-level port map file that includes your CPU, the Mem2IO entity, and four HexDrivers. The CPU is a high level entity that contains the majority of your modules, including the ISDU. The various memory control signal inputs and the memory address input should be connected to the corresponding outputs from the CPU (which are also output

from your top-level entity to the actual memory). The memory data inout port from your CPU should be connected to the Data_CPU inout port of the Mem2IO unit; the Data_Mem inout port of the Mem2IO unit should be connected to an inout port on your top-level entity, which should be assigned to the appropriate pin connected to physical memory. The four "HEX#" output signals should be connected to HexDriver inputs. See the partial block diagrams in figures 2 & 3.

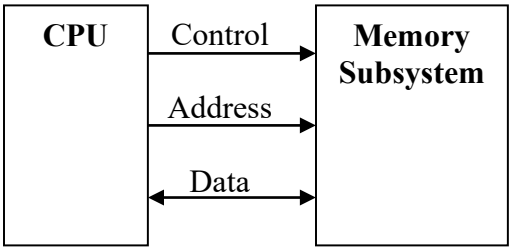


Figure 2: Conceptual Picture of Memory

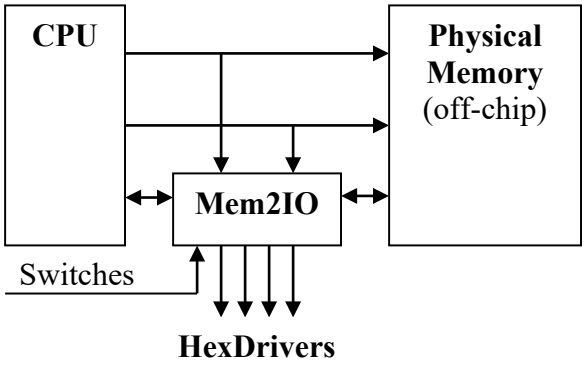


Figure 3: Memory with Mem2IO unit

Usage of the Pause Instruction

The Pause instruction is to be used in conjunction with I/O. Pausing allows the user time to set the switches before an input operation, and read the output after an output operation. The top two bits of the ledVect12 field of the instruction indicate the related I/O operation as indicated in Table 3. Note that these should be considered as masks, not as mutually exclusive values. For example, a ledVect12[11:10] value of "11" would indicate that both new data is being displayed on the hex displays and the program is asking for a new switch value. The remaining ledVect12 bits should be used to output a unique identifier to communicate the location in the program to keep track of the computer's progress. NOTE: The ledVect12 vector does not mean anything to the processor; their sole purpose is to be a visual cue that the user can define (when programming) so that he/she can tell where the program is during execution. The following table reflects the convention used in the test programs.

ledVect12(11:10) Mask	Meaning (cue for the user only)
"01"	Previous operation was a write to the hex display
"10"	Next operation is a read from switches

Table 3: ledVect12(11:10) Masks for Pause Instruction

Your top-level circuit should have **at least** the following inputs and outputs:

Inputs

S – logic [15:0]

Clk, Reset, Run, Continue –logic

Outputs

LED – logic [11:0]

HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 - logic [6:0]

CE, UB, LB, OE, WE –logic

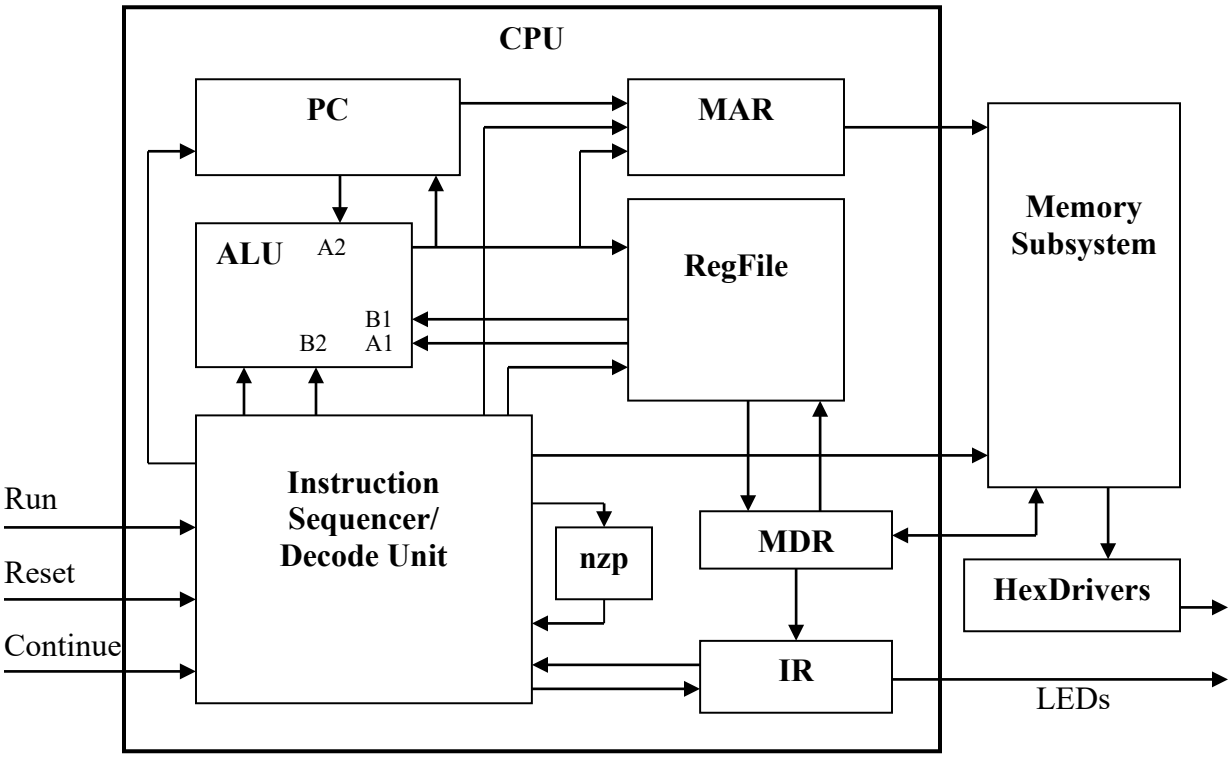
ADDR – logic [19:0]

Bidirectional ports (inout)

Data - logic [15:0]

(You may expand this list as needed for simulation and debugging.)

SIMPLIFIED SAMPLE CPU BLOCK DIAGRAM



Notes: Arrows are shown for connections between components. There may be multiple signals going from one block to the other even if there is only one connection shown between the blocks. One arrow does not mean one signal/bus in all cases. Signal multiplexing has been omitted (your diagram should show a mux in front of most registers). All registers will also have Clk as an input. Note that this block diagram does not show the mem2io module

that should serve as an interface between the CPU and the memory (it is implicit in the memory subsystem block, this is just a reminder that you need to include it).

LC3 STATE DIAGRAM FROM APPENDIX C OF PATT AND PATEL

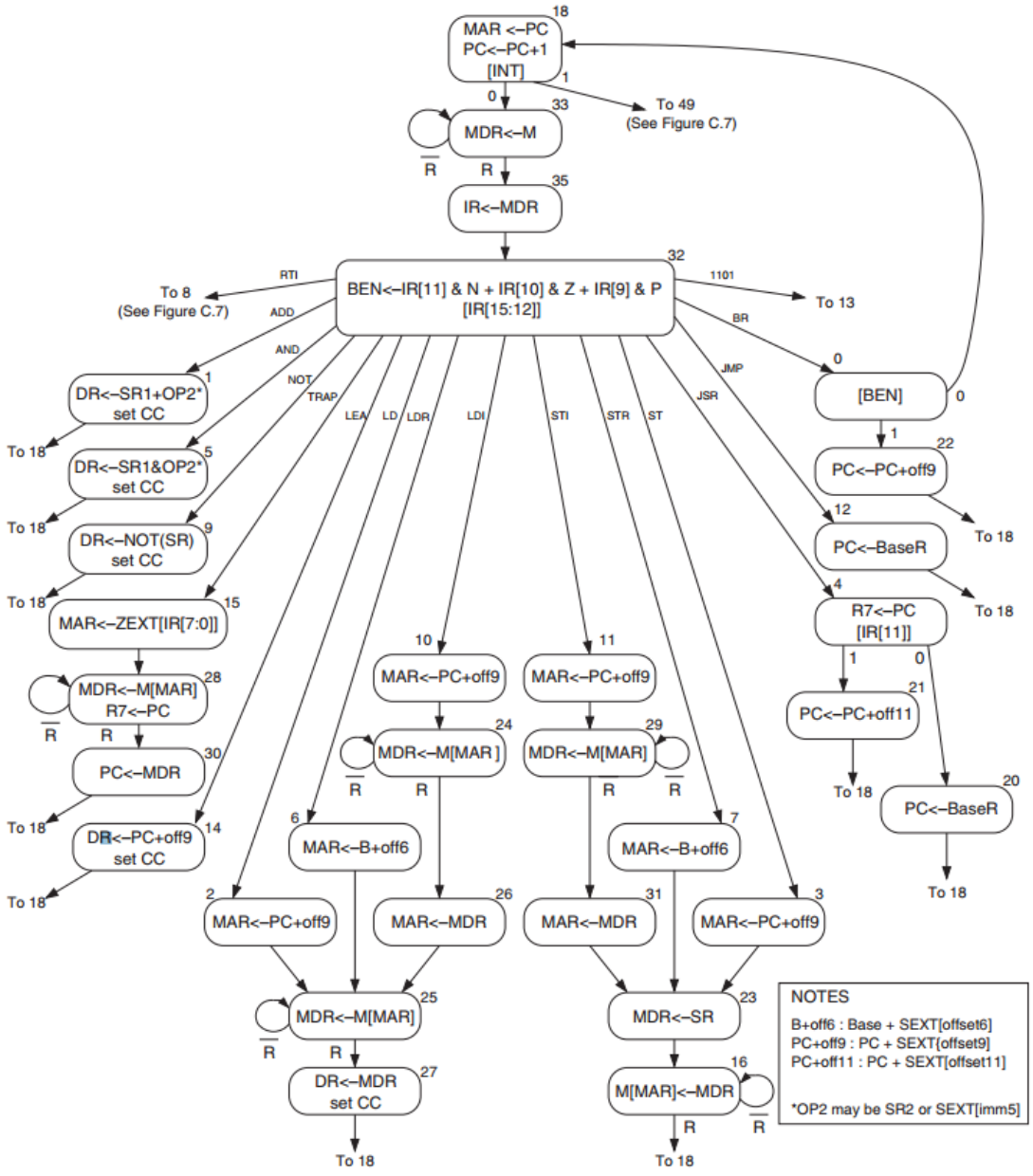


Figure C.2 A state machine for the LC-3

Notes: This is not the state diagram for the simplified LC3, it has some instructions that are not required to be implemented in this lab. Please refer to the instruction summary for details on which instructions are required to complete the lab.

III. PRE-LAB

A. **Week 1:**

Lab 6 is split up into two discrete tasks. In the first week, you will implement the FETCH phase. You will have to understand the structure of the memory system, and how the memory system interfaces with the CPU. You will also have to implement all the necessary CPU entities and ISDU controls to be able to successfully fetch the instructions line by line from the on-board memory to the CPU. Note that since you'll not be doing DECODE and EXECUTE during week 1, you don't have to pass the fetched instructions into the ISDU. But instead, you should display the content of the IR, which will be storing the fetched instructions at the end of the FETCH phase.

You are provided with the following entities on the website Mem2IO, Test_Memory, ISDU, and tristate. The use for Test_Memory is only to simulate your design, to replace the off chip memory.

For the purpose of Week 1 demo, you should connect the HEX displays directly to the IR rather than connecting them to the Mem2IO (in contrast, in week 2, you will specify a special address in memory that you write to in order to show data on the HEX displays). To display the content of the IR on the FPGA, there are extra pause states at the end of the FETCH phase to be able to hold and see the content of the IR. Pressing the 'continue' button, your ISDU should loop back to perform the FETCH phase all over again, instead of continuing onto the DECODE phase.

You should not need to modify the given ISDU for week 1, a higher level LC3 entity is given and you will have to implement all other component parts of the LC3 such as the program counter, register file, and data bus.

Week 1 Demo Point Breakdown:

1 point: Simulate the correct value of MAR = PC and PC = PC +1;
1 point: Simulate the correct value of MDR = M(MAR) and IR = MDR;
1 points: Correct operation on the FPGA: Displaying correct value of IR on HEX4-7 in state PauseIR1.

B. **Week 2:**

For the second week, you will need to implement the DECODE and the EXECUTE phase. You will first extend the provided skeleton ISDU to include all the necessary state transitions and the necessary inputs/outputs in each of the states.

You will need to learn and understand the specification of the LC3 and its state diagram to figure out how to assign the various control signals in each state to produce the desired operations. At this point, you will have to take out the pause states which you have inserted after the FETCH phase during week 1, for the ISDU to continue onto the DECODE and the EXECUTE phase instead.

Week 2 Demo Point Breakdown:
1 point: Basic I/O Test 1
1 point: Basic I/O Test 2
1 point: Self Modifying Code Test
1 point: XOR Test
1 point: Multiplication Test
1 point: Sort Test
1 point: Correct "Act Once" Behavior

IV. LAB

Follow the Lab 6 demo information on the course website. Follow the *Week 2 Test Programs Documentation* in the Lab 6 information page on the course website to demonstrate the 5 tests.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Run	PIN_R24	On-Board Push Button (KEY3)
Continue	PIN_N21	On-Board Push Button (KEY2)
Reset	PIN_M23	On-Board Push Button (KEY0)
S[0]	PIN_AB28	On-board slider switch (SW0)
S[1]	PIN_AC28	On-board slider switch (SW1)
S[2]	PIN_AC27	On-board slider switch (SW2)
S[3]	PIN_AD27	On-board slider switch (SW3)
S[4]	PIN_AB27	On-board slider switch (SW4)
S[5]	PIN_AC26	On-board slider switch (SW5)
S[6]	PIN_AD26	On-board slider switch (SW6)
S[7]	PIN_AB26	On-board slider switch (SW7)
S[8]	PIN_AC25	On-board slider switch (SW8)
S[9]	PIN_AB25	On-board slider switch (SW9)
S[10]	PIN_AC24	On-board slider switch (SW10)
S[11]	PIN_AB24	On-board slider switch (SW11)

S[12]	PIN_AB23	On-board slider switch (SW12)
S[13]	PIN_AA24	On-board slider switch (SW13)
S[14]	PIN_AA23	On-board slider switch (SW14)
S[15]	PIN_AA22	On-board slider switch (SW15)
LED[0]	PIN_G19	On-Board LED (LEDR0)
LED[1]	PIN_F19	On-Board LED (LEDR1)
LED[2]	PIN_E19	On-Board LED (LEDR2)
LED[3]	PIN_F21	On-Board LED (LEDR3)
LED[4]	PIN_F18	On-Board LED (LEDR4)
LED[5]	PIN_E18	On-Board LED (LEDR5)
LED[6]	PIN_J19	On-Board LED (LEDR6)
LED[7]	PIN_H19	On-Board LED (LEDR7)
LED[8]	PIN_J17	On-Board LED (LEDR8)
LED[9]	PIN_G17	On-Board LED (LEDR9)
LED[10]	PIN_J15	On-Board LED (LEDR10)
LED[11]	PIN_H16	On-Board LED (LEDR11)
HEX0[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
HEX0[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
HEX0[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
HEX0[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
HEX0[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
HEX0[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
HEX0[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
HEX1[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
HEX1[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
HEX1[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])
HEX1[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
HEX1[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
HEX1[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
HEX1[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])
HEX2[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
HEX2[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
HEX2[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
HEX2[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
HEX2[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
HEX2[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
HEX2[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
HEX3[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
HEX3[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
HEX3[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
HEX3[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
HEX3[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
HEX3[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
HEX3[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
HEX4[0]	PIN_AB19	On-Board seven-segment display segment (HEX4[0])
HEX4[1]	PIN_AA19	On-Board seven-segment display segment (HEX4[1])
HEX4[2]	PIN_AG21	On-Board seven-segment display segment (HEX4[2])
HEX4[3]	PIN_AH21	On-Board seven-segment display segment (HEX4[3])
HEX4[4]	PIN_AE19	On-Board seven-segment display segment (HEX4[4])
HEX4[5]	PIN_AF19	On-Board seven-segment display segment (HEX4[5])
HEX4[6]	PIN_AE18	On-Board seven-segment display segment (HEX4[6])
HEX5[0]	PIN_AD18	On-Board seven-segment display segment (HEX5[0])
HEX5[1]	PIN_AC18	On-Board seven-segment display segment (HEX5[1])

HEX5[2]	PIN_AB18	On-Board seven-segment display segment (HEX5[2])
HEX5[3]	PIN_AH19	On-Board seven-segment display segment (HEX5[3])
HEX5[4]	PIN_AG19	On-Board seven-segment display segment (HEX5[4])
HEX5[5]	PIN_AF18	On-Board seven-segment display segment (HEX5[5])
HEX5[6]	PIN_AH18	On-Board seven-segment display segment (HEX5[6])
HEX6[0]	PIN_AA17	On-Board seven-segment display segment (HEX6[0])
HEX6[1]	PIN_AB16	On-Board seven-segment display segment (HEX6[1])
HEX6[2]	PIN_AA16	On-Board seven-segment display segment (HEX6[2])
HEX6[3]	PIN_AB17	On-Board seven-segment display segment (HEX6[3])
HEX6[4]	PIN_AB15	On-Board seven-segment display segment (HEX6[4])
HEX6[5]	PIN_AA15	On-Board seven-segment display segment (HEX6[5])
HEX6[6]	PIN_AC17	On-Board seven-segment display segment (HEX6[6])
HEX7[0]	PIN_AD17	On-Board seven-segment display segment (HEX7[0])
HEX7[1]	PIN_AE17	On-Board seven-segment display segment (HEX7[1])
HEX7[2]	PIN_AG17	On-Board seven-segment display segment (HEX7[2])
HEX7[3]	PIN_AH17	On-Board seven-segment display segment (HEX7[3])
HEX7[4]	PIN_AF17	On-Board seven-segment display segment (HEX7[4])
HEX7[5]	PIN_AG18	On-Board seven-segment display segment (HEX7[5])
HEX7[6]	PIN_AA14	On-Board seven-segment display segment (HEX7[6])
CE	PIN_AF8	Chip Enable for SRAM – active low
UB	PIN_AC4	Upper Byte enable for SRAM– active low
LB	PIN_AD4	Lower Byte enable for SRAM – active low
OE	PIN_AD5	Output Enable for SRAM – active low
WE	PIN_AE8	Write Enable for SRAM – active low
Data[0]	PIN_AH3	Bidirectional Data Bus for SRAM - bit 0
Data[1]	PIN_AF4	Bidirectional Data Bus for SRAM - bit 1
Data[2]	PIN_AG4	Bidirectional Data Bus for SRAM - bit 2
Data[3]	PIN_AH4	Bidirectional Data Bus for SRAM - bit 3
Data[4]	PIN_AF6	Bidirectional Data Bus for SRAM - bit 4
Data[5]	PIN_AG6	Bidirectional Data Bus for SRAM - bit 5
Data[6]	PIN_AH6	Bidirectional Data Bus for SRAM - bit 6
Data[7]	PIN_AF7	Bidirectional Data Bus for SRAM - bit 7
Data[8]	PIN_AD1	Bidirectional Data Bus for SRAM - bit 8
Data[9]	PIN_AD2	Bidirectional Data Bus for SRAM - bit 9
Data[10]	PIN_AE2	Bidirectional Data Bus for SRAM - bit 10
Data[11]	PIN_AE1	Bidirectional Data Bus for SRAM - bit 11
Data[12]	PIN_AE3	Bidirectional Data Bus for SRAM - bit 12
Data[13]	PIN_AE4	Bidirectional Data Bus for SRAM - bit 13
Data[14]	PIN_AF3	Bidirectional Data Bus for SRAM - bit 14
Data[15]	PIN_AG3	Bidirectional Data Bus for SRAM - bit 15
ADDR[0]	PIN_AB7	SRAM address bit 0
ADDR[1]	PIN_AD7	SRAM address bit 1
ADDR[2]	PIN_AE7	SRAM address bit 2
ADDR[3]	PIN_AC7	SRAM address bit 3
ADDR[4]	PIN_AB6	SRAM address bit 4
ADDR[5]	PIN_AE6	SRAM address bit 5
ADDR[6]	PIN_AB5	SRAM address bit 6
ADDR[7]	PIN_AC5	SRAM address bit 7
ADDR[8]	PIN_AF5	SRAM address bit 8
ADDR[9]	PIN_T7	SRAM address bit 9
ADDR[10]	PIN_AF2	SRAM address bit 10
ADDR[11]	PIN_AD3	SRAM address bit 11
ADDR[12]	PIN_AB4	SRAM address bit 12

ADDR[13]	PIN_AC3	SRAM address bit 13
ADDR[14]	PIN_AA4	SRAM address bit 14
ADDR[15]	PIN_AB11	SRAM address bit 15
ADDR[16]	PIN_AC11	SRAM address bit 16
ADDR[17]	PIN_AB9	SRAM address bit 17
ADDR[18]	PIN_AB8	SRAM address bit 18
ADDR[19]	PIN_T8	SRAM address bit 19

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Document any problems you encountered and your solutions to them, and a short conclusion. Before you leave from your lab session submit your latest project code including the .sv files to your TA on his/her USB drive. TAs are under no obligation to accept late code, code that doesn't compile (unless you got 0 demo points) or code files that are intermixed with other project files.

2.) Answer at least the following questions in the lab report

- What is MEM2IO used for, i.e. what is its main function?
- What is the difference between BR and JMP instructions?
- What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of the signal in our design? What implications does this have for synchronization?

VI. REPORT

6.14

You do NOT need to write a report after week 1. Instead you will write one report for the entirety of Lab 6. Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Summarize the basic functionality of the SLC-3 processor
2. Written Description and Diagrams of SLC-3
    a. Summary of Operation
    b. Describe in words how the SLC-3 performs its functions. In particular, you should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.
    c. Block Diagram of slc3.sv
    d. This diagram should represent the placement of all your modules in the slc3.sv. Please only include the slc3.sv diagram and not the RTL view of every module (this can go into the individual module descriptions).
    e. Written Description of all .sv modules
        i. A guide on how to do this was shown in the Lab 5 report outline.
    f. Description of the operation of the ISDU (Instruction Sequence Decoder Unit)
        i. Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.
        ii. If you prefer to, you can lump this section into the module description section under ISDU.sv.
    g. State Diagram of ISDU
        i. This should represent all states present in the ISDU and their transitions. The diagram from Patt & Patel Appendix C can be used as a starting point but would need to be modified to be representative of the ECE385 implementation of the LC-3. You will lose points if you just copy the diagram.
3. Simulations of SLC-3 Instructions
    a. Simulate the completion of 3 instructions from the following groups: ADD/ADDi/AND/ANDi/NOT; BR/JMP/JSR; LDR/STR. For example, consecutively simulating ADD, BR and then LDR would be an acceptable simulation. You must annotate this diagram (for instance, label where instructions begin, where the answer is stored, etc.)

4. Post-Lab Questions
    a. Fill out the Design Resources and Statistics table from Post-Lab question one
    b. Answer all the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.
5. Conclusion
    a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it
    b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

ECE 385

EXPERIMENT #7

SOC with NIOS II in SystemVerilog

I. OBJECTIVE

In this experiment you will learn the basic capability of the NIOS II processor as the foundation of your System-On-Chip (SOC) projects. You will learn the fundamentals of memory-mapped I/Os and implement a simple SoC interfacing with peripherals such as the on-board switches and LEDs.

II. INTRODUCTION

The goal of this lab is creating a NIOS II based system on the Altera Cyclone IV device. The NIOS II is an IP based 32-bit CPU which can programmed using a high-level language (in this class, we'll be using C). A typical use case scenario is to have the NIOS II be the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic (designed using SystemVerilog) handles the high-performance operations.

The Introduction to NIOSII and Qsys will give you a walkthrough of the Platform Designer tool, which is used to instantiate IP blocks (including the NIOS II). We will set up a minimal NIOS II device with an SDRAM (Synchronous Dynamic RAM) controller and a PIO (Parallel I/O) block to blink some LEDs using a C program running on the NIOS II to confirm it is working. You will then be asked to write a program which reads 8-bit numbers from the switches on the DE2 board and sums into an accumulator, displaying the output using the green LEDs via the NIOS II. This will involve instantiating another PIO block to read data from the switches and modifying the C program to input data, add, and display the data.

Please read the **INTRODUCTION TO NIOS II AND QSYS** (INQ. 1-22).

Your top-level circuit should have **at least** the following inputs and outputs:
General Interface:

Inputs

```
KEY[0]      : logic      -- For Qsys-mapped hardware reset purposes
KEY[2]      : logic      -- For accumulator initialization ('Reset')
KEY[3]      : logic      -- For accumulator accumulation ('Accumulate')
CLOCK_50    : logic      -- 50 MHz clock input
```

```
        LEDG        : logic [7:0]          -- LED display of the accumulator
        SW          : logic [7:0]          -- Switches for the accumulation input
```

SDRAM Interface for Nios II Software:

Bidirectional ports (inout)

```
        DRAM_DQ     : logic [31:0]  -- SDRAM Data 32 Bits
```

Outputs

```
        DRAM_ADDR       : logic [12:0]  -- SDRAM Address 13 Bits
        DRAM_BA         : logic [1:0]   -- SDRAM Bank Address 2 Bits
        DRAM_DQM        : logic [3:0]   -- SDRAM Data Mast 4 Bits
        DRAM_RAS_N      : logic;        -- SDRAM Row Address Strobe
        DRAM_CAS_N      : logic;        -- SDRAM Column Address Strobe
        DRAM_CKE        : logic;        -- SDRAM Clock Enable
        DRAM_WE_N       : logic;        -- SDRAM Write Enable
        DRAM_CS_N       : logic;        -- SDRAM Chip Select
        DRAM_CLK        : logic;        -- SDRAM Clock
```

NOTE: For debugging, you may add LEDs, hex displays, switches, and/or buttons to the above lists.

III. PRE-LAB

A. Download the provided codes for Lab 7 on the ECE 385 course website. Follow the INQ tutorial to complete the NIOS II, memory, and the controller setup by performing the tasks as described in the tutorial. Your LEDG[0] should start blinking on your board as soon as the binary has been transmitted to the NIOS II CPU.

B. Modify the hardware and the software setup of the Lab 7 project to perform accumulation on the LED using the values from the switches as inputs. The green LEDs should always display the value of the accumulator in binary and the accumulator should be 0 on startup (all LEDs off). The accumulator should overflow at 255+1 to 0. (255 + 1 → 0, 255 + 2 → 1, etc.) Pressing 'Reset' (KEY[2]) at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off). Pressing 'Accumulate' (KEY[3]) loads the number represented by the switches into the CPU, adding it to the accumulator. The 8 right-most switches (SW [7:0]) are read as an 8-bit, unsigned, binary number with up being 1, down being 0. Push buttons should only react once to a single actuation.

C. Answer the *italicized* questions and fill in the table from the INQ tutorial. Be prepared to give answers to any of the questions from your TA when demoing. This is to ensure that you try to research what the settings do instead of simply trying to "make the picture look like your screen".

Hints: Unit test the input and output. The output should already work, but make sure you can turn on and off every segment. If you have problems, check the schematic for the DE2, and make sure you are toggling the correct pins.

For this, and the rest of the class, you may use the C standard libraries (stdlib.h) or the C++ equivalents, this can save you a lot of work when coding in C.

You will need to bring the following to the lab:

1. Your code for the Lab.
2. Block diagram of your design with components, ports, and interconnections labeled.

IV. LAB

Follow the Lab 7 demo information on the course website.

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Document any problems you encountered and your solutions to them, and a short conclusion.

VI. REPORT

7.4

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Summarize the basic functionality of the NIOS-II processor running on the Cyclone IV FPGA.
2. Written Description and Diagrams of NIOS-II System
        i. Describe in words the hardware component of the lab, in this lab, only the Platform Designer module is here.
3. Top Level Block Diagram
        i. For this lab, this may be trivial, since there is only the Platform Designer module.
4. Written Description of all .sv Modules
        i. For this lab, this may be trivial, since there is only the Platform Designer module.
        ii. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file lab7soc.v!
5. System Level Block Diagram (this is new for labs 7-9)
        i. The Platform Designer view of the SoC module should be found here, describe the functionality of each block (including those which are part of the SoC, such as the memories).
        ii. Note that this is not trivial, as there are many components within the Platform Designer view.
6. Describe in words the software component of the lab. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.
    a. Answers to all INQ Questions
        i. The Prelab question (part A) is one of these questions
    b. Post-lab Question
    c. Document the Design Resources and Statistics in table provided in the lab.
7. Conclusion
    a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it?
    b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?

ECE 385

EXPERIMENT #8

SOC with USB and VGA Interface in SystemVerilog

I. OBJECTIVE

In this experiment you will write a protocol to interface a keyboard and a monitor with the DE2 board using the on-board USB and VGA ports.

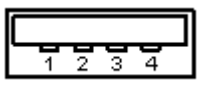
II. INTRODUCTION

You will connect the monitor to the VGA port and the keyboard to the USB port and depending on the key pressed on the keyboard, a small ball will move and bounce in either the X or Y direction on the monitor screen

How the USB Keyboard Works

The Universal Serial Bus (USB) standard defines the connection and communication protocols between computers and electronic devices. It also provides power supply to the connected devices. Due to the compatibility with a wide variety of devices, the USB standard has become prevalent since its introduction in the 1990s.

A USB cable has either a type A or a type B port on its ends. A USB port consists of four pins: VDD, D-, D+, and GND. Figure 1 shows the configuration. The VDD and GND pins are power lines, and D- and D+ are data lines. When data is being transmitted, D- and D+ take opposite voltage levels in a single time frame to represent one bit of data. On low and full speed devices, a differential '1' is transmitted by pulling D+ high and D- low, while a differential '0' is a D- pulled high a D+ pulled low.



Type A

Type B

Pin	Name
1	VDD (5V)
2	D-
3	D+
4	GND

Figure 1

The USB port on the DE2 board is equipped with the Cypress EZ-OTG (CY7C67200) USB Controller, which handles all the data transmission via the physical USB port and manages structured information to and from the DE2 board. CY7C67200 can act as either a Host Controller, which controls the connected devices, or a Device Controller, which makes the DE2 board itself a USB device. In this lab, we will be using CY7C67200 as a Host Controller.

A USB keyboard is a Human Interface Device (HID). HIDs usually do not require massive data transfers at any given moment, so a low speed transmission would suffice. (Other USB devices such as a camera or a mass storage device would often need to send large files, which would require bulk transfers, a topic not covered in this lab.) Unlike earlier standards such as PS/2, a USB keyboard does not send key press signals on its own. All USB devices send information only when requested by the host. In order to receive key press signals promptly, the host needs to constantly poll information from the keyboard. In this lab, after proper configuration, the CY7 will constantly send *interrupt requests* to the keyboard, and the keyboard will respond with key press information in *report descriptors*. A descriptor simply means a data structure in which the information is stored.

Table 1 shows the keyboard input report format (8 bytes). In this format, a maximum of 6 simultaneous key presses can be handled, but here we will assume only one key is pressed at a time, which means we only need to look at the first key code. Each key code is an 8-bit hex number. For example, the character A is represented by 0x04, B by 0x05, and so on. When the key is not pressed, or is released, the key code will be 0x00 (No Event).

Byte	Description
0	Modifiers Keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

Table 1

A more detailed explanation of how the keyboard works (and all the key code combinations) can be found in the INTRODUCTION TO USB AND EZ-OTG ON NIOS II (IUQ). The USB 2.0 Specification and the HID Device Class Definition (refer to the ECE 385 course website) are two documents that define all the behavior of a USB keyboard.

How the VGA Monitor works

For detailed explanation on how the VGA monitor works, please refer to the lectures and section 4.10 of the DE2-115 User Guide available on the ECE 385 website. Additional information can also be found by doing a web search.

Some sample codes are given on the ECE 385 website which generates the horizontal sync, vertical sync, horizontal pixel and vertical pixel location.

Instructions for the lab

The goal of this circuit is to make a small ball move on the VGA monitor screen. The ball can either move in the X (horizontal) direction or the Y (vertical) direction. (Remember that on the monitor, Y=0 is the top and Y=479 is the bottom!)

When the program starts, a stationary red ball should be displayed in the center of the screen. The ball should be waiting for a direction signal from the keyboard. As soon as a direction key (W-A-S-D) is pressed on the keyboard, the ball will start moving in the direction specified by the key.

W - Up
S - Down
A - Left
D - Right

When the ball reaches the edge of the screen, it should bounce back and start moving in the opposite direction.

The ball will keep moving and bouncing until another command is received from the keyboard. When a different direction key is pressed, the ball should start moving in the newly specified direction immediately, without returning to the center of the screen. NOTE: The ball should never move diagonally, and once set into motion by the initial key press, should never come to a stop.

Sample SystemVerilog code for the ball is given on ECE 385 web site. The sample code only implements the bouncing of the ball in the Y direction. You must add support for motion in the X direction and response to keyboard input.

Please do not take this lab lightly! Working with the VGA and Keyboard is a big time sink.

Summarizing, complete working code for a ball, moving and bouncing in the Y direction, can be found on the ECE 385 website. You have to add the following features:

- A keyboard entity that outputs the code of the last received key (Completed USB tutorial with additional logic)
- Motion and bouncing in the X and Y direction
- Immediately changing the ball's motion using the direction keys (W,A,S,D) (The ball should respond to the scan code)
- All of these functions should work in any sequence without having to reset the circuit

Your top-level circuit should have **at least** the following inputs and outputs:

General Interface:

Inputs

KEY[0]	: logic	-- Reset for any initialization purposes
CLOCK_50	: logic	-- 50 MHz clock input
HEX0, HEX1	: logic [6:0]	-- Makecode output in HEX

VGA Interface:

Outputs

VGA_R	: logic [7:0]	-- Red color output to the VGA
VGA_G	: logic [7:0]	-- Green color output to the VGA
VGA_B	: logic [7:0]	-- Blue color output to the VGA
VGA_CLK	: logic;	-- 25 MHz pixel clock for DAC chip
VGA_SYNC_N	: logic;	-- Sync signal for DAC chip
VGA_BLANK_N	: logic;	-- Blanking signal for DAC chip
VGA_VS	: logic;	-- Vertical Sync Signal to the VGA
VGA_HS	: logic;	-- Horizontal Sync Signal to the VGA

CY7C67200 Interface:

Inputs

OTG_INT	: logic	-- CY7C67200 Interrupt

Bidirectional ports (inout)

OTG_DATA	: logic [15:0]	-- CY7C67200 Data Bus 16 Bits

Outputs

OTG_ADDR	: logic [1:0]	-- CY7C67200 Address 2 Bits
OTG_CS_N	: logic	-- CY7C67200 Chip Select
OTG_RD_N	: logic	-- CY7C67200 Read
OTG_WR_N	: logic	-- CY7C67200 Write
OTG_RST_N	: logic	-- CY7C67200 Reset

SDRAM Interface for Nios II Software:

Bidirectional ports (inout)

sdram_wire_dq	: logic [31:0]	-- SDRAM Data 32 Bits

Outputs

sdram_wire_addr	: logic [12:0]	-- SDRAM Address 13 Bits
sdram_wire_ba	: logic [1:0]	-- SDRAM Bank Address 2 Bits
sdram_wire_dqm	: logic [3:0]	-- SDRAM Data Mast 4 Bits
sdram_wire_ras_n	: logic;	-- SDRAM Row Address Strobe
sdram_wire_cas_n	: logic;	-- SDRAM Column Address Strobe

```
sdram_wire_cke      : logic;        -- SDRAM Clock Enable
sdram_wire_we_n     : logic;        -- SDRAM Write Enable
sdram_wire_cs_n     : logic;        -- SDRAM Chip Select
sdram_clk           : logic;        -- SDRAM Clock
```

NOTE: For the partial credit, you may add LEDs, hex displays, switches, and/or buttons to the above lists.

III. PRE-LAB

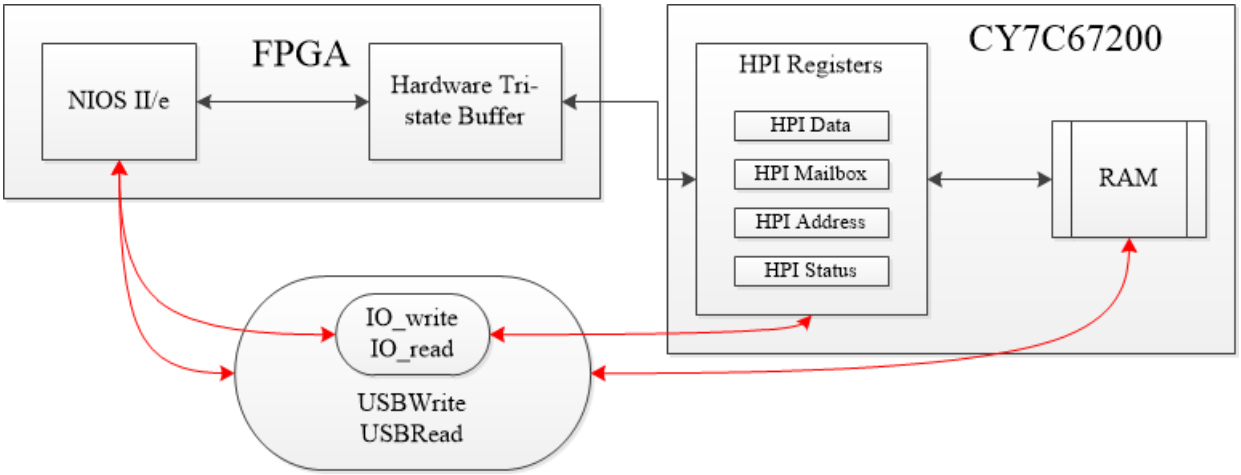
The bulk of this lab can be broken down into six distinct tasks.

1. Defining an appropriate Platform Designer setup for interfacing the NIOS II Processor with the CY7C67200 chip
2. Putting together the top level entity in system Verilog to include all the files given to you (including color_mapper, vga_controller and ball)
3. Writing a hardware I/O wrapper to tri-state the NIOS II driven data bus and to handle setting the CY7C67200 control pins
4. Filling in the two stub methods, IO_read and IO_write, with the appropriate code to correctly read and write a single register on the CYC67200 chip
5. Fill in the two stub methods, USBRead and USBWrite, with code that uses the register reads and writes in step 3 to properly read and write the CY7C67200's RAM
6. Edit ball.sv to satisfy the requirements described in the instructions section above

To elaborate on the difference between steps 4 and 5: in step 4, you are defining a way to write to one of the four registers in the register bank (HPI_ADDRESS, HPI_DATA, HPI_MAILBOX, or HPI_STATUS) on the CY7C67200 chip, while in step 5, you are defining the proper sequence of writes and reads to use those registers to access the CY7C67200's memory.

You will be using Platform Designer setup from your previous labs for **step 1**. Your task is to define PIO's (parallel input output connections) so that the NIOS II can interface with your System Verilog hardware I/O wrapper to control the CY7C67200 chip that handles USB input and output. Keep in mind that the NIOS II needs to be able to both read and write data (so your data PIO may be best setup as an in/out). You should need a PIO for each of the following at the least: address, data, read, write and chip select. You need to look at the datasheet to figure out what direction and width the PIO's need to be. (I would recommend using an **inout** instead of **bidir** for the data port)

For **step 2** you would need to download the files from the website and connect them together and to input/output pins as required. You should read the VGA Tutorial to understand how to connect the VGA controller. The incomplete Hardware Tri-state Buffer is provided in hpi_io_intf.sv.



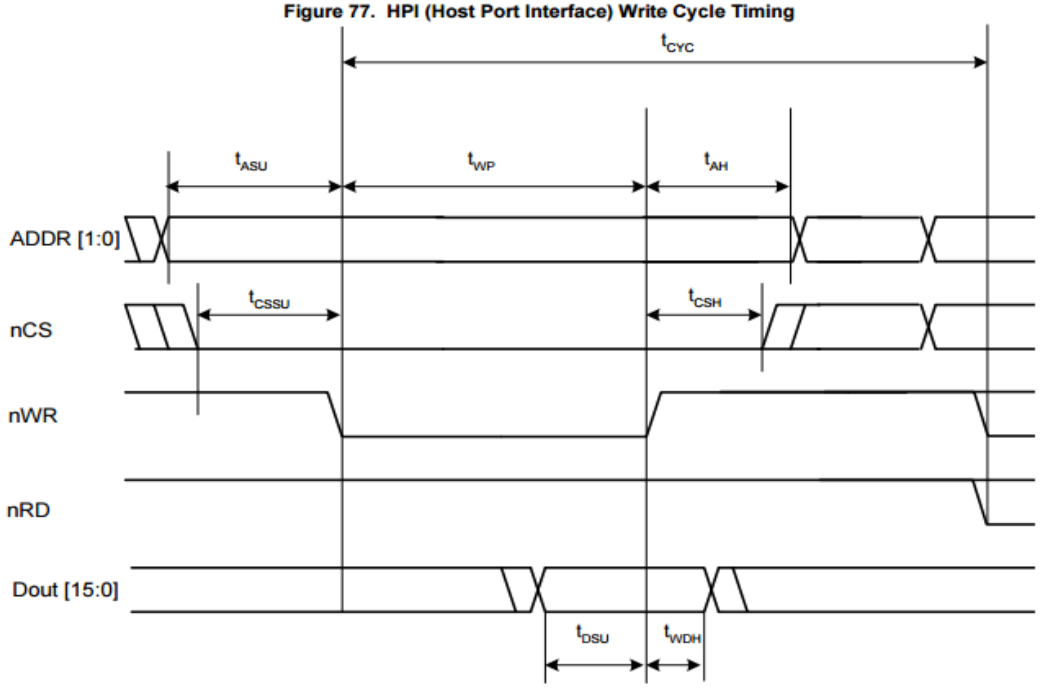
Step 3 consists of defining a hardware I/O wrapper in System Verilog that handles the connections between the PIO's coming from the NIOS II processor to the pins that control the CY7C67200 chip. Note that the data bus will need to be tri-stated, as it can be driven both by the NIOS and by the CY7C67200 chip.

Step 4 will mainly rely on you properly interpreting the provided timing diagrams for the HPI (Host Port Interface) read and write cycle timing. You won't necessarily need to concern yourself with issues of setup and hold times (as they are very small compared to the time a single cycle of our processor takes), but more the general sequence of setting changes (i.e. in what order do address, chip select, read, and write changes for a typical HPI read and write). There is an image of the write cycle timing diagrams below which has been taken from the data sheet for your reference, you can find the read cycle timing diagram in the CY7C67200 datasheet.

Step 5 is where you define how a single read and write of data from the CY7C67200 chip memory occurs. This relies on an understanding of what the HPI_ADDRESS and HPI_DATA registers on the CY7C67200 chip do.

For **step 6** you will need to add if/else conditions to take care of all the other cases, i.e. right and left edge conditions and key command conditions.

Figure 77. HPI (Host Port Interface) Write Cycle Timing



Parameter	Description	Min.	Typical	Max.	Unit
t_{ASU}	Address Setup	–1	–	–	ns
t_{AH}	Address Hold	–1	–	–	ns
t_{CSSU}	Chip Select Setup	–1	–	–	ns
t_{CSH}	Chip Select Hold	–1	–	–	ns
t_{DSU}	Data Setup	6	–	–	ns
t_{WDH}	Write Data Hold	2	–	–	ns
t_{WP}	Write Pulse Width	2	–	–	T[18]
t_{CYC}	Write Cycle Time	6	–	–	T[18]

Demo Point Breakdown:

This is the breakdown for partial credit, if you can get the keyboard and VGA controller working together perfectly with all the conditions met, you will get full demo points without having to demo the individual steps.

- Display the last received key (scan code) from the keyboard (1 point)
- Somehow show that the ball can move in X as well as Y direction without reprogramming the FPGA (perhaps by using switch inputs as stimulus) (1 point)
- Somehow show that your keyboard code can identify the 4 different directions and light a different LED for each direction (1 point)
- Get the ball to respond to the keyboard (1 point)
- Somehow show that the ball can bounce when moving in the X as well as Y direction without reprogramming the FPGA. Ball does not move diagonally (1 point)

IV. LAB

Follow the Lab 8 demo information on the course website.

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.29-31 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Document any problems you encountered and your solutions to them and write a short conclusion. Before you leave from your lab session submit your latest project code including both the .sv files and the software code to your TA on his/her USB drive. TAs are under no obligation to accept late code, code that doesn't compile (unless you got 0 demo points) or code files that are intermixed with other project files.

2.) Answer the following questions in the lab report

- What is the difference between VGA_CLK and Clk?
- In the file io_handler.h, why is it that the otg_hpi_data is defined as an integer pointer while the otg_hpi_r is defined as a char pointer?

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Briefly summarize the operation of the USB/VGA interface
2. Written Description of Lab 8 System
    a. Written Description of the entire Lab 8 system
    b. Describe in words how the NIOS interacts with both the CY7C67200 USB chip and the VGA components

   c. Written description of the CY7 to host protocol (HPI)
   d. Describe the purpose of the USBRead, USBWrite, IO_read and IO_write functions in the C code
3. Block diagram
   a. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.
4. Module descriptions
   a. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file for your Nios system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip). The Qsys view of the NIOS SoC is helpful here.
5. Answer to Post Lab Questions
6. Document the Design Resources and Statistics from the lab manual.
7. Conclusion
   a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it
   b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

ECE 385

EXPERIMENT #9

SOC with Advanced Encryption Standard in SystemVerilog

I. OBJECTIVE

In this experiment you will implement a 128-bit Advanced Encryption Standard (AES) using SystemVerilog as an Intellectual Property (IP) core. In the first part of this lab you will implement 128-bit AES encryption on the software IP core, in the second part of this lab you will implement 128-bit AES decryption in SystemVerilog, and will learn to design your own hardware IP core.

II. INTRODUCTION

An Intellectual Property (IP) core is a reusable hardware design unit that is the intellectual property of a party. It can come in many different forms, ranging from hardware description language (HDL) to transistor layouts. An IP core in the hardware world is analogous to a library in the software language. The IP cores are often designed to gear towards specific functionalities rather than complete hardware systems. An IP core is usually specified through its inputs and outputs, with emphasis on the various design specs and performance evaluations such as the ones we have introduced in Experiment 4.

Users of an IP core usually obtain their license from the owner of the IP core to incorporate the IP core into their own design for extended functionality. As a notable example, manufacturers of ARM-based processors obtain their license from ARM Holdings to design their own processors. The owner of the ARM IP core does not manufacture its own processors, but instead focuses on the development of the ARM architecture itself.

In this experiment, you will learn how to design a decryption IP core, and incorporate it with your software interface through the embedded NIOS II processor to form a complete system on chip (SoC). Through the software interface, your circuit will be able to decrypt the input data using the provided cipher key to reconstruct the original message.

Please read the INTRODUCTION TO ADVANCED ENCRYPTION STANDARD (IAES) and the INTRODUCTION TO THE AVALON-MM INTERFACE (IAMM).

III. PRE-LAB

A. **Week 1**:

Your primary task for the first week of this lab is to implement a 128-bit AES algorithm in C and then to run that algorithm on the NIOS II-e processor. The lookup tables which contains S-Box, Inverse S-Box, gf_mul (pre-calculated values for all possible GF(2^8) calculations), and Rcon, is be provided to you in aes.h, while the bulk of the algorithm will be implemented by you.

The sections for encryption include:

1. Your basic encryption input and control loop
2. A KeyExpansion function
3. An AddRoundKey function
4. A SubBytes function
5. A ShiftRows function
6. A MixColumns function

Psuedo-code is provided in the appendix for the progression through these various states, so ideally the process should be as simple as writing each component, and then writing a control loop that applies each step of the algorithm to the plaintext (called the state in the appendix) in the proper sequence.

NOTE: For debugging purposes, the course website contains a table of intermediate results. This file should allow you to figure out what sections of your AES code are having issues.

Once this AES algorithm is working (I would suggest having 2-3 different values for the plaintext and cipherkey that you can swap between to test it), your next task is to get the algorithm to work on the NIOS II-e. The main barrier here is getting the JTAG UART set up to handle input and output properly. You should already have this mostly implemented due to Lab 8, but if you haven't, the same set of steps applies.

For week 1, your code should have the abilities to receive plaintext and a cipher key from the Eclipse console. It should then use your C code to encrypt the message, with the results appearing in the console and the key on the hex displays. The course website has some example messages and keys (with results) if you want to use them.

To display your key on the hex displays, you will also need to complete the Avalon-MM Interface, described in IAMM, in week 1. You need to be able to send a 128 bit key to the hardware and display the first 2 and last 2 bytes on the hex display.

For week 1, you will be required to benchmark your C code on the NIOS II/e software core. It is sufficient to use the provided codes for benchmarking.

Week 1 Demo Point Breakdown:

1 point: Correct key expansion
1 point: Correct initial round key and looping rounds operations
1 point: Correct AES encryption test vectors
1 point: Benchmark of NIOS II/e based AES in kB/s
1 point: Correct Hardware Software communication to display the correct key on the hex displays

B. **Week 2:**

For the second week, you will need to design, document, and implement a 128-bit hardware decryption IP core in SystemVerilog that uses the AES algorithm. You will then use Platform Designer to link the AES IP core to your project. A state machine is **REQUIRED** to transition through the different AES modules, as well as to incorporate the capabilities to initialize, run, reset, and flag the completion of the AES decryption algorithm.

You are provided with avalon_aes_interface, AES, KeyExpansion, InvShiftRows, InvMixColumns and SubBytes. In the previous week you should have the completed the avalon_aes_interface to communicate between hardware and software and should also have a working Platform Designer environment.

To complete the decryption algorithm, you should need to modify AES.sv. You need to make a state machine to handle the looping of the AES algorithm. Since you are only allowed to include one instance of InvMixColumns you need to handle creating additional states in the above state machine or add another state machine. Using the

Hardware/Software interface you should be able to output the decrypted plaintext to the terminal. For simulating the AES decryption part, you may set AES.sv as the top-level entity temporarily, and write a testbench to simulate AES.sv directly.

For week 2 you will benchmark your HDL decryption core, separate from the encryption step. You can also use the provided codes for this step.

You will need to bring the following to the lab:

1. Your code for the Lab.
2. Block diagram of your design with components, ports, and interconnections labeled.
3. An annotated simulation of the AES decryption core showing the correct state transition of the AES state machine.

Week 2 Demo Point Breakdown:

1 point: Correctness of a single looping round operation. (on-board)
1 point: Correctness of stepping through the looping rounds using state machine. (on-board)
1 point: Transmitting correct result to NIOS II-e and displaying plaintext via terminal
1 point: Able to run consecutive decryption operations correctly without restarting the program.
1 point: Show benchmark results for HDL based AES decryption in kB/s

IV. LAB

Follow the Lab 9 demo information on the course website.

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.25-27 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	

Frequency	
Static Power	
Dynamic Power	
Total Power	

Document any problems you encountered and your solutions to them, and provide a short conclusion. Before you leave, submit your latest project code including both the .sv files and the software code to your TA on his/her USB drive. The TAs are under no obligation to accept late code or code files that are intermixed with other project files.

2.) Answer the following questions in the lab report

- Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show?
- If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Briefly summarize the operation of the AES encryptor/decryptor.
2. Written Description and Diagrams of the AES encryptor/decryptor
    a. Written description of the software encryptor
        i. Describe the role of the NIOS processor as well as the basic functionality of your C code
    b. Written description of the hardware decryptor
        i. Describe the basic steps of decryption and how this is controlled and computed in hardware
    c. Written description of the hardware/software interface (avalon_aes_interface.sv)
        i. Describe how the system sends data between NIOS and the hardware decryptor and how the register file is designed

   d. Block diagram(s)
      i. Include the RTL view of avalon_aes_interface.sv
      ii. Also include a top-level RTL diagram
   e. State Diagram of AES decryptor controller
      i. This is the state machine that was written in AES.sv.
      ii. You may abbreviate the 9 looping rounds in the state diagram like in figure 9 on page IAES.9 of the lab manual.
3. Module Descriptions
   a. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file for your Nios system! The Qsys view of the NIOS processor or lab9_top.sv is helpful here
4. Annotated Simulation of the AES decryptor
   a. Write a testbench for AES.sv, and set AES.sv as top level for simulation.
   b. In this simulation, you should display the input encrypted message, the input plaintext, the output decrypted message and the current state of the state machine.
   c. Notate various points of interest in the simulation (such as when the decryptor finishes decrypting, etc.).
5. Post-Lab Questions
   a. Fill out the design resources and statistics table
   b. Answers to the post-lab questions
6. Conclusion
   a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it
   b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

INTRODUCTION TO SYSTEMVERILOG AND TUTORIAL

ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to SystemVerilog

What is SystemVerilog:

- First adopted in 2005 (IEEE 1800-2005) [1] as an IEEE standard (current version IEEE 1800-2012) and an extension and successor of the standard Verilog (IEEE 1364-2005)
- Both a hardware description language and a hardware verification language (HDVL)
- Synthesizable (able to convert to actual hardware circuit) extensions from Accelera's Superlog
- Verification (not necessary synthesizable) extensions from Synopsys® OpenVera™
- Higher level of abstraction than Verilog and VHDL. Features inherited from Verilog, VHDL, C, C++

The hardware description language can be broken into two different categories: the weakly-typed Verilog/SystemVerilog [2] and the strongly-typed VHDL [3], whose differences and pros/cons will be discussed in the following paragraphs. The industry embraces Verilog/SystemVerilog these days due to their advantages described below, but this doesn't make VHDL obsolete. In fact, many intellectual property cores (source codes) are still written in VHDL, so a good hardware designer most likely needs to be familiar with both languages.

VHDL is a strongly-typed language. That is, its syntax and semantics explicitly represent all the operations and connections in the circuit, i.e., a designer can easily dictate how his code synthesizes, with all the data flow, type conversions, and circuit behavior written in an explicit, straightforward, and self-documenting way. All of the above features make VHDL closer to the actual circuit implementation and less error-prone. But as a result, its drawbacks are the lengthiness of the code and the difficulty of implementing very complicated circuits due to its verbosity. Lacking the higher level abstractions found in weakly-typed languages, the verification aspect of VHDL is also limited compared to Verilog/SystemVerilog.

Verilog and SystemVerilog on the other hand, are weakly-typed languages, that is, the coding reflects more of a functionality of the circuit instead of the actual bit-level wiring. For example, data type conversion in Verilog/SystemVerilog is done implicitly since all data types are inherently interchangeable through their natural bit-level representations. Contrary to the bit-level operation for the VHDL signals, Verilog/SystemVerilog has the capability to manipulate vectors like in a software language. It also supports complicated verifications such as assertions and the built-in random stimulus generator. Since much of the design intentions are implicitly interpreted by the compiler, the coding of Verilog/SystemVerilog is much more concise, but this also adds a degree of uncertainty since the designer may not always predict how his code synthesizes precisely.

Example: VHDL vs. Verilog

```
// Tri-state buffer in VHDL, strongly-typed
library IEEE;                        // Needs to invoke libraries
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tristate is                      // Interface declaration
port ( inarr : in std_logic_vector (15 downto 0);
      enable: in std_logic;
      outarr: out std_logic_vector (15 downto 0));
end entity;

architecture Behavioral of tristate is    // Behavioral architecture declaration
signal out_signal : std_logic_vector (15 downto 0); // Ports and signals are
separated
begin
       process (enable)              // Behavioral description process
       begin
         if (enable = '0') then          // Long behavioral statement
          out_signal <= "ZZZZZZZZZZZZZZZZ";
         else
          out_signal <= inarr;
         end if;
       end process;
outarr <= out_signal;
end Behavioral;

// Tri-state buffer in Verilog, weakly typed
module tristate (inarr, enable, outarr);  // Combined Interface and behavioral
       input [15:0] inarr;                                    // declaration
       input enable;
       output [15:0] outarr;

       reg [15:0] outarr;

       assign outarr = (enable) ? inarr : 16'bz; // C-style conditional statement
endmodule
```

SystemVerilog specifically, is a superset and a newer standard of Verilog intended to both more efficiently model and verify complex designs comparing to Verilog. To achieve this, it incorporates some of the strongly-typed features from VHDL, such as data type casting, as well as higher abstractions from software languages, such as classes and structs. SystemVerilog also improves upon the various constructs and syntax to make the coding more concise. We will focus on SystemVerilog and its syntax, but we will compare its features against VHDL and Verilog so you can better understand the differences. Below is an example showing how SystemVerilog greatly reduces the port and data type declarations comparing to Verilog.

Example: Verilog vs. SystemVerilog

```
// Verilog module construct
module Verilog (clk, enable, data,
out);
      // Input port declaration
      input        clk, enable;
      input  [2:0]  data;
      // Output port declaration
      output       out;
      // Optional data type
      //         declaration
      wire     clk, reset, enable;
      reg          out;

      always @ (posedge clk)
      begin
        if (enable)
         out <= /*do something*/;
      end
endmodule
```

```
// SystemVerilog module construct
module SysVerilog (input clk,
enable, input [2:0] data, output
logic out);
      // Data type always default
      to type logic

      always_ff @ (posedge clk)
      begin
        if (enable)
         out <= /*do something*/;
      end
endmodule
```

It is important to keep in mind that not all the constructs in VHDL, Verilog and SystemVerilog are synthesizable. This is because the verification aspect of these languages borrows ideologies from software languages to different degrees. Many constructs (especially in SystemVerilog) are intended for the convenience of simulations and are therefore non-synthesizable. It is not easy to keep a complete list of synthesizable/non-synthesizable constructs, especially when the list is not yet standardized for SystemVerilog [4].

One general way to quickly guess if a specific construct is synthesizable or not is by looking at its relationship with its actual hardware circuit: if a construct is easily correlating to some kind of circuits, then it is most likely a synthesizable construct; if a construct is styled like a software language and it's hard to imagine it to any hardware component at all, then it is most likely a non-synthesizable construct. However, since each compiler vendor defines its own subset of synthesizable constructs, the best way to check if your code is synthesizable is to see if the compiler generates any error during the compiling process. We will use the keyword "non-synthesizable" in this tutorial for any non-synthesizable constructs.

Uses both 2-state and 4-state Data Types:

2-state Data Types (0,1)	
bit	User defined vector size. Default unsigned
byte	8-bit signed integer
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer

4-state Data Types (0, 1, x, z)	
logic (**reg**)	User defined vector size. Default unsigned
integer	32-bit signed integer
time	64-bit signed integer

The above synthesizable data types can be enumerated using **enum**, where its values are defined by names. Besides the synthesizable data types, SystemVerilog also supports a few other non-synthesizable data types for verification purposes: **time** (64-bit unsigned integer as simulation time units), **shortreal** (same as float in C), **real** (same as double in C) and **string** (same as string in C).

Note: The **logic** data type is equivalent to the **reg** data type. It is introduced in SystemVerilog intended to replace the commonly used **reg** data type in Verilog to prevent misconception, where the latter is often mistaken to be an actual hardware register (it is in fact just a signal!).

Hierarchy:

- **Module and Procedural Blocks**

  A **module** is the primary design unit in SystemVerilog. Multiple modules may be combined to form a larger design. A module-procedural block pair provides design description.

  module declaration provides the external interface to the design entity. It contains pin-out description, interface description, input-output port definitions etc.

Example:

```
// Logic is contained within modules
module Example (input      clk, enable,  // Single-bit inputs, default logic type
                input  [2:0]  data,       // Three-bit input, default logic type
                output logic  out);      // Single-bit output, default logic type

       initial            // "initial" procedure block initializes the variables
       begin             // Used for testbench initialization. Not synthesizable.
              out = 0;
       end

       always_ff @ (posedge clk) // "always" procedure block triggered by the
       begin                                        // rising edge of clk
              if (enable)
                out <= /*do something with the input*/;
       end
endmodule
```

Procedure blocks describe the behavior of an entity or its structure (gates, wires, etc.) using SystemVerilog constructs. There are three types of procedure blocks in SystemVerilog, and they can be used in a nested fashion.

- **initial**: an initial block is used to initialize testbench assignments and is carried out only once at the beginning of the execution at time zero. It executes all the statements within the "**begin**" and the "**end**" statement without waiting. Non-synthesizable.

- **always**: an always block namely execute repeatedly throughout the execution period. When a triggering event occurs, all the statements within "**begin**" and "**end**" are executed once, and then waits for the next triggering event to occur. This waiting and executing of the "**always**" block is repeated throughout the circuit operation. To avoid unintended latches, SystemVerilog has three types of the "always" procedures.
  - **always_comb**: forcing combinational logic behavior; inferred sensitivity list (sensitivity list does not need to be explicitly written).
  - **always_latch**: forcing latched logic behavior; inferred sensitivity list.
  - **always_ff**: forcing synthesizable sequential logic behavior; sensitivity list needs to be explicitly written.

- **fork-join**: a fork-join block parallelizes the statements contained within themselves. It spawns multiple sequential procedures in parallel in addition to the original. It is used in testbench to divert simultaneous tasks (e.g. one parallel procedure executes the design and the other monitors any abnormalities such as an infinite loop). Non-synthesizable.
  - **fork-join**: waits for all parallel processes to complete before continuing the sequential process.
  - **fork-join_any**: waits for the first of all the parallel processes to complete before continuing the sequential process.
  - **fork-join_none**: does not wait for any of the parallel processes to complete before continuing the sequential process.

Example:

```
// Logic is contained within modules
module Example (input      a, b, z, enable, clk, reset,
                output logic z);

/*=========================================================*/
       // "always_comb" procedure

       always_comb
       begin: myComb    // "Named block" for readability. The naming does
              z = a + b;            // nothing, so its presence is not necessary.
       end: myComb     // End the named block here

/*=========================================================*/
       // "always_latch" procedure

       always_latch
       begin
              if (enable) begin
```

```
            z <= a + b;
        end
    end

/*=========================================================*/
    // "always_ff" procedure triggered by either the rising edge of the reset
    or the rising edge of clock

    always_ff @ (posedge clk or posedge reset) // ordering within the parenthesis
    begin                                      //does not matter
        if (reset) begin
            z <= 0;
        end else begin
            z <= a + b;
        end
    end
endmodule
```

Procedural blocks execute concurrently with respect to each other. Only the statements within a sequential block (**begin**-**end** block) execute sequentially in the order of their appearance. If there is more than one statement in the module, you will have to either place them in separate procedural blocks so they execute in parallel or in the same procedural block with a sequential **begin**-**end** block. This is the fundamental difference between a hardware language like SystemVerilog and a software language.

SystemVerilog is **case sensitive**, that is, lower case letters are unique from upper case letters. All SystemVerilog keywords are lower case.

- **Ports and Interfaces**

  Ports are the interconnections between a module and the outside world. There are three types of ports in Verilog and SystemVerilog:
  - **input**: Has a data type of Nets or Registers. Default to be "**wire**" in Verilog, and "**logic**" in SystemVerilog.
  - **output**: Has a data type of Nets or Registers. Default to be "**reg**" in Verilog, and "**logic**" in SystemVerilog. (Note: *The simulator, ModelSim, sometimes interprets untyped* ***output*** *as wire, so it is recommended to explicitly declare the type.*)
  - **inout**: Has a data type of Nets. Default to be "**wire**" in Verilog, and "**logic**" in SystemVerilog.

  Notice the differences in the port data type declarations between Verilog and SystemVerilog. While the port data types are distinguished in Verilog, all the ports in SystemVerilog share the same **logic** type. This reduces the coding complexity as shown in the first example of this tutorial.

  Simple combinational logic can be assigned outside of the procedural blocks using the "**assign**" statement. The statement is carried out continuously without the need of a sensitivity list. In this case, the output of some simple logic can be connected directly to

the output port of type **logic**. The below two examples are two different ways of implementing an identical AND gate:

Example:

```
// "assign" statement
module AND_Gate (input      a, b,
                 output logic z);

       assign z = a & b;

endmodule
```

```
// Prodecure block
module AND_Gate (input      a, b,
                 output logic z);

       always_comb
       begin
              z = a & b;
       end

endmodule
```

SystemVerilog has the capability to bundle the module ports together to cut down repetitions in the module interconnections. This is especially effective in the case where a master module sends many controlling signals to many slave modules. This is achieved by using the **interface** block to group the relevant ports, while using the **modport** statement to define the directional views of the ports.

Example:

```
// Interface declaration
interface Bus #(SIZE=2); // Interface declaration with variable SIZE
       logic [SIZE-1:0] m2s;    // Master to slave connection
       logic [SIZE-1:0] s2m;    // Slave to master connection

       modport master (input s2m, output m2s);  // Master view of the interfaces
       modport slave (input m2s, output s2m);   // Slave view of the interfaces

endinterface

// Master module
module Master (Bus.master interface1);

       // Do something using "interface1.m2s" and "interface1.s2m"

endmodule

// Slave module
module Slave (Bus.slave interface2);

       // Do something using "interface2.m2s" and "interface2.s2m"

endmodule

// Top-level module
module Example ();

       Bus   myInterface;
```

```
    Master myMaster (myInterface);
    Slave  mySlave (myInterface);

endmodule
```

Data Types:

Verilog uses three primary data types: "Nets", "Registers" and "Constants", while SystemVerilog unifies "Nets" and "Registers" into simple **logic**. We will only introduce the most commonly used ones here.

- Nets: Represents a physical wire in the circuit for connections between components. They do not hold any value themselves.
  - **wire**: Simple interconnecting wire between the design components.
    Syntax: **wire**; **wire** [<*MSB*>:<*LSB*>]
    - Example: **wire** my1BitWire; **wire** [3:0] my4BitWire;
  - **supply0**, **supply1**: Wires that are tied to logic '0' (ground) or '1' (power).
    - Example: **supply0** myGround; **supply1** myPower;

- Registers: Variables used to store the value assigned to them until another assignment changes their value.
  - **logic**: unsigned variable. Syntax: **logic**; **logic** [<*MSB*>:<*LSB*>]
    - Example: **logic** my1BitVariable; **logic** [3:0] my4BitVariable;
  - **integer**: signed variable (32 bits)
    - Example: **integer** myInteger;
  - **real**: double-precision floating point variable. Non-synthesizable.
    - Example: **real** myReal;

- Constants:
  - Integer Constants: Syntax: <*sign*><*size*>'<*radix*><*value*>
    - Sign: "–" for signed negative, else unsigned or signed positive.
    - Size: Size of the number in binary bits. Default to 32 bits if unspecified.
    - Radix: *b* (binary), *o* (octal), *d* (decimal), *h* (hexadecimal)
    - Value: Value of the number in the form indicated by the radix.

    Note: If <*size*> is smaller than <*value*>, then the excess bits in <*value*> are truncated. If <*size*> is larger than <*value*>, then leftmost bits are filled based on the value of the MSB in <*value*>. Specifically, a MSB of '0' and '1' is filled with '0', 'Z' is filled with 'Z', and 'X' is filled with 'X'.

- Examples:

Integer	Stored as	Comments
1	00000000000000000000000000000001	Default to be 32 bits
-8'hFA	00000101	2's complement of 'FA
5'b11010	11010	5 bits of binary
'hF	00000000000000000000000000001111	Default to be 32 bits
6'hF	001111	Extension to 6 bits
6'h8F	001111	Truncation to 6 bits
7'bZ	ZZZZZZZ	7 bits of high impedance

- Real Numbers (Non-synthesizable.): <*value*>.<*value*> or <*mantissa*>E<*exponent* >
  - Examples: 1.5; 3.6E3;

Data Structures, Classes and Packages:

Designed like a software language, SystemVerilog supports simple data structures **struct** and classes **class** to package variables and functions for verification purposes. "**package**" can also hold variables and functions, but is more equivalent to "namespace" in C language for sharing common code across modules. This greatly reduces the amount of code required to model complex circuits comparing to Verilog. While structures are synthesizable, classes and packages are non-synthesizable.

To initialize the entire structure, use the syntax: <*struct_name*> = *'{*<*member1_value*>, <*member2_value*>, *...}*, where the "'" operator in front of the bracket represents data casting (from array to struct). It is used to distinguish data casting from concatenation.

To assign a specific member, use: <*struct_name*>.<*member_name*> = <*member_value*>

Example:

```
// Anonymous structure
struct {
      logic        mybit,       // Single-bit logic
      int         myint,       // 32-bit integer
      logic  [2:0]  myarray,     // Three-bit array
} mystruct_instance;

// Typed structure (defined somewhere)
typedef struct {
      logic        mybit,       // Single-bit logic
      int         myint,       // 32-bit integer
      logic  [2:0]  myarray,     // Three-bit array
} mystruct_type

mystruct_type mystruct_instance;    // Struct instance declared elsewhere

// Class (mainly used in verification testbench)
```

```
class myclass_name;
	// Class properties
	logic        mybit,       // Single-bit logic
	int         myint,       // 32-bit integer
	logic  [2:0]  myarray,     // Three-bit array

	// Class method 1 - function that does something and returns something
	function int myfunc_get
		return myint;
	endfunction

	// Class method 2 - task that does something without returning anything
	task mytask_set (int i)
		myint = i;
	endtask

endclass

/*=========================================================*/
// File mypackage.sv
package mypackage_name;
	// Class properties
	logic        mybit;       // Single-bit logic
	int         myint;       // 32-bit integer
	logic  [2:0]  myarray;     // Three-bit array

	// Class method 1 - function that does something and returns something
	function int myfunc_get();
		return myint;
	endfunction

	// Class method 2 - task that does something without returning anything
	task mytask_set (int i);
		myint = i;
	endtask

endpackage
// End file mypackage.sv

// File mymodule.sv
`include "mypackage.sv"
import mypackage_name::*;    // import everything from the package

// Top-level module
module use_package ();

  /*access package members using mypackage_name::<member_name> */
  int   newint = 10;
  mypackage_name::mytask_set(newint);       // access member function
  logic [2:0] newarr = mypackage_name::array;  // access member variable

endmodule
// End file mymodule.sv
/*=========================================================*/
```

Data Assignment:

Two types of data assignments are available depending on whether the user wants the statements to be executed in parallel or sequentially. The two different assignments will often produce very different results, so it is important to be careful when using the two different assignments.

- Blocking Assignment: Sequential assignments made with "=" symbol. It blocks the execution of next statement until the current statement is executed, which means the assignment results are updated immediately. Analogous to variable assignment in VHDL.
- Non-blocking Assignment: Parallel assignments made with "<=" symbol. The next statement is not blocked due to the execution of the current statement, which means the assignment results are updated only until the end of the current cycle. Analogous to ":=" in VHDL.

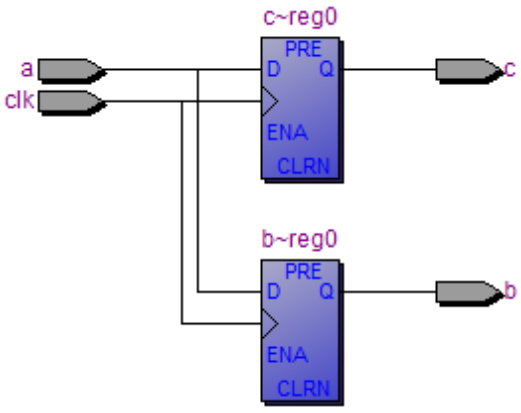
Note: Non-blocking assignment of multiple statements in a single procedural block is equivalent to single assignments in multiple procedural blocks.

A general guideline is to use non-blocking assignments for sequential procedure blocks, and blocking assignments for combinational procedure blocks.

Example: Blocking Assignment

```
module blocking (input      clk, a,
                 output logic b, c);

       always_ff @ (posedge clk)
       begin
// This is evaluated first serially
              b = a;
// Wait for the above evaluation to
// complete before its own evaluation,
// which makes the final evaluation to
//  be c=b=a.
              c = b;
       end
endmodule
```

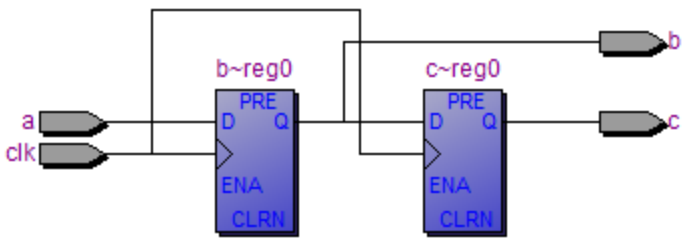
RTL Synthesis:



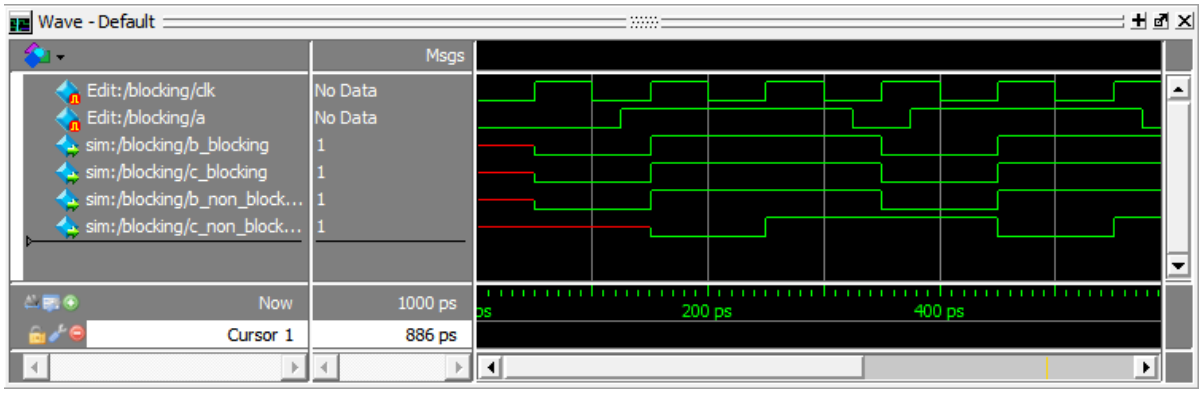
Example: Non-blocking Assignment

```
module nonblocking (input     clk, a,
                    output logic b, c);

       always_ff @ (posedge clk)
       begin
// These two statements are evaluated
//   in parallel disregarding the
//   dependence upon one another,
//   which means b is assigned to a
//  independently from c assigned to
b
              b <= a;
              c <= b;
       end
endmodule
```

RTL Synthesis:



Simulation:



- Notice that *b_blocking* and *c_blocking* holds identical value from a blocking simulation, while *c_non_blocking* lags a clock cycle from *b_non_blocking* from a non-blocking simulation.

Data Casting:

SystemVerilog can use the "**'**" operator for conversion between data types.

Example: **int**'(x) (casting x to type **int**); **signed**'(x) (casting x to **signed**); 5'(x) (casting x to change to 5 bits in size)

Operators:

Operators can be used in expressions involving signals, variables, or constant object types. Here are some of the useful operators:

- Arithmetic: +, -, *, /, % (modulus)
- Relational: === (case equality – where *'X'* and *'Z'* also have to match precisely), *!*== (case inequality), == (logical equality – where only *'0'* and *'1'* have to match. Result is 'X' if either operand contains *'X'* or *'Z'*), *!*= (logical inequality), <, <=, >, >=
- Logical: *!* (logical not), && (logical and), || (logical or)
- Bit-wise Operators: ~ (negation), & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ^~ or ~^ (xnor)
- Concatenation: *{,}* (used to combine bits). Ex: {a, b[2:0], 3'b010}
- Shift: << (left shift), >> (right shift)
- Replication: {<*n*>{<*m*>}} (replicate the value m, n times). Ex: {a, 2{b, c}} //equivalent to {a, b, c, b, c}
- Conditional: <*condition_expr*> ? <*true_expr*> : <*false_expr*> (select either the true or the false expression based on the condition expression)

If- Else and If- Else if Statements:

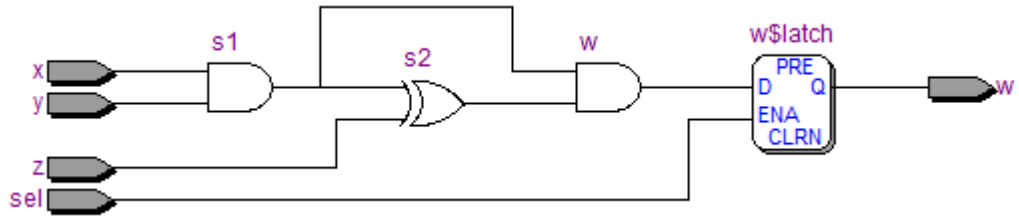
- An **if** statement executes a block of sequential statements upon matching certain condition(s).
- It can also include an **else** component or one or more **else if** components.

- *if* statements are only allowed within a procedural block. The statements are executed sequentially if a conditional match is detected.
- To avoid inferred latches, all outputs should be assigned values on all execution paths.
- Keep common statements outside of the *if- else if* statements.

Example: Inference

```
module if_example (input      x, y, z, sel,
                   output logic w);

       always_comb
       begin: inference
              logic   s1, s2;

              if (sel == 1'b1)
              begin
                s1 = x & y;
                s2 = s1 ^ z;
                w <= s1 & s2;  //Since w gets a value only conditionally, a latch
       is
              end              // inferred. Add an "else" statement or default w
       end                    // right after "begin" to avoid latches
endmodule
```

RTL Synthesis:



- To avoid the inferred latch, assign a default value to w outside the *if* statement. For example, you can add "w <= 0;" before the *if* statement. You can also add the same statement in the *else* part using an *if- else* statement. You will have to do the same thing even when using *always_comb*.

Case Statement:

- Equivalent to nested set of *if-else if* constructs but produces less logic since it does not force priority order.
- It identifies mutually exclusive blocks of code.
- Since all possible values of select inputs must be covered, use the *default* clause.
- **case** statements have to be inside a module. It can be used to describe multiplexors.
- **casez** statements treat 'Z' as *don't cares*, where a match is asserted when the rest of the word matches.
- **casex** statements treat both 'X' and 'Z' as *don't cares*.

Example: '*case*' Statement

```
module case_example(input a, b, sel,
                    output logic c);

always_comb
begin: case_process

  case (sel)
    1'b0 : c = a;
    1'b1 : c = b;
    default : c = 1'b0;
  endcase
end
endmodule
```

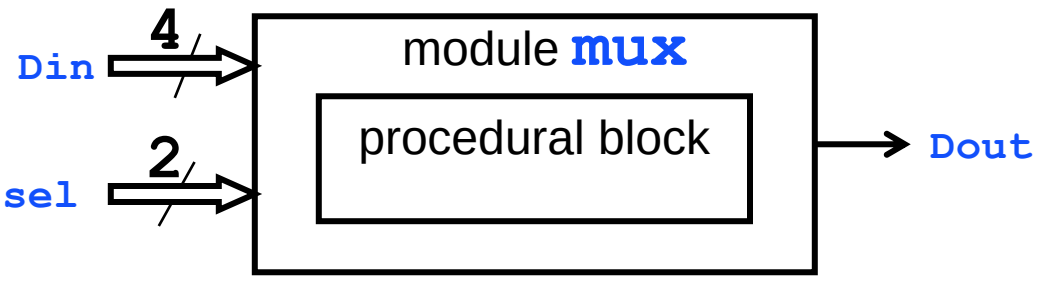
Example: '*casez*' Statement

```
module case_example(input a, b, sel,
                    output logic c);

always_comb
begin: case_process

  casez (sel)
    1'z0 : c = a;
    1'zx : c = b;
    default : c = 1'b0;
  endcase
end
endmodule
```

Note: The '*case*' example to the left employs precise matching, that is, '*sel*' has to be strictly either the value '1'b0' or '1'b0' in order for '*c*' to receive '*a*' or '*b*'. As for the '*casez*' example to the right, both case items will ignore their own <*radix*> when comparing against '*sel*', since 'Z' is treated as *don't cares* here. But they still need to match their <*value*>. For example, a '*sel'* of '1'h0' will only match the first case item, and a '1'hF' will only match the second case item.

- Although *case* statement and *if-else if* statements are logically similar, SystemVerilog will actually interpret them into different physical circuits, as shown below

Example: 4-to-1 Multiplexer

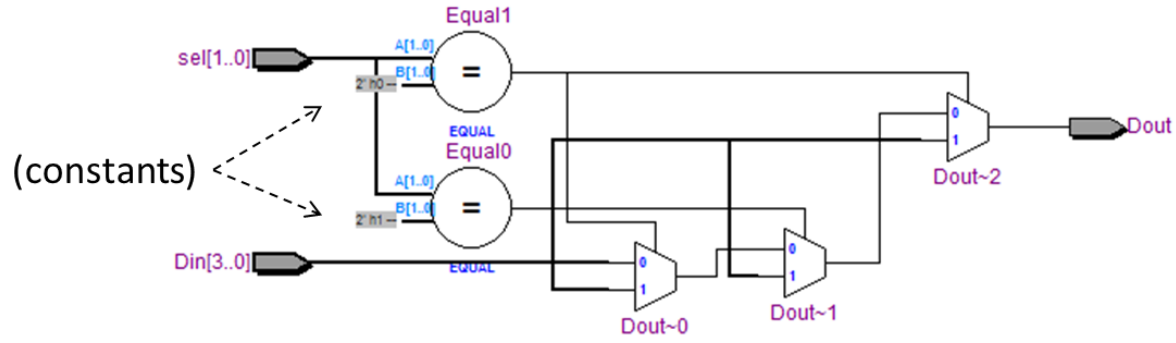


Design 1:

```
module mux (input [3:0] Din,
            input [1:0] sel,
            output logic Dout);

      always_comb
```

```
        begin
                if (sel == 1'b00)
                  Dout = Din[0];
                else if (sel == 1'b01)
                  Dout = Din[1];
                else if (sel == 1'b10)
                  Dout = Din[2];
                else
                  Dout = Din[3];
        end
endmodule
```

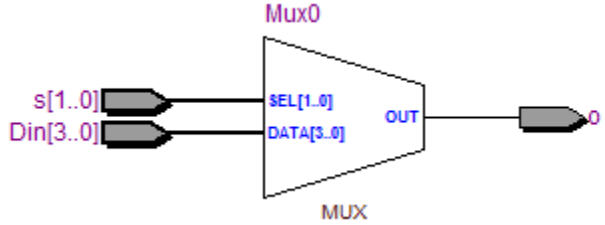
RTL Synthesis:



<u>Design 2</u>: (Produces less logic)

```
module mux (input [3:0] Din,
            input [1:0] sel,
            output logic Dout);

        always_comb
        begin
                case (sel)
                  2'b00  : Dout = Din[0];
                  2'b01  : Dout = Din[1];
                  2'b10  : Dout = Din[2];
                  default : Dout = Din[3];
                 endcase
        end
endmodule
```

RTL Synthesis:



- Notice how the *case* statement gives a cleaner circuit with less logic. In general, although there are many different syntactic approaches to achieve the same logical functionality, most often one will be better than the other due to the translation of the circuit. Therefore, it is somewhat important to try to write SystemVerilog in an organized manner and verify that the synthesized design is consistent with the intention by looking at the RTL diagrams.

Creating Hierarchical Design Using Components:

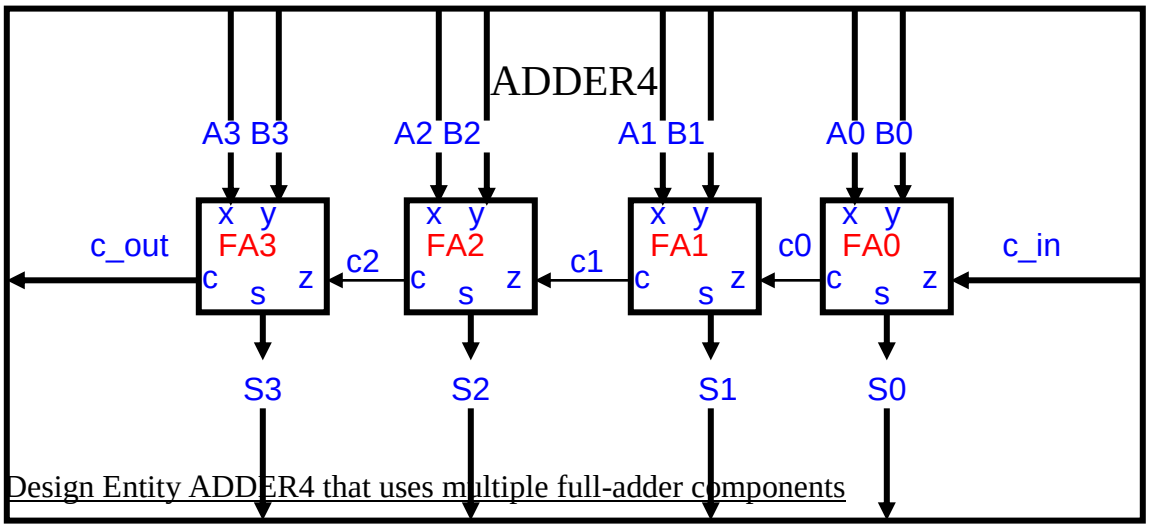
As mentioned earlier, you can create larger designs using modules as the components that make up the design. One module can instantiate other modules as components of it. One module can use different modules as components as well as instantiate multiple copies of the same module.

Example: Creating a 4-bit adder using a full-adder component

Design Entity Full-Adder

```
module full_adder (input x, y, z,
                   output logic s, c);

       assign s = x^y^z;
       assign c = (x&y)|(y&z)|(x&z);

endmodule
```

Use component full_adder to create a 4-bit adder:

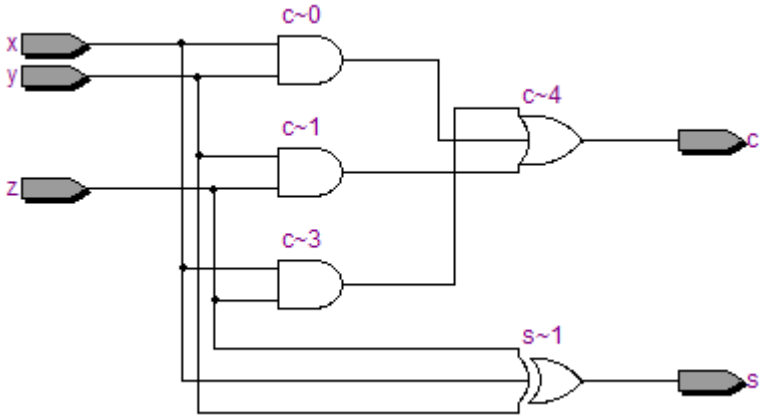


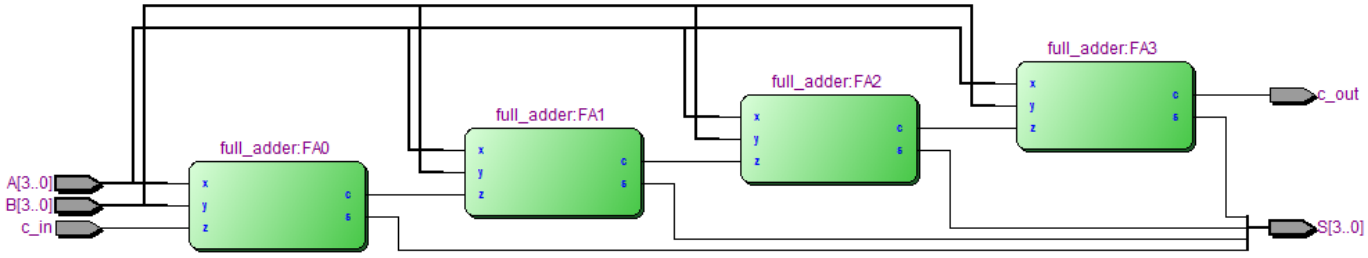
Design Entity ADDER4 that uses multiple full-adder components

```
module ADDER4 (input  [3:0] A, B,
               input        c_in,
               output logic [3:0] S,
               output logic       c_out);
```

```
    // Internal carries in the 4-bit adder
    logic         c0, c1, c2;

    /*=========================================*/
    // Netlists with named (explicit) port connection
   // Syntax: <module> <name>(.<parameter_name> (<connection_name>), …)
    full_adder    FA0 (.x (A[0]), .y (B[0]), .z (c_in), .s (S[0]), .c (c0));
    full_adder    FA1 (.x (A[1]), .y (B[1]), .z (c0), .s (S[1]), .c (c1));
    full_adder    FA2 (.x (A[2]), .y (B[2]), .z (c1), .s (S[2]), .c (c2));
    full_adder    FA3 (.x (A[3]), .y (B[3]), .z (c2), .s (S[3]), .c (c_out));

endmodule
```

RTL Synthesis:





To simplify the port connections and coding redundancy, SystemVerilog introduces implicit ".name" and ".*" port connections. This is based on the fact that most of the port connections share the same port names on each end of the connection. For the ".name" implicit port connection, SystemVerilog automatically connects ports that share the same name and size with the connecting **wire**. For the ".*" port connection, SystemVerilog automatically connects ports that share the same name and size without the need of rewriting them.

Example: .name connection

```
module top_level(input clock, reset, in,
                 output logic out);

        logic     tmp;
        logic [2:0] data;


        // module1 myModule1(input clock, reset, in, output result, output [2:0]
        data)
        module1 myModule1(.clock, .reset, .in, .result(tmp), .data(data));
        // module2 myModule2(input clock, reset, feed, output [1:0] data, output out)
        module2 myModule2(.clock, .reset, .feed(tmp), .data(data[1:0]), .out);

endmodule
```

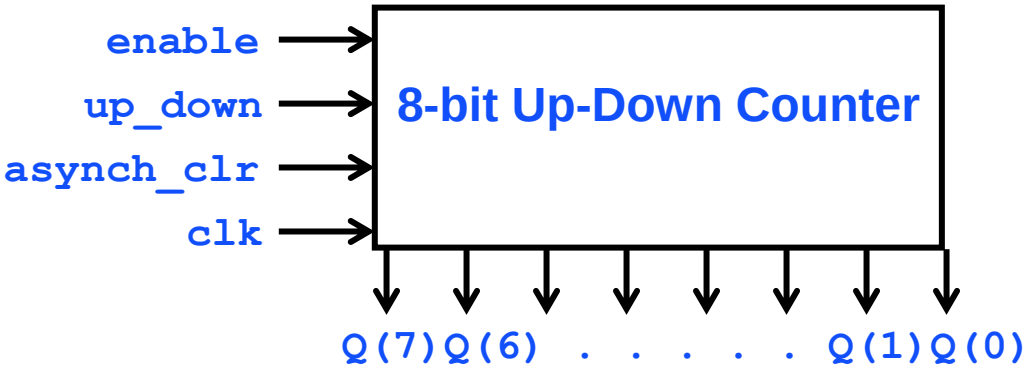
Note: The example above employs implicit port connection. If the name and size of the connections match (‘*clock*’, ‘*reset*’, ‘*in*’ and ‘*out*’ port in this case), then the matching ports will be automatically connected. However, if the name (‘*result*’/’*feed*’ ports) or the size (‘*[2:0]data*’/‘*[1:0]data*’ ports) of the connecting ports doesn’t match, we will need to write out the connection explicitly by adding a parenthesis behind the port name and indicate the specific wiring.

Example: .* connection

```
module top_level(input clock, reset, in,
                 output logic out);

        logic     tmp;
        logic [1:0] data;

        // .* “Wild card” port connection. Only need to write out the connection if
        the
        //  name or size doesn’t match
        module1 myModule1(.*, .result(tmp));
        module2 myModule2(.*, .feed(tmp));

endmodule
```

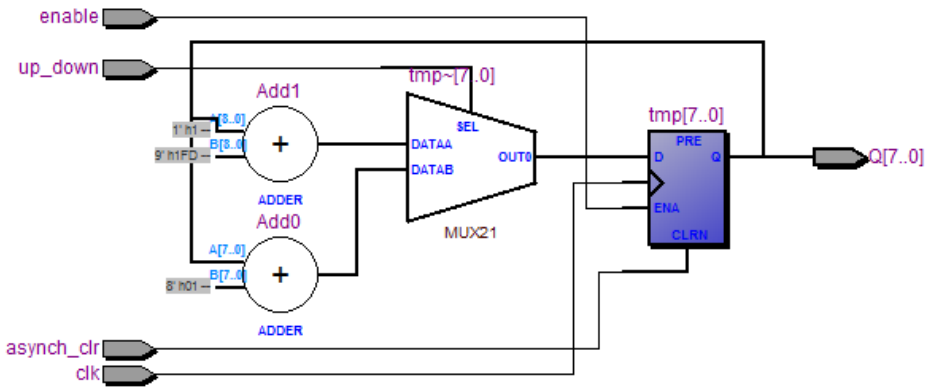
Note: In both implicit connections, an unconnected port must be explicitly named with blank connection. Ex: .<*port_name*>()

More Examples:
Example: Sequential Circuit




```
module counter (input clk, asynch_clr, enable, up_down,
                output logic [7:0] Q);

       always_ff @ (posedge clk or posedge asynch_clr)
/* Placing "posedge asynch_clr" in the sensitivity list enables clear whenever
  asynch_clr is on a rising edge. If "posedge asynch_clr" is not in the sensitivity
  list, then the circuit will only be cleared if asynch_clr is high during a rising
edge
  of the clk, i.e., the circuit will be synchronously cleared */

       begin
              if (asynch_clr)
                Q <= 8'b0;
              else if (enable)
                if (up_down)
                 Q <= Q + 1'b1;
                else
                 Q <= Q - 1'b1;
       end

endmodule
```

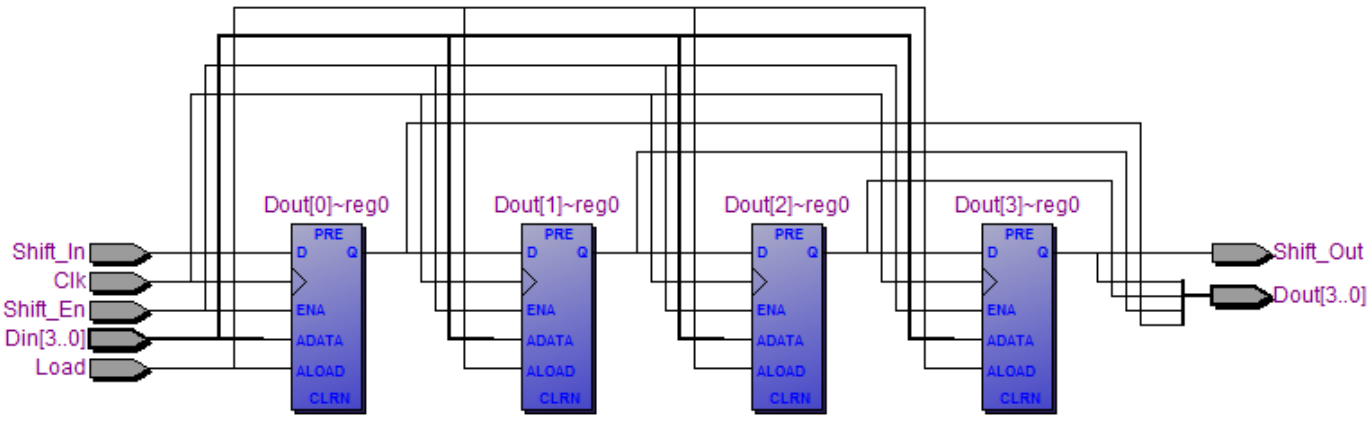
RTL Synthesis:



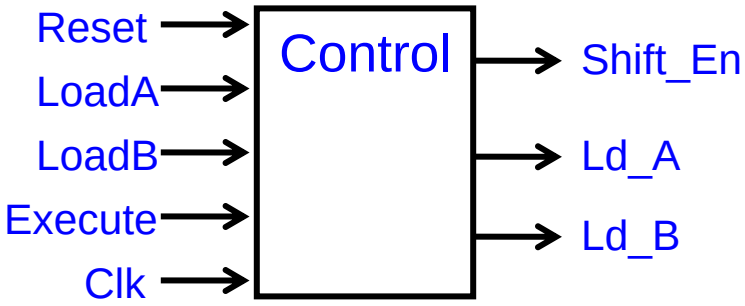
Example: N-Bit Left Shift Register


```
// #(N=4) is a variable parameter
module shiftReg #( N = 4) (input         Clk, Shift_En, Shift_In, Load,
                           input [N-1:0]  Din,
                           output logic [N-1:0] Dout,
                           output logic         Shift_Out);

       always_ff @ (posedge Clk)
       begin
              if (Load)
                Dout <= Din;
              else if (Shift_En)
                Dout <= { Dout[N-2:0], Shift_In };   // Concatenation
       end

       assign Shift_Out = Dout[N-1];

endmodule                  // shiftReg
```

RTL Synthesis:



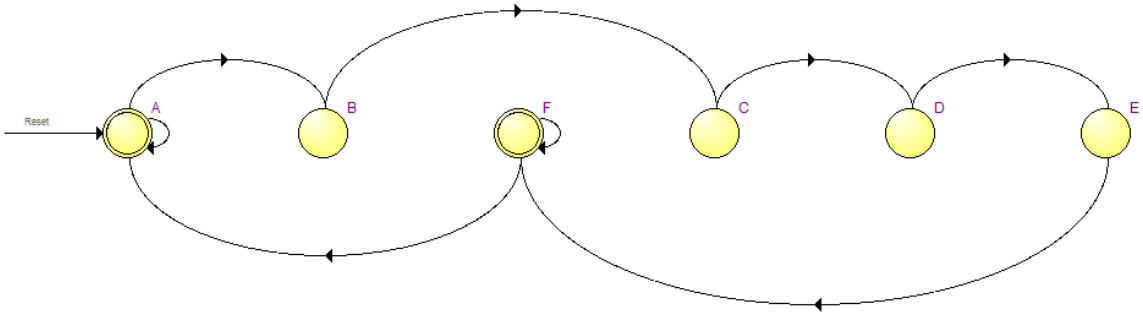
State Machine Design:
Example: Control Unit




```
module control (input Clk, Reset, Execute, LoadA, LoadB,
                output logic Shift_En, Ld_A, Ld_B);

       enum logic [2:0] {A, B, C, D, E, F}  curr_state, next_state;  // Internal
state logic

       // Assign 'next_state' based on 'state' and 'Execute'
       always_ff @ (posedge Clk or posedge Reset ) // Sequential logic
       begin
              if (Reset)
                curr_state <= A;  // alternatively use the enum method
              else          //  <enum_name>.first to point to the first enum value
                curr_state <= next_state;
       end

       // Assign outputs based on ‘state’
       always_comb
       begin
            // Default to be self-looping (stay in current state unless triggered),
so
            // therefore no potential latch will be inferred
             next_state = curr_state;

       // The keyword “unique” asserts that there are no overlapping cases. It will
       //  generate warnings when multiple case item expressions are true at the
      //  same time
              unique case (curr_state)
                // use <enum_name> = <enum_value> to assign the next state, or
               //   use <enum_name>.next to go to the immediate next state
                 A :   if (Execute)          // conditional state transition
                         next_state = B;
                 B :    next_state = C;    // deterministic state transition
                 C :    next_state = D;
                 D :    next_state = E;
                 E :     next_state = F;
                 F :   if (~Execute)
                         next_state = A;
              endcase
       end
```

```
        // Assign outputs based on 'state'
        always_comb
        begin
                case (curr_state)
                  A:
                   begin
                     Ld_A = LoadA;
                     Ld_B = LoadB;
                     Shift_En = 1'b0;
                   end
                  F:
                   begin
                     Ld_A = 1'b0;
                     Ld_B = 1'b0;
                     Shift_En = 1'b0;
                   end
                  default:
                   begin
                     Ld_A = 1'b0;
                     Ld_B = 1'b0;
                     Shift_En = 1'b1;
                   end
                endcase
        end

endmodule
```

RTL Synthesis:



Verification:

After designing and implementing your circuit in SystemVerilog, the next step is to make sure that it works according to the design requirement free from bugs through the verification process. A simulation instantiates the top level module of your circuit and does the following things:

- Generates the input vector waveform
- Apply the input waveform to the DUT (your circuit)
- Outputs the response

Once the circuit response is obtained, you can then verify it to see if it complies with the design requirement. If any response does not comply properly, you should trace back the specific faulty response all the way back to where the design flaw is generated and try to fix

it. You should always verify your circuit as complete as possible to avoid unexpected flaws.

There are two ways to verify your circuit. The first one is using the SystemVerilog testbench module. SystemVerilog is a very powerful verification language, with constructs borrowed from the software languages. The compiler will generate the input signals according to the constructed testbench module, and the simulated result will be outputted. The testbench approach is especially useful when dealing with large and complicated circuits, which usually requires the generation of numerous inputs and the verification of numerous outputs.

If your circuit is simple and straightforward, then using the GUI waveform generator approach might be more convenient. Many HDL compilers come with built-in or custom-made GUI waveform generators which allow the designers to directly generate the input signals and view the output signals graphically. The Quartus II compiler we will be using employs such a custom-made GUI waveform generator. Please refer to IQT. 6-9 for a tutorial on the ModelSim-Altera GUI verification, and IQT. 29-34 for a tutorial on using testbenches in ModelSim-Altera.

Other Notes/Hints:

- A design that simulates is not guaranteed to synthesize. Simulation tools allow simulating designs that may not be physically possible to implement.
- It is always a good idea to assign values to outputs in all possible cases for any diverging statement. This way we can avoid inferred latches. If we do not specify the value of a signal for some input combination, then the design will have inferred latch(es) because it will try to hold the old value of the signal.
- Keep in mind that HDL statements are usually executed concurrently and not sequentially.
  - When a value is assigned to a signal, it does not take effect until the next simulation cycle, unless you specify it to be updated immediately, i.e., through the blocking statement.
- Include all inputs that can change the outputs in the sensitivity list of a procedure to avoid different circuit behavior from simulation and synthesis.
  - You can manipulate when the procedure executes by including/excluding certain inputs from the sensitivity list, but that will only work for the simulation model. If the synthesized design is combinational, then the logic will respond to events on any inputs, not just the inputs in the sensitivity list.
- SystemVerilog allows delay/wait statements, but that is only guaranteed during simulation. The synthesized design may not have similar delays.
  - For example: #5 x <= y; // delay x <= y by 5 time units.
    - ➢ This is allowed, but synthesis behavior is not guaranteed.
- SystemVerilog compilers generally allow looping statements (**for**, **while**, **do**…**while**, …), but synthesis tools will only be able to synthesize it if the statement can be unrolled and determined during the synthesis, i.e., the looping statements are to save repetitive statements and are not intended to execute run-time dependent numbers of cycles.

- Do not specify initial values in variable declarations. Synthesis compilers cannot synthesize the **initial** procedural block. Initial values can be assigned explicitly under a controlling signal (e.g. reset).
- Assign *don't care* values to signals when possible for default cases. It may allow the synthesis compiler to produce a better design.
- Case statements produce less logic than *if-else* statements since *if-else* statements produce priority logic also.
- Do not assign value to the same signal via multiple statements that can execute simultaneously. The compiler will either handle only the last assignment or produce an error.
- Write your code for synthesis rather than just simulation, as you will need to synthesize your designs for all experiments involving SystemVerilog.

References:

[1] "SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog", Accellera, 2004. Available at: http://www.eda.org/sv/

[2] S. Sutherland, S. Davidmann and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, 2nd ed. Springer, 2006.

[3] Yalamanchili, Sudhakar. *Introductory VHDL – From Simulation to Synthesis*. New Jersey: Prentice-Hall, 2001.

[4] "A Proposal for a Standard Synthesizable Subset for SystemVerilog-2005: What the IEEE failed to Define", by Stuart Sutherland. presented at DVCon, March 2006. Available at http://www.sutherland-hdl.com/papers/2006-DVCon_SystemVerilog_synthesis_subset_paper.pdf

INTRODUCTION TO
QUARTUS PRIME AND TUTORIAL

ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to Quartus Prime

A Simple Design in SystemVerilog Using Altera Quartus Prime 18.1 Lite Edition

Design a full-adder and use it to design a 2-bit adder:

You need to be on a machine with Linux or Windows 7, 8, or 10, either on your own machine or in the ECE Open Lab (3022 ECEB). A Linux version is also available online for your own Linux system. To perform this exercise on your own computer, you will need to install the Quartus Prime software (you may download the installer from the link below). Follow these steps to fully install the software:

- Install Quartus Prime Lite Edition v18.1, ModelSim-Intel FPGA Edition, and Cyclone IV device support from the Intel FPGA website (https://fpgasoftware.intel.com/18.1/?edition=lite&platform=windows&download_manager=dlm3). You do not have to install the three downloads separately. Rather, if the downloads are placed in the same directory, they will all be automatically selected for installation through the main installer, QuartusSetupWeb-XXX-XXX.exe.
- Once the installation has completed, check the box ***Launch USB Blaster II driver installation*** upon existing the Quartus installation, then follow the prompt to complete the USB Blaster driver installation.
- Alternatively, you can manually install the Altera USB Blaster driver through the following steps:

    1. Plug the USB-Blaster download cable into PC. The ***Found New Hardware*** dialog box will appear.
    2. Select ***Locate and install driver software (recommended)***.
    3. Select ***Don't search online***.
    4. When you are prompted to ***Insert the disc that came with your USB-Blaster***, select ***I don't have the disc. Show me other options***.
    5. Select ***Browse my computer for driver software (advanced)*** when you see the ***Windows couldn't find driver software for your device***.
    6. Click ***Browse*** and browse to the ***<Path to Quartus Prime installation>\drivers\<cable type>*** directory.
        - For USB-Blaster cables, the driver is in the ***<Path to Quartus Prime installation>\drivers\usb-blaster***
        - Note: Do not select the x32 or x64 directories.
    7. Click ***OK***.
    8. Select the ***Include subfolders*** option and click ***Next***.
    9. If you are prompted ***Windows can't verify the publisher of this driver software***, select ***Install this driver software anyway*** in the ***Window Security*** dialog box.

10. The installation begins.

- Make sure the ModelSim-Altera directory is entered properly.

1. Go to ***Tools***->***Options*** from the menu bar. Navigate to the ***EDA Tool Options*** page.
2. Enter the corresponding directory next to ***ModelSim-Altera***. The default path on Windows is < ***Path to Quartus Prime installation >\18.1\modelsim_ase\win32aloem***.
3. Click ***OK***.

You are now ready to begin using Quartus. Below is a sample 2-bit adder project.

Create a New Project:

- From the ***File*** menu select ***New Project Wizard***. Click *Next* to pass the intro screen.
- The window in Figure 1 will appear. Fill in the fields from Figure 1 (make sure there are no spaces in any of your entries). The program will ask you if it should create the specified directory if it does not exist; choose *yes*.
- Select ***Next*** on page 2 without adding any files.
- On page 3, select the device family ***Cyclone IV E***, make sure the second option under Target device is selected, and chose ***EP4CE115F29C7*** in the Available devices list. See Figure 2.
- Click ***Next*** on page 4. Select ModelSim-Altrea as the simulation tool, and SystemVerilog HDL as the simulation format. See Figure 3. Click ***Finish*** on page 5.
- You should see an entry for the project in the ***Project Navigator*** window. It should appear as in Figure 4.

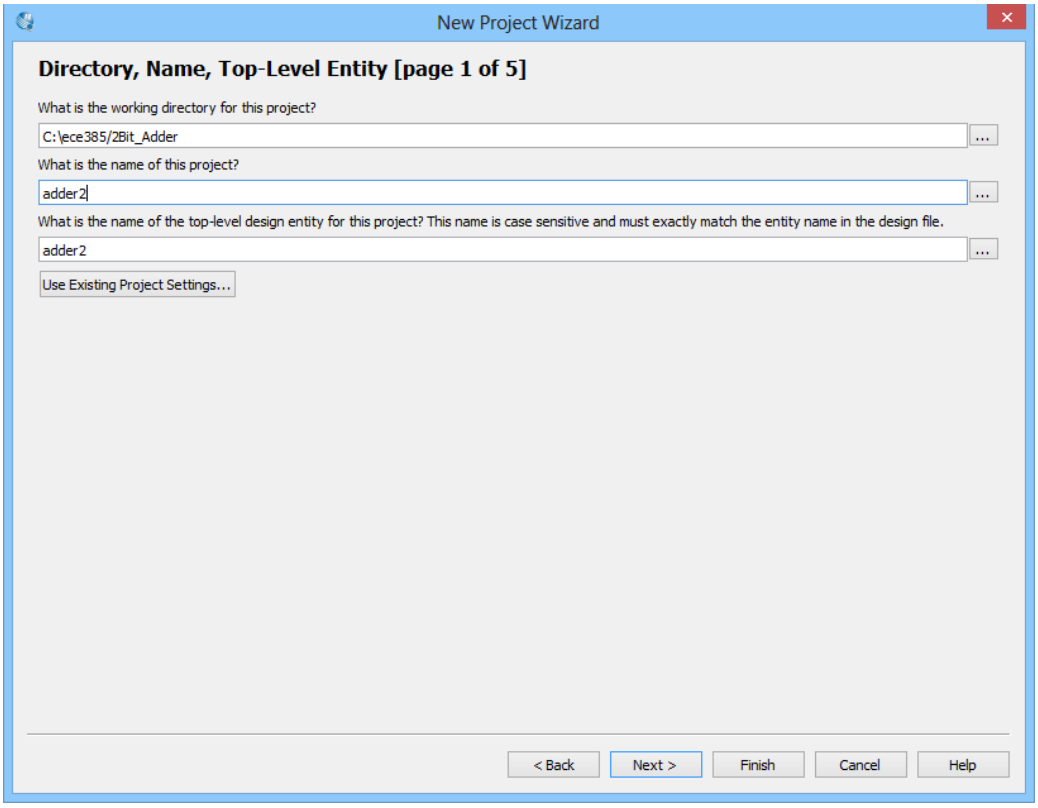


Figure 1

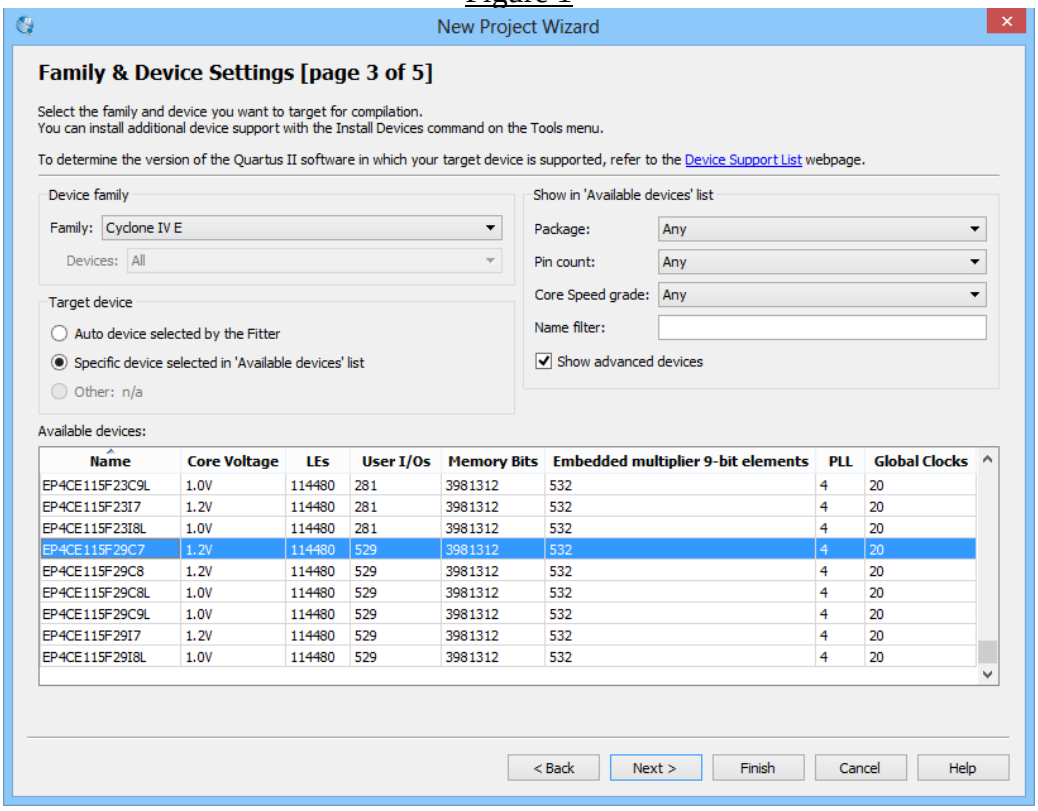


Figure 2

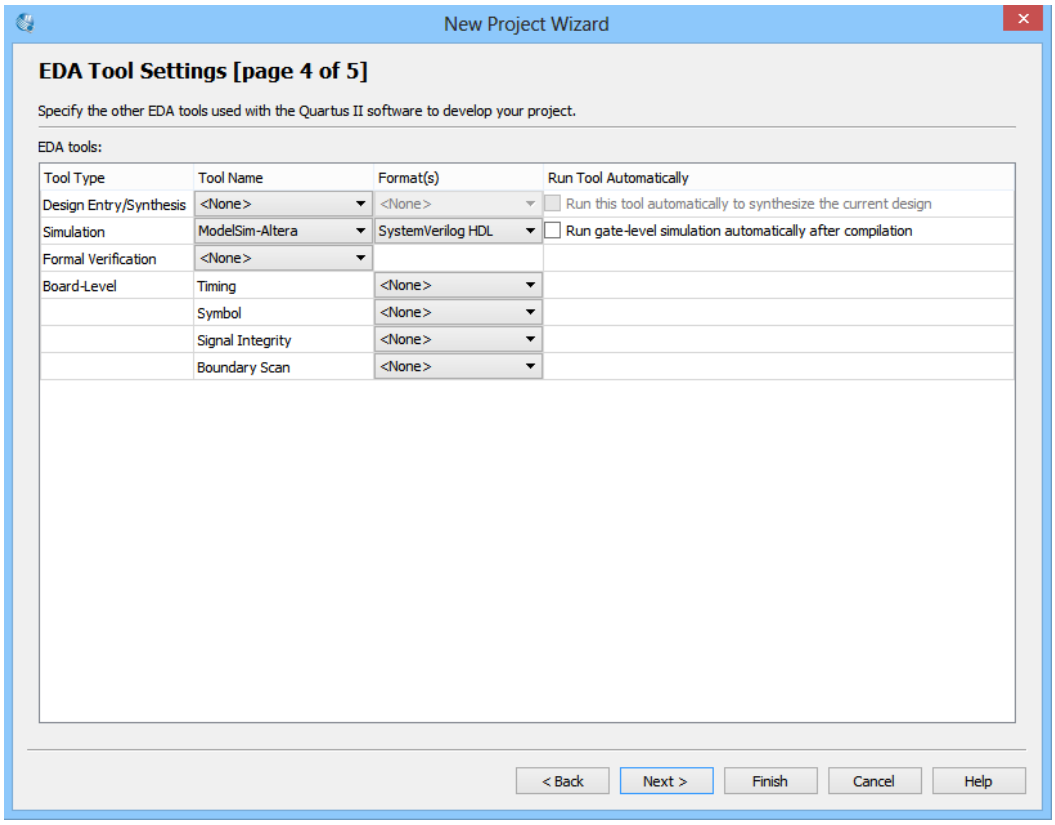


Figure 3

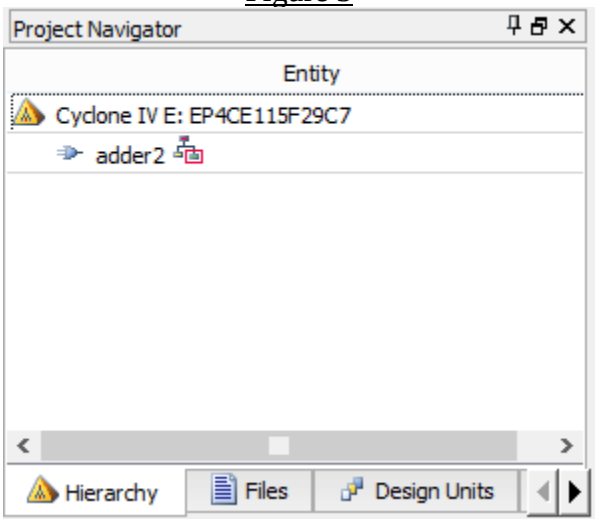


Figure 4

Create a Full-Adder Entity:

Now you can start entering your design. In the ***File*** menu, select *New…*. Select ***SystemVerilog HDL File*** in the window that pops up as shown in Figure 5 and click ***OK***. You will be presented with a blank SystemVerilog file. Enter the code below to define the full adder unit. Save the file as full_adder.sv. Figure 6 shows schematic representations of what the code defines; if you do not

understand what the code means, you should review the Introduction to SystemVerilog on pages IST.1-25.

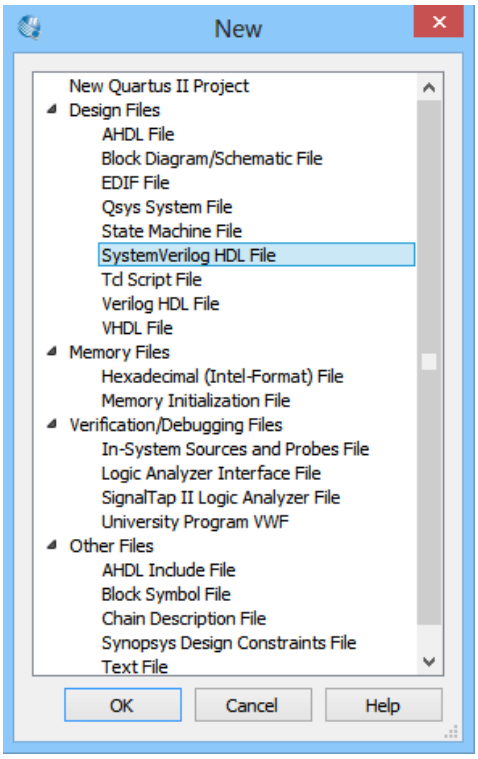


Figure 5

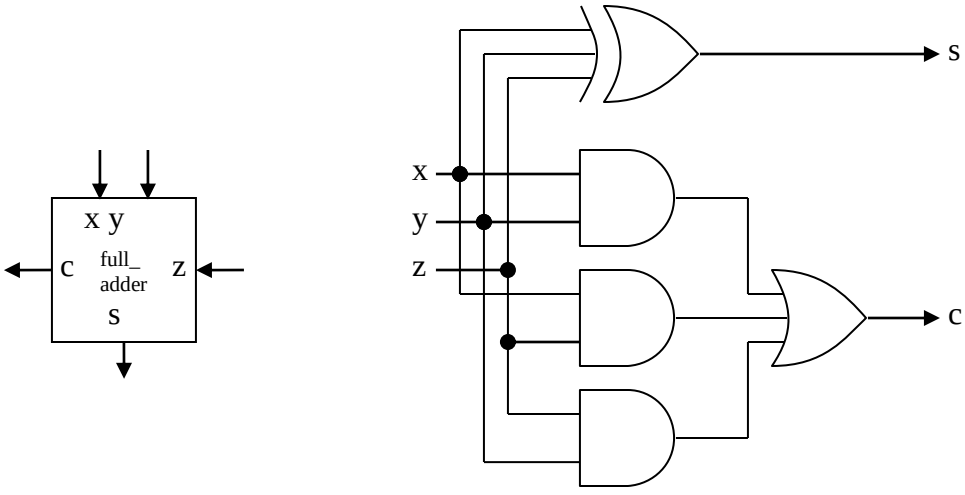


Figure 6: Schematic depiction of the meaning behind the Full Adder (a) `module` block and (b) `combinational logic`.

```
module full_adder (input x, y, z,
      output s, c);
```

```
    assign s = x^y^z;
    assign c = (x&y)|(y&z)|(x&z);

endmodule
```

Create an Entity for 2-Bit Adder:
Now, follow the same steps to create a new file that defines a 2-bit adder with inputs A (2-bits), B (2-bits) and Cin, and outputs Sum (2-bits) and Cout. The code is given below. Save the file as adder2.sv.

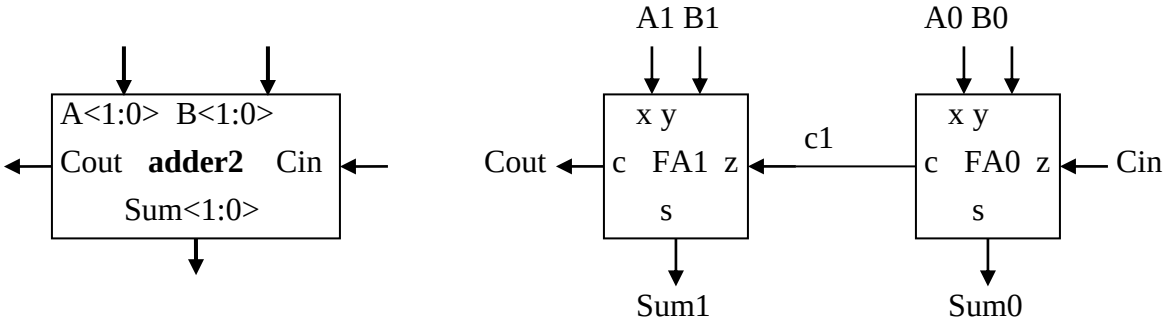


Figure 7: Module depiction of the meaning behind the 2-Bit Adder
(a) `module` block and (b) `procedural` block.

```
module adder2 (input [1:0] A, B,
    input   c_in,
    output [1:0] S,
    output  c_out);

        // Internal carries in the 2-bit adder
        logic    c1;

        full_adder  FA0 (.x (A[0]), .y (B[0]), .z (c_in), .s (S[0]), .c (c1));
        full_adder  FA1 (.x (A[1]), .y (B[1]), .z (c1), .s (S[1]), .c (c_out));

endmodule
```

Once you have entered and saved the code for adder2.sv, click the ***Start Analysis & Synthesis*** button () in the toolbar. Quartus Prime will check the code for correct syntax and generate errors or warnings if there is anything wrong or questionable about your code. A message pops up when the program has finished; you should get no errors or warnings if you have entered the code correctly. If you find errors in synthesis, correct the code, save the file, and run ***Analysis & Synthesis*** again.

Quartus Prime also builds the hierarchy in the ***Project Navigator*** window, which should look like figure 7. Next, click the ***Start Compilation*** button ( ) in the toolbar. This time, you will get a few warnings about timing characteristics and load capacitances. You can ignore these. Now you have created a 2-bit adder and you can simulate the design.

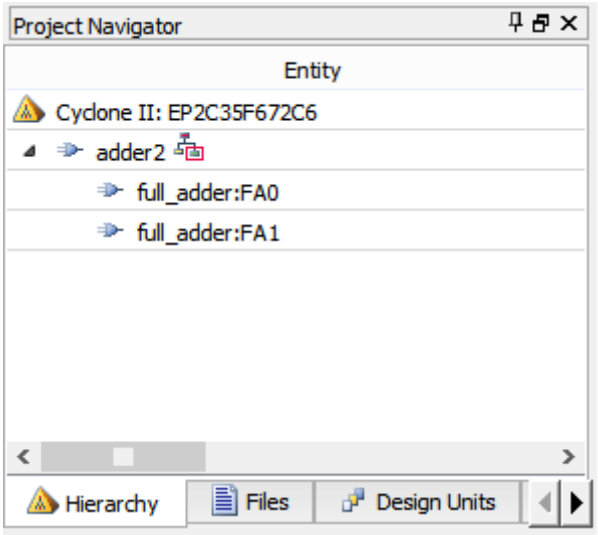


Figure 8

Using Simulator:

To create a testbench for your design and do simulations in ModelSim, please follow the instructions starting from IQT. 20.

Entering Pin Assignments:

Click the ***Pin Planner*** button ( ) in the toolbar to open the Pin Planner. You will see the pin layout of the Cyclone IV chip on the top right of the window, as well as the pin assignment table at the bottom of the window. Enter the assignments from Table 1 in sequence. The planner should look like Figure 9. Save the assignment and recompile the design ( ). Rows with white background mean that those signals have not been assigned to any pins, and rows which have been assigned to pins will appear with color background.

Port Name	Location	Comments
Cin	PIN_AB28	On-board slider switch (SW0)
A[0]	PIN_AC28	On-board slider switch (SW1)
A[1]	PIN_AC27	On-board slider switch (SW2)
B[0]	PIN_AD27	On-board slider switch (SW3)
B[1]	PIN_AB27	On-board slider switch (SW4)
S[0]	PIN_G19	On-Board LED (LEDR0)
S[1]	PIN_F19	On-Board LED (LEDR1)
Cout	PIN_E19	On-Board LED (LEDR2)

Table 1: Pin Assignments

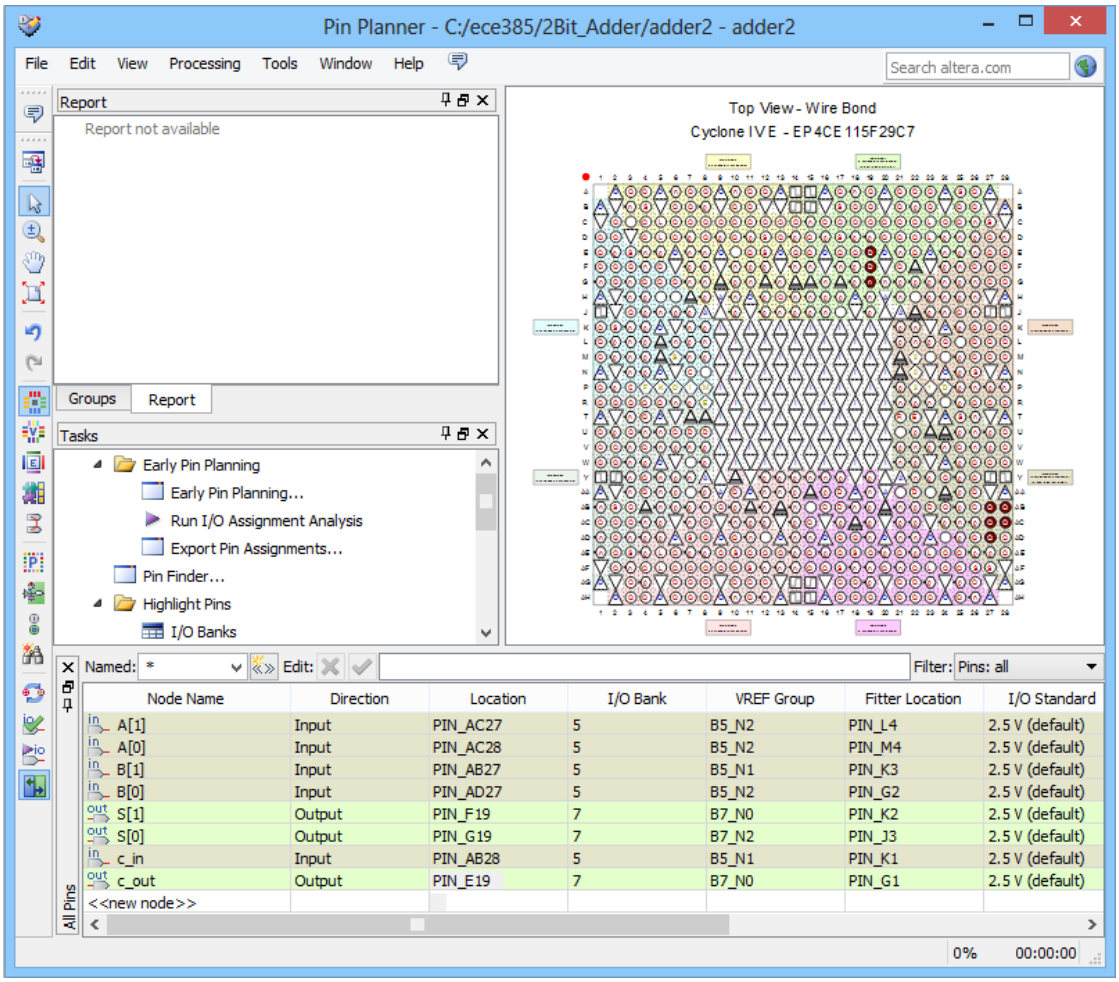


Figure 9

Programming the FPGA:

Plug in the FPGA power, and connect the FPGA Blaster port to the computer with the included USB cable. Click the ***Programmer* button** () to open the programmer. The screen should list the file *adder2*.sof. Turn on the FPGA power, and click the ***Hardware Setup…*** button. Select ***USB Blaster*** from the ***Currently selected hardware*** dropdown list, and click ***Close***. Back in the programmer, check the box next to adder2.sof under ***Program/Configure***. Finally, click the ***Start*** button. The FPGA will be programmed with your design. Toggle the switches and verify that the circuit is performing correct 2-bit addition with carry-in. Note that there is a switch on the bottom left of the FPGA board marked with “Run” and “Program”. Please leave that switch at “Run” even when you are programming the FPGA.

Instructions for Performing Experiments Involving SystemVerilog in the Lab

Creating a Project

- Start Quartus Prime.
- From the ***File*** menu, select ***New Project Wizard***.
- Enter a location, name, and top-level entity for the project. Create your project on the W: drive.
- Add any files that you have brought with you to the lab (you may need to copy them to the directory you specified above first).
- Select ***Cyclone IV*** for Device Family
- Select ***EP4CE115F29C7*** for Device
- Select ***ModelSim-Altrea*** as the simulation tool name, and ***SystemVerilog HDL*** as the simulation format. Keep clicking ***Next*** until you get to ***Finish***.
- You can see in the ***Project Navigator*** window at the left of the screen that a project has been created.

Adding Files to a Project

- If you didn't add your files in the step above, or if you wish to add additional files, right click on ***Cyclone IV: EP4CE115F29C7*** in the ***Project Navigator*** window on the left of the screen and click ***Settings***. Select ***Files*** under ***Category***. You can add files by entering their names directly or by finding them with the file browser; the ***add all*** button will add all vhd files in the project directory. Get the testbench files (*.vwf) from your TA and add to the project the same way as above. Additionally, in the ***Settings*** window, select ***Simulator*** in the ***Category*** box, and set the simulation ***Time scale*** to "1ns".

Analyzing, Synthesizing, and Implementing the Design

- Click the ***Start Compilation*** () button to begin the compilation process. If there are no errors, you will see each of the steps complete to 100%. If there are errors, you can view them on the ***error*** tab of the ***Messages*** window.
- You will need to fix any errors in your design before continuing. Additionally, you may receive warnings from the various steps. You should correct all warnings from the ***Analysis & Synthesis*** step. Warnings from the other steps may be acceptable; check with your TA.

Viewing the Synthesized Design

- Once the design is fully compiled, you can view the synthesized circuit in ***Tools > Netlist Viewers > RTL Viewer*** in an interactive GUI. If your design contains a state machine, and if your state machine is well structured, Quartus Prime should be able to extract the state machine for you as well. It can be viewed in ***Tools > Netlist Viewers > State Machine Viewer***. This may help you generate the block diagrams required in lab reports.

Simulating the Design

- Once you are ready to run simulations on your design, click the *RTL Simulation* () button. This will bring up the ***ModelSim-Altrea*** simulator, which will automatically generate the simulation netlist.

Setting pin mappings and programming the DE2 Board
Once your design simulates correctly, you can download the design bit-stream onto the DE2 board and test it.

- First you have to provide pin assignments for inputs and outputs. Click the ***Pin Planner*** button ().
- This will open the Pin Planner Editor. The list of circuit input and output pins should have been automatically listed at the bottom portion of the window. The list of assignments can be found in the lab manual entry for the current lab. Type in these assignments. (If you prefer, you can double-click the fields in the editor and select signals and pins from a dropdown menu.)
- Once you have entered the pin assignments for all ports, save the assignment editor view and recompile the design ().

Downloading the design to the DE2 Board
The Programmer sends the completed bitstream to the FPGA.

- Connect the programming cable to the computer's USB port and the board's ***USB Blaster*** connector.
- Power on the dev board.
- Click the ***Programmer*** button () to open the programmer. You should see a ***.sof*** file bearing the same name as the top-level entity in the window. If it does not appear automatically, you can add it from the "output files" folder.
- If the box next to the ***Hardware Setup…*** button reads ***No Hardware***, click the ***Hardware Setup…*** button and select the ***USB-Blaster*** option.
- If ***Select Device*** is prompt, select ***EP4CE115***.
- Check the box under ***Program/Configure*** to select the programming file.
- Make sure the slider switch next to the LCD display on the DE2 board is set to ***Run***, not ***Prog***.
- Click ***Start***. The FPGA will be programmed with your design.
- Once the programming process completes, test your design for functionality.
- If your design does not work or exhibits bugs, alter your design, recompile, and reprogram the FPGA. If you cannot easily determine how to fix your design, design a simulation to capture the incorrect behavior to better understand what is going wrong.
- **If the FPGA does not program correctly, ensure the RUN/PROG switch on the DE2-115 board is set to RUN**

I/O Pins available on the DE2-115 Development Board

Pins used in the lab exercises are listed in the manual description for each experiment. A full listing of the devices and associated pins on the DE2 development board can be found in the DE2 User Manual, which can be found on the DE2 System CD in the file *\DE2_115_user_manual\DE2-115_UserManual.pdf*. The contents of the DE2-115 System CD can be downloaded at **http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html** if you do not have the CD handy.

A spreadsheet containing all the pins listed here can also be found on the DE2 System CD, in the file *\DE2_115_lab_exercises\DE2_115_pin_assignments.csv*. You may use this file to easily copy and paste groups of assignments into the Assignment Editor.

Quartus Tutorial Exercise – Bit-Serial Processor in SystemVerilog

The 8-bit serial processor will be very similar to the design you have done for Experiment 3. The only difference is that you will now use two 8-bit shift registers to store the data inputs and the result. You will first load the registers with the values that you want to be used as operands. Then you will specify the operation that you want to perform on the data and specify the register that should hold the result. The result will be computed using bit-wise operators. Every cycle during computation, the least significant bits from each register will be shifted out, desired logic operation will be performed on the two bits, and the result bits will be shifted into the registers as the most significant bits. After repeating this procedure eight times, the specified register will contain the 8-bit result.

Task: Download the 4-bit module files from the course website (under Lab 4). Use it to build an 8-bit serial processor project using the provided modules by creating a project, adding the provided files to the project and test the circuit operation using the included testbench. Follow the SystemVerilog tutorial ISV and the instructions for testbench (**IQT. 23-30**) to complete the exercise. Include a copy of the schematic block diagram and the simulation waveform (with annotations) in your Lab 4 lab report.

Your circuit should have the following inputs and outputs (once it has been extended to 8-bits):

Inputs

```
Clk, Reset, Execute, LoadA, LoadB – logic
Din – logic [7:0]
F – logic [2:0]
R – logic [1:0]
```

Descriptions of the inputs:

- `Clk`: Clock signal for the processor
- `Reset`: Reset signal to initialize the controller to the start state.
- `Execute`: Signal that triggers the execution of a computation cycle.
- `LoadA`: Signal that loads Register A with the data specified by the input switches (`Din<7:0>`).
- `LoadB`: Signal that loads Register B with the data specified by the input switches (`Din<7:0>`).
- `Din<7: 0>`: Input switches for the input data
- `F<2:0>`: Function select
- `R<1:0>`: Router select

Outputs

`Aval, Bval – logic [7:0]`
`AhexU, AhexL, BhexU, BhexL – logic [6:0]`
`LED – logic [3:0]`

Descriptions of the outputs:

`Aval<7:0>, Bval<7:0>`: Binary values in registers A, B
`AhexU<6:0>, AhexL<6:0>`: 7-segment display driver signals (Upper and Lower nibbles) for register A
`BhexU<6:0>, BhexL<6:0>`: 7-segment display driver signals (Upper and Lower nibbles) for register B
`LED<3:0>`: Concatenated values of `Execute`, `LoadA`, `LoadB`, & `Reset` signals, to be displayed on LEDs.

Note: For Experiments 4-9, you should use the same input/output port names as provided for each experiment in this manual.

The block diagram for the circuit is shown in Figure 14. The register unit contains two 8-bit registers, Register A and Register B. The computation unit performs the desired logical operation based on the function select (`F<2:0>`). The routing unit selects the input bits to Register A and Register B after the computation. The control unit provides control signals (e.g. load, shift) to the register unit.

Control Unit

The control unit will accept `Clock`, `LoadA`, `LoadB`, `Execute`, and `Reset` as inputs. It will output the control signals to the register unit to instruct the registers when to load the values from the switches and when to shift the register values. The Reset signal, when set to high, will put the controller in the reset/start state. A 0→1 transition on the Execute signal will trigger the execution of a computation cycle. Once the computation cycle has started, the controller should ignore the Execute signal until the appropriate logical operation is performed on all bits and the registers contain appropriate 8-bit values. That is, during execution, the Execute signal is a 'Don't Care'. The processor should halt at the end of a computation cycle until the Execute signal is set to low. Another computation cycle can then begin when the Execute signal again switches from low to high. Load A and Load B should only be allowed to take effect in the reset/start state and should be ignored during the execution of a computation cycle.

Register Unit

The register unit will contain two 8-bit registers, Register A and Register B. The control signals to parallel load the values of the data inputs (Din<7:0>) to one or both of the registers and to shift in the result bits from the routing unit will come from the control unit. During a computation cycle each register will shift-out one bit at a time to the computation unit as the result bit from the routing unit shifts in. The 8-bit values of the registers will also be provided as outputs that the user can observe.

Computation Unit

The computation unit will perform a logical operation based on the function select (F<2:0>) on the operands provided by the register unit. The computation unit will output three one-bit values to the routing unit – A, B, and F(A, B). Table 1 provides the functions associated with the value of F<2:0> and the values that should be transmitted to the register unit from the routing unit based on R<1:0>.

Routing Unit

The routing unit will accept A, B, and F(A, B) as inputs and will output A' (shift-in to Register A, newA) and B' (shift-in to Register B, newB) based on R<1:0> as shown in Table 1.

HexDriver Units

Each HexDriver unit accepts one 4-bit hex digit as input and outputs the 7 bits necessary to correctly display this digit on a 7-segment display. You will instance 4 HexDriver units, because you need to display 16 bits of data.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Execute	PIN_R24	On-Board Push Button (KEY3)
LoadA	PIN_N21	On-Board Push Button (KEY2)
LoadB	PIN_M21	On-Board Push Button (KEY1)
Reset	PIN_M23	On-Board Push Button (KEY0)
Din[0]	PIN_AB28	On-board slider switch (SW0)
Din[1]	PIN_AC28	On-board slider switch (SW1)
Din[2]	PIN_AC27	On-board slider switch (SW2)
Din[3]	PIN_AD27	On-board slider switch (SW3)

Din[4]	PIN_AB27	On-board slider switch (SW4)
Din[5]	PIN_AC26	On-board slider switch (SW5)
Din[6]	PIN_AD26	On-board slider switch (SW6)
Din[7]	PIN_AB26	On-board slider switch (SW7)
R[0]	PIN_AA24	On-board slider switch (SW13)
R[1]	PIN_AA23	On-board slider switch (SW14)
F[0]	PIN_AA22	On-board slider switch (SW15)
F[1]	PIN_Y24	On-board slider switch (SW16)
F[2]	PIN_Y23	On-board slider switch (SW17)
Aval[0]	PIN_J15	On-Board LED (LEDR10)
Aval[1]	PIN_H16	On-Board LED (LEDR11)
Aval[2]	PIN_J16	On-Board LED (LEDR12)
Aval[3]	PIN_H17	On-Board LED (LEDR13)
Aval[4]	PIN_F15	On-Board LED (LEDR14)
Aval[5]	PIN_G15	On-Board LED (LEDR15)
Aval[6]	PIN_G16	On-Board LED (LEDR16)
Aval[7]	PIN_H15	On-Board LED (LEDR17)
Bval[0]	PIN_G19	On-Board LED (LEDR0)
Bval[1]	PIN_F19	On-Board LED (LEDR1)
Bval[2]	PIN_E19	On-Board LED (LEDR2)
Bval[3]	PIN_F21	On-Board LED (LEDR3)
Bval[4]	PIN_F18	On-Board LED (LEDR4)
Bval[5]	PIN_E18	On-Board LED (LEDR5)
Bval[6]	PIN_J19	On-Board LED (LEDR6)
Bval[7]	PIN_H19	On-Board LED (LEDR7)
LED[0]	PIN_E21	On-Board LED (LEDG0)
LED[1]	PIN_E22	On-Board LED (LEDG1)
LED[2]	PIN_E25	On-Board LED (LEDG2)
LED[3]	PIN_E24	On-Board LED (LEDG3)
AhexL[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
AhexL[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
AhexL[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
AhexL[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
AhexL[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
AhexL[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
AhexL[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
AhexU[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
AhexU[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
AhexU[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
AhexU[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
AhexU[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
AhexU[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
AhexU[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
BhexL[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
BhexL[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
BhexL[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
BhexL[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
BhexL[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
BhexL[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
BhexL[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
BhexU[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
BhexU[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
BhexU[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])

BhexU[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
BhexU[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
BhexU[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
BhexU[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])

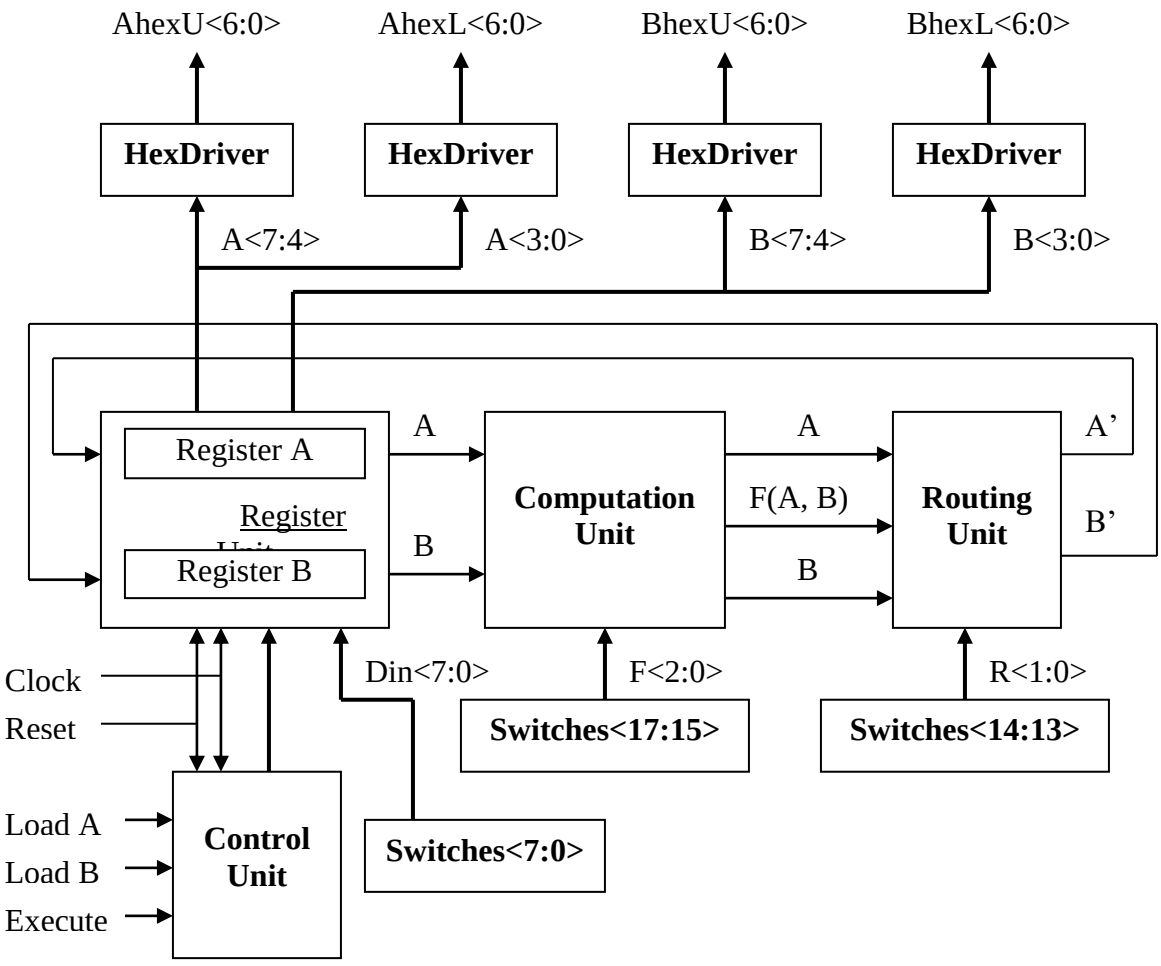


Figure 14: Block Diagram

Table 2. Function Table

Function Selection Inputs			Computation Unit Output
F<2>	F<1>	F<0>	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R<1>	R<0>	A’	B’
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

Design Resources and Statistics

Altera FPGAs use the term Logic Element (LE) to describe a functional block that contains 1 look-up table (LUT), 1 register (FF), and some additional circuitry. In FPGAs, all the combinational logic is implemented in LUTs. To find out the usage of LUTs after compilation, go to the ***Compilation Report*** tab. On the left, go to ***Fitter > Resource Section > Resource Usage Summary***. The summary should look like what is shown in Figure 1.

Fitter Resource Usage Summary

	Resource	Usage
1	◢ Total logic elements	2,661 / 33,216 (8 %)
1	-- Combinational with no register	898
2	-- Register only	432
3	-- Combinational with a register	1331
2		
3	◢ Logic element usage by number of LUT inputs	
1	-- 4 input functions	1128
2	-- 3 input functions	709
3	-- <=2 input functions	392
4	-- Register only	432
4		
5	◢ Logic elements by mode	
1	-- normal mode	2069
2	-- arithmetic mode	160
6		
7	◢ Total registers*	1,831 / 34,593 (5 %)
1	-- Dedicated logic registers	1,763 / 33,216 (5 %)
2	-- I/O registers	68 / 1,377 (5 %)
8		
9	Total LABs: partially or completely used	198 / 2,076 (10 %)
10	Virtual pins	0
11	◢ I/O pins	47 / 475 (10 %)
1	-- Clock pins	1 / 8 (13 %)
12		
13	Global signals	8
14	M4Ks	14 / 105 (13 %)
15	Total block memory bits	44,032 / 483,840 (9 %)
16	Total block memory implementation bits	64,512 / 483,840 (13 %)
17	Embedded Multiplier 9-bit elements	0 / 70 (0 %)
18	PLLs	1 / 4 (25 %)
19	Global clocks	8 / 16 (50 %)
20	JTAGs	1 / 1 (100 %)
21	ASMI blocks	0 / 1 (0 %)

Figure 1

Here, the total number of LEs is reported. LEs are divided into 3 categories: combinational with no register, register only, and combination with register. Add the numbers of *combinational with no register* and *combinational with register* to get the total number of LUTs used. Next, the numbers of LEs categorized by the number of LUT inputs are reported. Sum up the numbers of *4 input function*, *3 input funtions*, and *<= 2 input functions*. Note that this number is equal to the number of LUTs we just computed. Further below, we can find a report of total number of registers (flip-flips). Note that the total number of dedicated logic registers is equal to the numbers of LEs that are *combinational with register* and *registers only*. Further below, we can find a report of total block memory implementation bits. There is an on-chip RAM block on the Cyclone II FPGA, sizing 483,840 bits, which we call block memory (BRAM). BRAM is not used in most of the

SystemVerilog labs, but in the SoC lab, it is required for to store the software for the Nios II processor. The reported number gives you the usage of BRAM in the design.

If the design utilizes Altera's built-in digital signal processing (DSP) blocks, a separate report can be found at ***Fitter*** -> ***Resource Section*** -> ***DSP Block Usage Summary Report***. If no such report is there, it means that no DSP blocks are used.

For the timing analysis, the TimeQuest Timing Analyzer is used in Quartus Prime. To perform a detailed analysis on the operating frequency, input and output timing constraints of the circuit needs to be provided using the Synopsys Design Constraints (.sdc) format. These constraints usually arise when the circuit designed for the FPGA is not a standalone circuit, but rather it is merely a part of a larger system. The other parts of the system often have their own operating constraints, which eventually become the input and output constraints for the FPGA circuit. However, since the FPGA designs in our labs are standalone systems with manual inputs, it is not easy to come up with a SDC file with detailed timing constraints. However, simple timing analysis can still be performed by simply defining the system clock in the SDC file. This is because that the maximum possible operating frequency of the design can still be deduced from the design itself disregard of any information from the inputs and outputs. To define the system clock in the SDC file, simply save the following script in a .sdc file for your project:

```
#**************************************************************
# Create Clock (where 'Clk' is the user-defined system clock name)
#**************************************************************
create_clock -name {Clk} -period 20ns -waveform {0.000 5.000} [get_ports {Clk}]

# Constrain the input I/O path
set_input_delay -clock {Clk} -max 3 [all_inputs]
set_input_delay -clock {Clk} -min 2 [all_inputs]

# Constrain the output I/O path
set_output_delay -clock {Clk} 2 [all_outputs]
```

The maximum allowed frequency for the system clock can be found under ***TimeQuest Timing Analyzer -> Fmax Summary***.

Note: It does not matter what name you give to the .sdc file, as long as it's declared as a .sdc file instead of a regular .sv file. With the inclusion of the .sdc file, the 'TimeQuest Timing Analyzer' section in the 'Compilation Report' after the full compilation will show constrained result for 'Slow 1200mV 85C Model', 'Slow 1200mV 0C Model', and 'Fast 1200mV 0C Model'. You can pick the

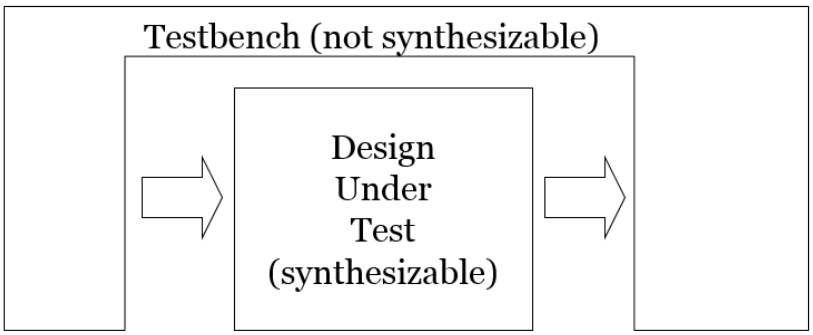
frequency from the 'Slow 1200mV 85C Model', since high temperature makes the chips slower, and 85C is the upper operating temperature limit for commercial-grade electrical devices. If you do not include the .sdc file in your project, TimeQuest Timing Analyzer will still give you some unconstrained frequencies in the same report (the three subsections will be shown in red instead of black). Notice that the frequency now is much higher. This is because in the .sdc constraint file we use the 50MHz clock, but without the file Quartus defaults the clock to be 1000MHz for all inputs. You can see the clock information under 'Clocks' subsection in the 'TimeQuest Timing Analyzer' report.

For the Power consumption analysis, the PowerPlay Power Analyzer is used in Quartus Prime. To activate the PowerPlay Power Analyzer during project compilation, go to ***Assignments*** -> ***Settings***, and find the ***PowerPlay Power Analyzer Settings*** under ***Category***. Check the box labeled ***Run PowerPlay Power Analyzer during compilation***. The power analysis report can be found in ***PowerPlay Power Analyzer*** -> ***Summary*** in the ***Compilation Report*** tab.

Instructions for Testbench for the Bit-Serial Logic Processor

These instructions are a tutorial for creating a testbench that generates simulation waveforms and results for the bit-serial processor in the Quartus Prime environment, with the use of the ModelSim-Altera simulator. It is **required in the demo**.

A testbench envelopes the design under test (DUT), provides it with test stimuli, collects the results, and analyzes it, as illustrated below. In the bit-serial processor, the DUT is the processor module. We will use the testbench to provide some inputs for testing. The results can be viewed as waveforms in ModelSim-Altera.



Note that we have so far emphasized on the synthesizability of SystemVerilog coding because the code in concern describes the design itself. On the other hand, a testbench is used in simulations and is not required to be synthesizable on hardware. Therefore, some syntax used in the testbench is not synthesizable. Pay attention to the differences and avoid using syntax that are not synthesizable in the designs!

Preliminary Work:
Build the bit-serial processor project and include all the provided code. For the "Processor (Main Module)" part, either code it or use the schematic editor to complete it as taught in the previous section. Make sure the top-level "Processor" module uses all the port names given in the lab manual because they will be used in the testbench.

Writing the Testbench:
The following is the contents of ***testbench_8.sv*** in the provided code:

```
module testbench();

timeunit 10ns;   // Half clock cycle at 50 MHz
                 // This is the amount of time represented by #1
timeprecision 1ns;

// These signals are internal because the processor will be
// instantiated as a submodule in testbench.
logic Clk = 0;
logic Reset, LoadA, LoadB, Execute;
logic [7:0] Din;
```

```
logic [2:0] F;
logic [1:0] R;
logic [3:0] LED;
logic [7:0] Aval,
            Bval;
logic [6:0] AhexL,
            AhexU,
            BhexL,
            BhexU;

// To store expected results
logic [7:0] ans_1a, ans_2b;

// A counter to count the instances where simulation results
// do no match with expected results
integer ErrorCnt = 0;

// Instantiating the DUT
// Make sure the module and signal names match with those in your design
processor processor0(.*);

// Toggle the clock
// #1 means wait for a delay of 1 timeunit
always begin : CLOCK_GENERATION
    #1 Clk = ~Clk;
end

initial begin: CLOCK_INITIALIZATION
 Clk = 0;
end

// Testing begins here
// The initial block is not synthesizable
// Everything happens sequentially inside an initial block
// as in a software program
initial begin: TEST_VECTORS
Reset = 0;       // Toggle Rest
LoadA = 1;
LoadB = 1;
Execute = 1;
Din = 8'h33;     // Specify Din, F, and R
F = 3'b010;
R = 2'b10;

#2 Reset = 1;

#2 LoadA = 0;    // Toggle LoadA
#2 LoadA = 1;

#2 LoadB = 0;    // Toggle LoadB
 Din = 8'h55;    // Change Din
#2 LoadB = 1;
```

```
 Din = 8'h00;    // Change Din again

#2 Execute = 0;  // Toggle Execute

#22 Execute = 1;
 ans_1a = (8'h33 ^ 8'h55); // Expected result of 1st cycle
 // Aval is expected to be 8'h33 XOR 8'h55
 // Bval is expected to be the original 8'h55
 if (Aval != ans_1a)
       ErrorCnt++;
 if (Bval != 8'h55)
       ErrorCnt++;
 F = 3'b110;     // Change F and R
 R = 2'b01;

#2 Execute = 0;  // Toggle Execute
#2 Execute = 1;

#22 Execute = 0;
 // Aval is expected to stay the same
 // Bval is expected to be the answer of 1st cycle XNOR 8'h55
 if (Aval != ans_1a)
       ErrorCnt++;
 ans_2b = ~(ans_1a ^ 8'h55); // Expected result of 2nd cycle
 if (Bval != ans_2b)
       ErrorCnt++;
 R = 2'b11;
#2 Execute = 1;

// Aval and Bval are expected to swap
#22 if (Aval != ans_2b)
       ErrorCnt++;
 if (Bval != ans_1a)
       ErrorCnt++;

if (ErrorCnt == 0)
 $display("Success!"); // Command line output in ModelSim
else
    $display("%d error(s) detected. Try again!", ErrorCnt);

end

endmodule
```

Configurations:

Go to the menu ***Assignments > Settings***. Click on ***EDA Tool Settings*** on the left. Make sure the simulation tool is set to *ModelSim-Altera*, and the format is *SystemVerilog HDL*. Figure 1 shows the dialog box. If the project was created according to the lab manual, these settings should already be there.

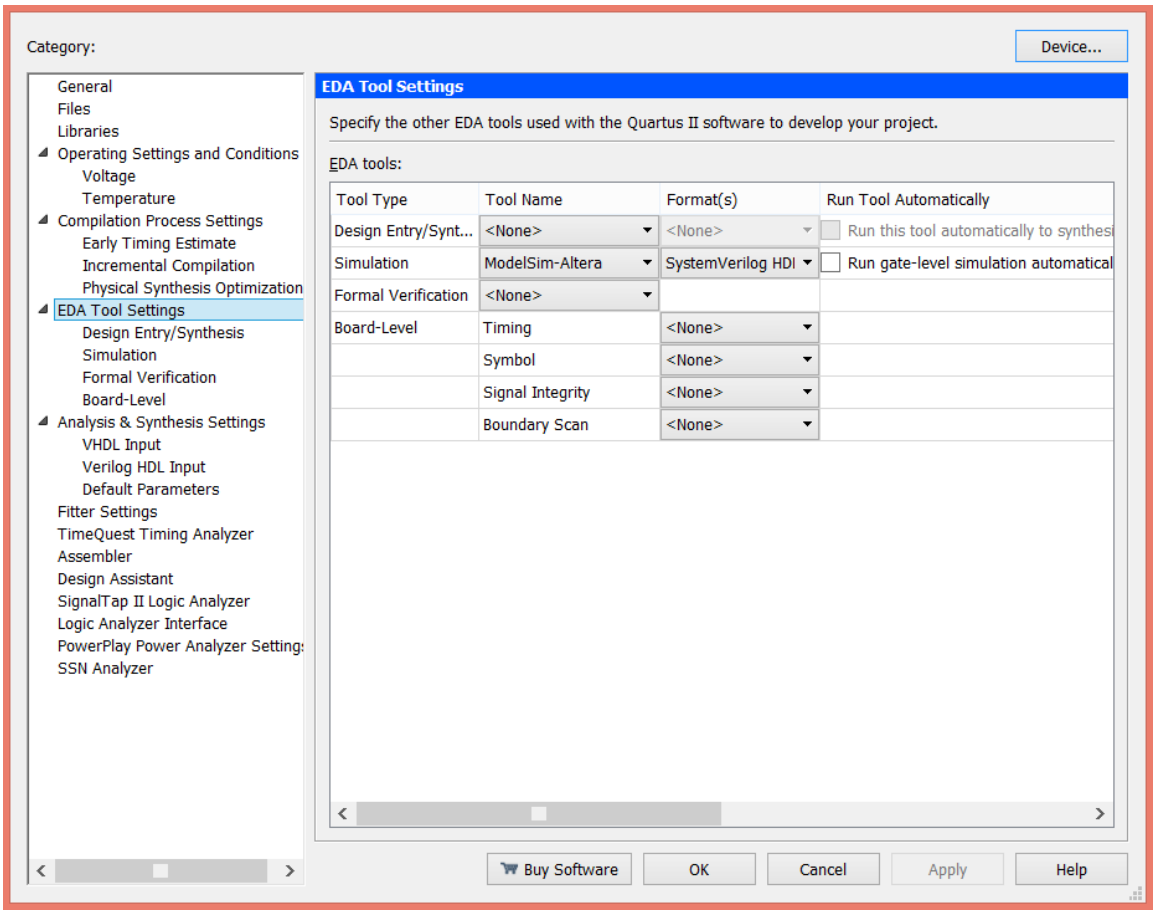


Figure 1

Next, click on ***Simulation*** under ***EDA Tool Settings***. Select *ModelSim-Altera* for *Tool name*, *SystemVerilog HDL* for *Format of output netlist*, and "*simulation/modelsim*" for *Output directory*. Figure 2 shows the dialog box.

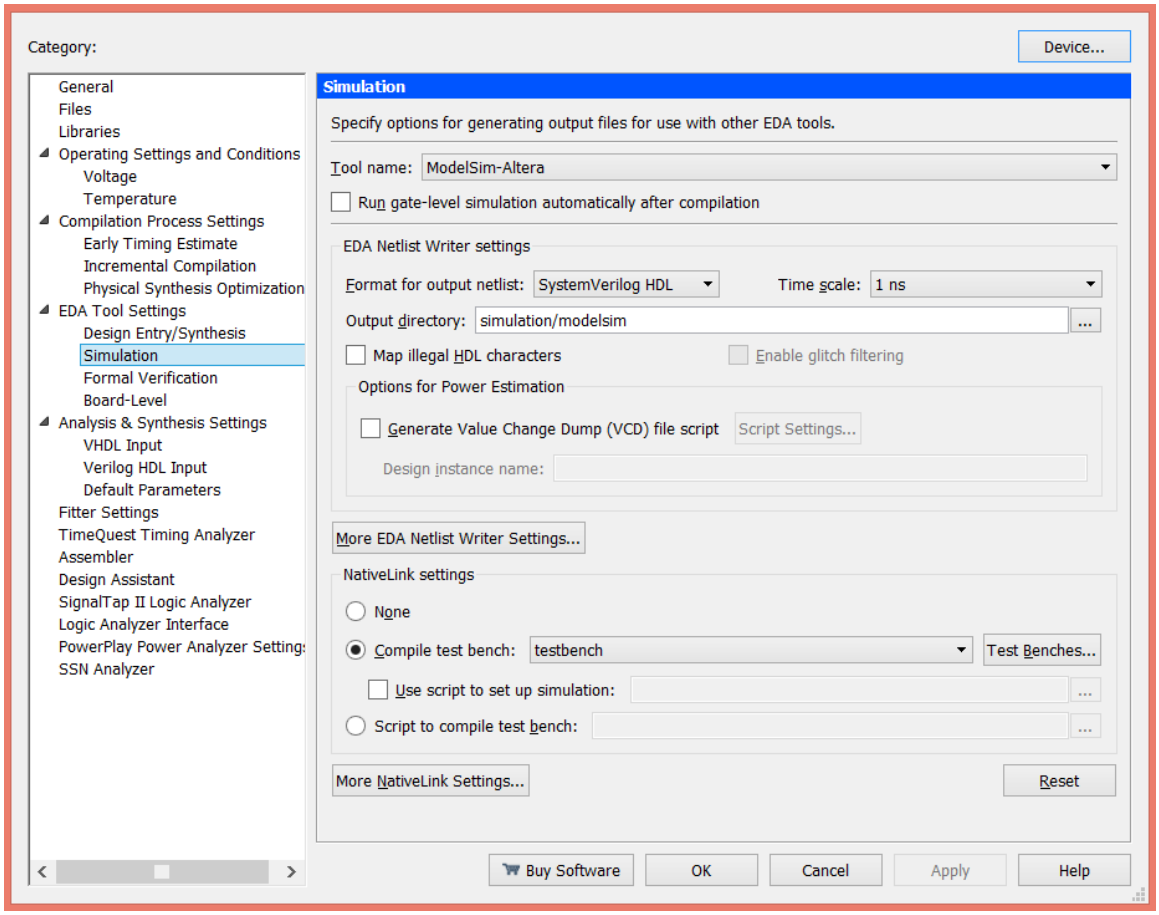


Figure 2

Next, click on the ***Test Benches…*** button in ***NativeLink settings***, on the bottom of the dialog box shown in Figure 2. A window should pop up. Select ***New…***, and a window shown in Figure 3 should pop up. Enter *testbench* for *Test bench name* and *Top level module in test bench*, *1000 ns* for *simulation end time*, and then click on the **…** button and add *testbench.sv* into *test bench and simulation files*.

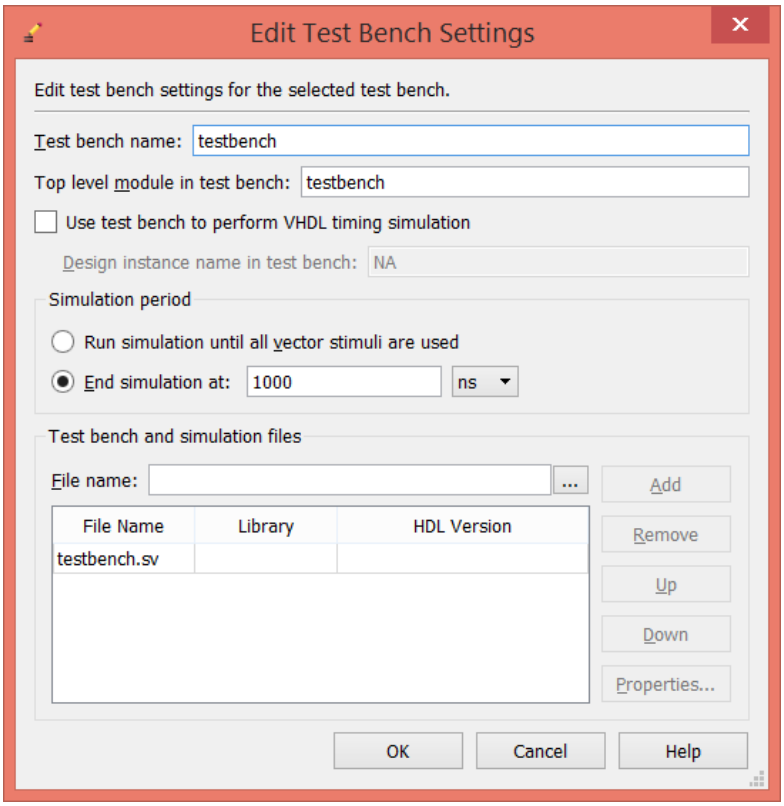


Figure 3

Now we're ready to run the testbench. Go back to the main Quartus Prime window, compile the project by clicking ***Start Analysis & Synthesis***, and then select ***Tools > Run Simulation Tool > RTL simulation*** or simply click on the *RTL Simulation button* ( ). The results will be shown in ModelSim-Altera as in Figure 4. You may need to right click on the waveform and zoom to a proper scale for better reading. Press 'f' to see the entire waveform. Use Ctrl+Scroll Wheel to zoom in/out the waveform.

Note: By default the buses are displayed in binary value. To change the display format, right click on the bus name and select ***Radix -> <format>***. For example, if you wish to see the decimal value, you should select ***Radix -> Decimal***.

To run a functional simulation of your design, click the ***Run-All*** button ( ) in the toolbar. If you wish to modify any portion of the waveform and re-simulate, you should first click the ***Restart*** button ( ) to erase the existing outputs.

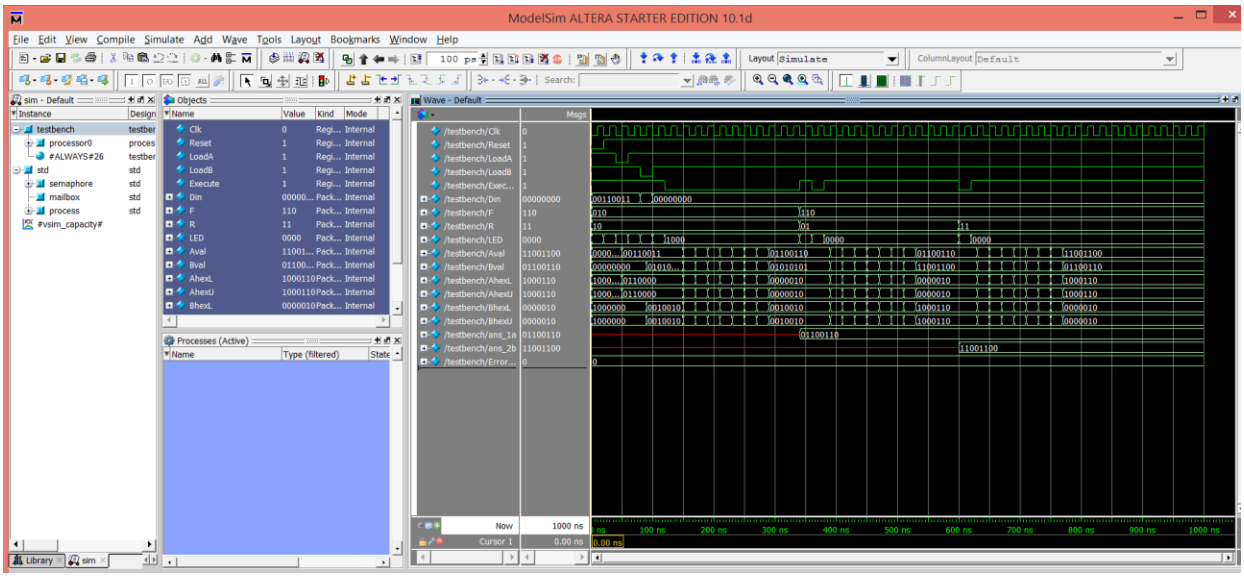


Figure 4

Note that the red lines indicating that ans_1a and ans_2b are unknown values in the beginning are expected because these two variables would not be computed until the middle of the TEST_VECTORS initial block.

Monitoring Internal Signals:

In ModelSim, we also have the ability to monitor internal signals in the hierarchical design in addition to the ones specified in the testbench. To do this, navigate the design hierarchy on the left, select the signal to be observed, and then right click and choose ***Add Wave***. The selected signal will be added into the simulation waveform window. Figure 5 shows an example of adding the state variable in the controller.

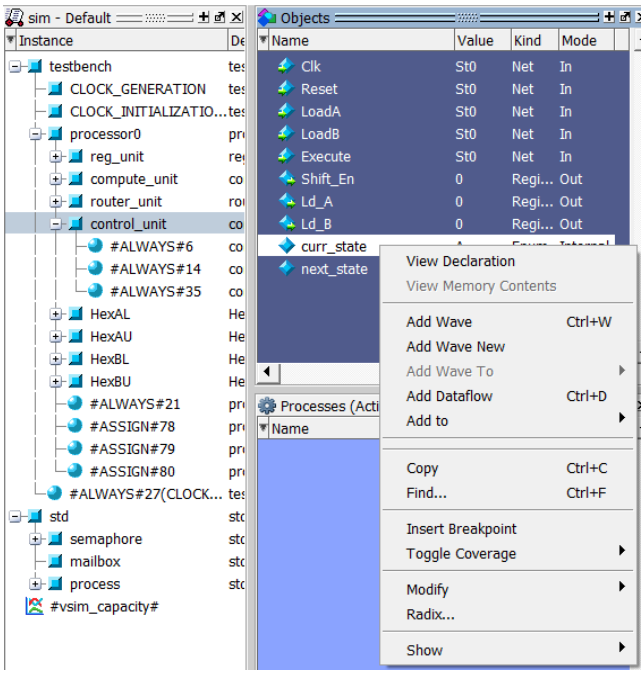


Figure 5

After all signals of your choice are added, go to the command line below and type in:

```
restart –f
```

This will reset the waveform window such that the testbench starts from the beginning again.

```
log –r *
```

This will tell ModelSim to record all signals in the circuit recursively.

```
run 1000ns
```

This will run the testbench for 1000ns. You may change the simulation time if needed.
The waveforms of the selected internal signals should be available for view now.

Troubleshooting:
If a certain part of your design is not showing up in the design hierarchy window, it's usually because the design file fails to compile successfully in ModelSim. Look into the error messages in the *Transcript* to find out what went wrong.

ModelSim has its own internal compiler that's different from Quartus Prime's. The accepted syntax and the ways they handle certain statements are slightly different. Therefore, the compiled circuits may be slightly different as well, especially if a good coding style is not employed. Also, ideal circuits with no delays are assumed in simulations, whereas delays do exist in real world. If timing is not handled safely, the values read in a specific clock cycle may be one cycle off. These two reasons may lead to different results in simulation from those on-board.

If you have output signals computed in *always_ff* blocks, declare them as *output logic* instead of *output*. The ModelSim compiler tends to complain about this.

(Optional) Instructions for Schematic Design Entry for the Bit-Serial Logic Processor

These instructions are a tutorial for creating your top-level entity as a schematic diagram, rather than as a SystemVerilog module. Quartus II can generate Verilog code automatically based on your schematic. (SystemVerilog is not yet supported in the current version.) This is useful for small designs, but discouraged for larger designs as text designs are more maintainable and legible.

These instructions are optional; if you would rather enter your port maps in text form, you are free to do so. Note that the schematic design introduced here isn't directly compatible with the Testbench simulation described in **IQT. 33-40**. Conversion to Verilog is needed (you may follow the steps provided in **IQT. 28** for conversion between schematic entry and HDL code, or simply use the provided 'processor.sv' module. However, we recommend reading through these instructions, since this procedure can save you time and frustration on the later labs and on the final project.

Preliminary Work:
Build the bit-serial processor project and include all the provided code, **except** for the "Processor (Main Module)" code.

Add New Source – Schematic:
In the ***File*** menu, select ***New…***. Select ***Block Diagram/Schematic File*** in the window that pops up and click ***OK***. A blank schematic will be opened for editing.

Prepare Schematic Symbols:
In the ***Project Navigator*** window, click the ***Files*** tab. Expand the ***Device Design Files*** folder if it isn't already expanded; this will show all the design files currently in the project. Open the module from the Project Navigator, then click ***File -> Create/Update -> Create Symbol Files for Current File*** (See Figure 1). This creates a symbol for the Register Unit which you will be able to place within your schematic. Repeat this sequence for all the other SystemVerilog modules in the project.

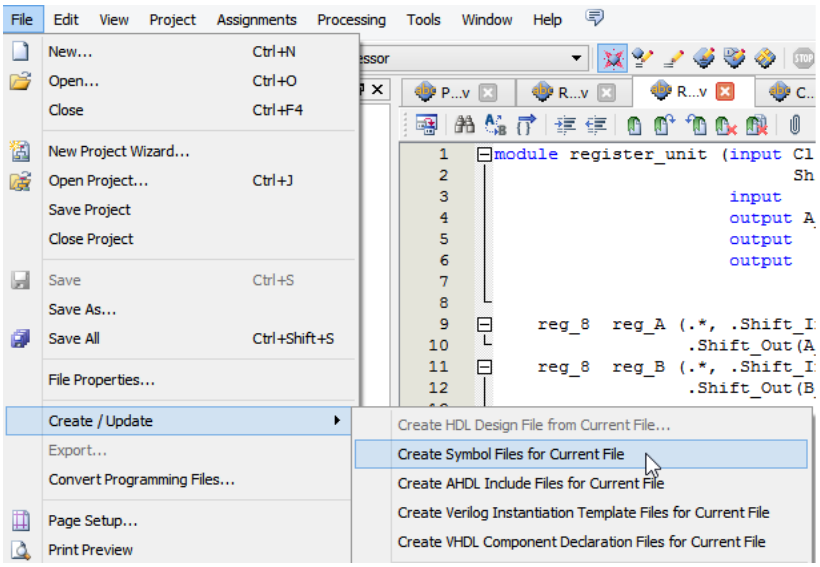


Figure 1

Add Symbols to Schematic:

If your blank schematic is not currently displayed, bring it to the top by clicking on its tab. Click the ***Symbol Tool*** () button. In the ***Libraries*** selection box, expand the ***Project*** folder. You should see a list of all the entities for which you have created a symbol. Click on the Register Unit, uncheck ***Repeat-insert mode***, and click ***OK*** (see Figure 2). Move the mouse out over the schematic. A preview of the symbol will follow the cursor. Click to lay down an instance of the Register Unit. Repeat this for each unit in the top-level entity. For the HexDrivers, leave ***Repeat-insert mode*** checked to allow you to place multiple instances of the unit at a time; hit escape to exit repeat-insert mode when you're done. Save your design now, naming it ***Processor***.

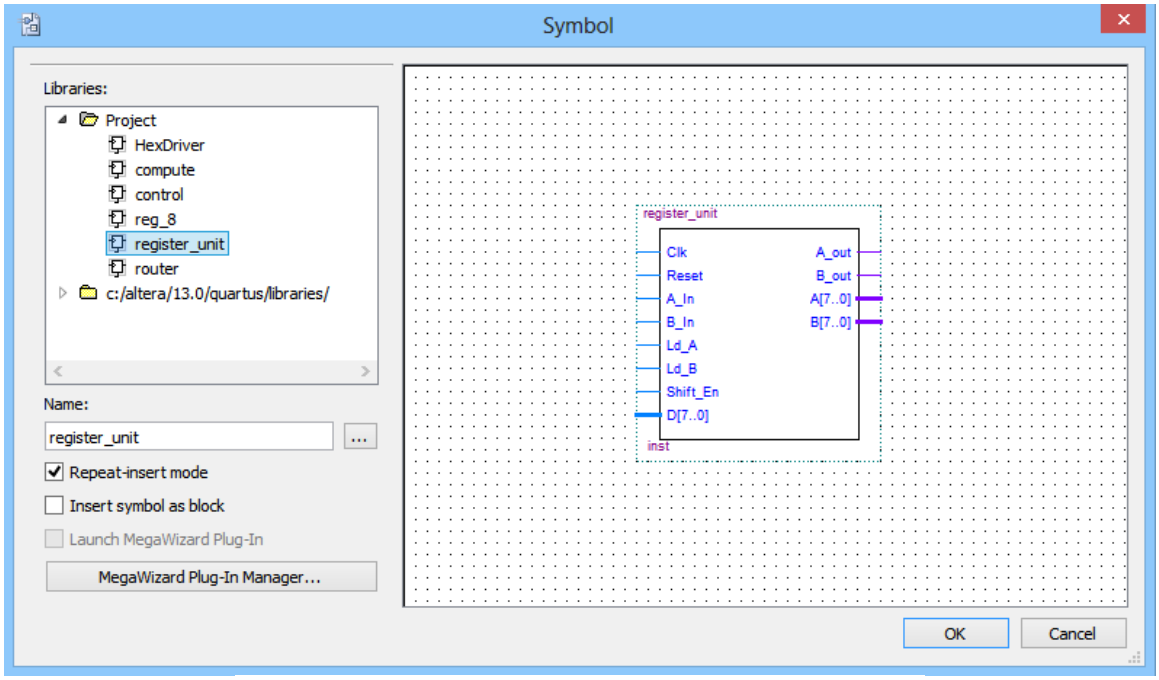


Figure 2

Navigating & Altering Your Design:

Click the ***Zoom Tool*** () button. Left-clicking with the Zoom Tool will zoom in on the spot you clicked; right clicking will zoom out. You can also draw a box with the tool to zoom the window on the box. Draw a box around one of your instanced entities to take a closer look at the symbol. By default, inputs are arranged on the left, with outputs on the right. Note the visual distinction between single-bit signals and multi-bit buses. Zoom back out. If you are unhappy with the placement of any of your symbols, you can click and drag it to another location with the *Selection Tool* (). The symbol will automatically snap to the grid defined in the schematic. You should lay out the register unit, compute unit, and router units so that they form a left-to-right data path, as in the block diagram in the lab manual.

Editing Symbols:

The visual layout of the default symbols is tolerable, but in complex designs, it is often inconvenient to have all inputs on the left side of a block and all outputs on the right. With the selection tool, right-click on the register unit symbol and select ***Edit Selected Symbol***. A view of the register unit's symbol will open. With the selection tool, you can drag and drop pins around, resize the drawn box that forms the body of the symbol, and resize the overall dimensions of the symbol by dragging the

edges of the outer hashed box. Rearrange the symbol to better resemble the configuration in the lab manual's block diagram: put the A and B output buses on the top, the Ld_A, Ld_B, and Shift_En pins on the bottom, and move the Clk and Reset pins on the lower left instead of the upper left. Save the view and close the window to return to the top-level schematic view. Notice that the symbol hasn't changed; we need to force it to update. Right-click on the symbol and select *Update Symbol or Block*, and click *OK* in the box that comes up. The old symbol will be replaced by your edited version. Repeat this for the other symbols in the diagram: move the control signals for the function and routing units to the bottom, the control signal outputs of the control unit to the top, and shorten the HexDriver symbol vertically to save some space.

Other notes on symbol editing:

- To shorten a symbol vertically, you may have to move the "inst" text box up. Be aware that everything must fit inside of the dashed symbol boundary.
- Pins will always snap to the outer perimeter of the symbol. To move a pin to the top or bottom of the symbol, it can help to move the mouse to just below the outer boundary.
- Make sure to resize your symbols so that all the pin labels are readable.
- When laying out pins for closely-linked symbols, you can save yourself some time by putting the adjoining pins in the same configuration in both symbols. The utility of this will be demonstrated in the next section.

Wire it Up:

If you've set up the pins on adjoining units as stated above, you can quickly connect the pins by dragging one unit next to the other so that the pins overlap. Provided that the ***Use Rubberbanding*** option () is selected, you can pull the units apart again and the pins will remain connected; the program will automatically draw wires to keep them so. You should be able to do this with the register unit to the function unit, the function unit to the routing unit, and the control unit to the register unit.

Other connections aren't so easily made, however. Click the ***Orthogonal Node Tool*** () button. You can draw wires by clicking and dragging the cursor from one point to another. Connect the A_out and B_out ports of the router unit to the A_in and B_in ports of the register unit; you will have to do this in several stages to properly route the wires around the placed units. If you make a mistake, you can use the selection tool to correct it, by clicking on a wire segment to select it, and dragging it to the correct location. Save your design now; it should look something like Figure 3.

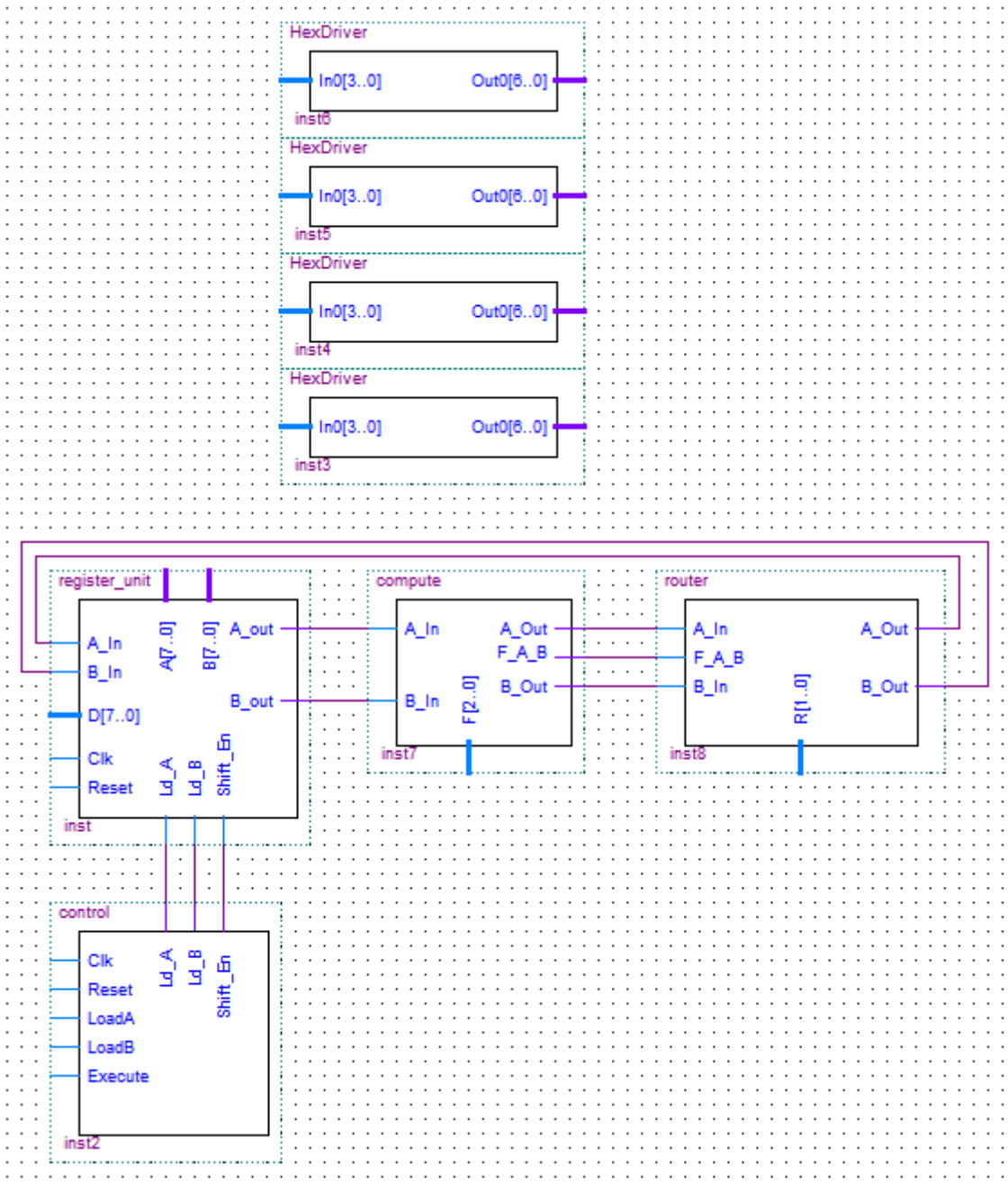


Figure 3

Ripping Buses:

You may have noticed that the outputs of the register unit that are to be displayed on the seven segment displays are eight bits wide, but the inputs to the HexDrivers are only four bits wide. We need to "rip" four-bit slices from the eight-bit signals so that they can be connected to the HexDrivers. First, use the ***Orthogonal Bus*** **Tool** () to draw a bus from the A[7..0] output of the register unit to the inputs of both of the upper HexDrivers. Similarly, connect the B[7..0] output to the other two HexDrivers. Notice that these wires are thicker than the ones you drew before; this indicates that these are multi-bit buses.

In order to rip the slices we want, we need to give these buses recognizable names. Right-click on the wire segment attached to the A output and select ***Properties***. Enter "A[7..0]" in the ***Name*** field, and click *OK*. A text box will appear near the wire showing its name. Repeat this for bus B, using the name "B[7..0]". (The syntax [7..0] is Altera-specific, and means the same thing as <7 downto 0> in SystemVerilog.)

To chose the slice that will be connected to the top HexDriver, right-click on the segment leading into it and select ***Properties***. Here, enter the name "A[7..4]". Repeat this for each of the other three HexDrivers, with the names "A[3..0]", "B[7..4]", and "B[3..0]", respectively. Names for all these signals will appear nearby, letting us visually confirm which bits are going where. You can use the selection tool to click and drag these labels to another location if other things get in the way of reading them. Overall, your HexDriver connections should look like Figure 4. Save your design again.

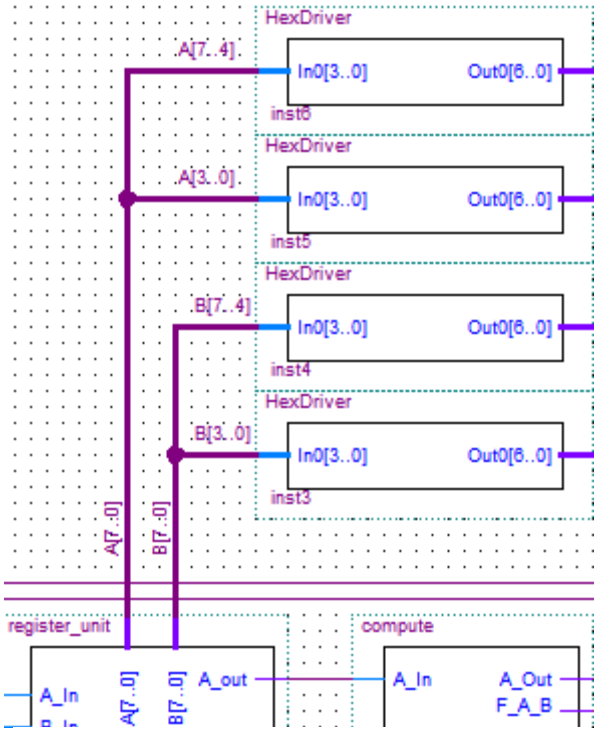


Figure 4

Add Input Pins:

We need to create 8 input pins and 7 output pins for our circuit. Click the symbol tool, and this time, expand the other folder in the ***Libraries*** window. (I will call this the ***Libraries*** folder, since it points to the location of the design libraries within the Quartus II install directory). Under that, expand ***Primitives***, and then ***Pin***. Select ***input***, and click *OK*. (See **Figure 5**.) Place an input pin in front of the D input of the register unit, one in front of each input to the control unit, and two pointing between the register and control units. Name these pins, from top to bottom, "D[7..0]", "F[2..0]", "R[1..0]", "Clk", "Reset", "Load_A", "Load_B", and "Execute" (or, for the last five, use whatever order appears on the control unit). You can name pins individually by double-clicking on them with the select tool to access the pin properties dialogue, or en-mass by first selecting the top pin with a single-click, and then double-clicking on the name text to edit it directly; when you press enter, it

will cycle forward to the next pin. Once this is done, connect the Clk, D, F, and R inputs to the correct pins on the register, function, routing, and control units. Hold off on the Reset, Load_A, Load_B, and Execute inputs for the time being.

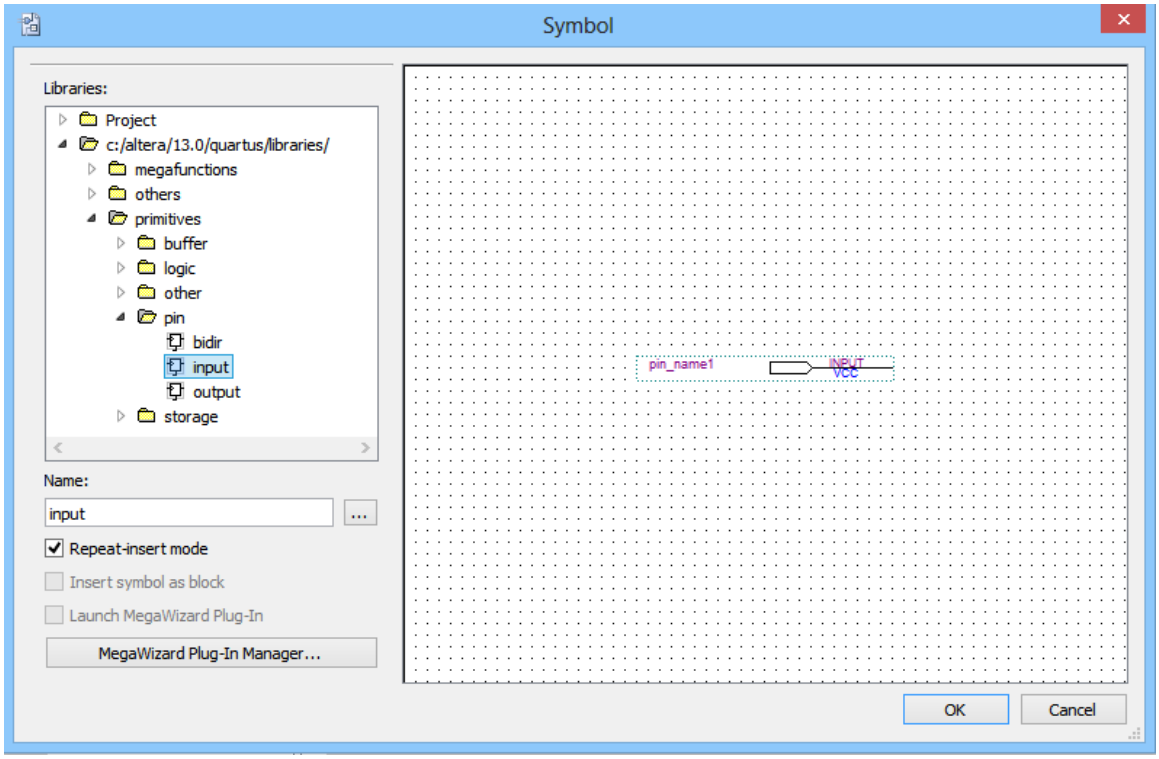


Figure 5

Add Inverters:

There's a subtlety regarding the pushbuttons on the DE2 board that is mentioned in the given code in the bit-serial processor for the top-level entity: the pushbuttons are active-low! All the given code assumes active-high logic, for consistency and simplicity. We therefore need to add inverters to the Reset, Load_A, Load_B, and Execute inputs. Click the symbol tool, and expand the ***Libraries*** folder, then ***Primitives***, and finally ***Logic***. Select ***not*** near the bottom of this list. An image of an inverter will appear. Click ***OK***, instance four of these near the four pushbutton input pins, connect these pins to the inverter inputs, and connect the inverter outputs to the appropriate ports on the register and control units.

Add Output Pins:

We're almost finished! For the outputs, we use a similar procedure. Use the symbols tool to place seven output pins – one next to each HexDriver, two beneath these (between the HexDrivers and the datapath), and one beneath the control unit. Name these "AhexU[6..0]", "AhexL[6..0]", "BhexU[6..0]", "BhexL[6..0]", "Aval[7..0]", "Bval[7..0]", and "LED[3..0]", from top to bottom. The hex outputs can be connected directly to the outputs of the HexDrivers; the Aval and Bval outputs should be connected to the A and B signals. For the LED output, we need to merge the Execute, Load_A, LoadB, and Reset signals. Draw a bus out from the LED output to the left, under the control unit and just under the input pins, then draw a wire from each of the four signals we need to this bus. Label the bus "LED[3..0]" and the wires individually, as before. The one connected to Execute should be "LED[3]", Load_A "LED[2]", Load_B "LED[1]", and Reset "LED[0]". At this point, your design should look something like **Figure 6**. Save your design now.

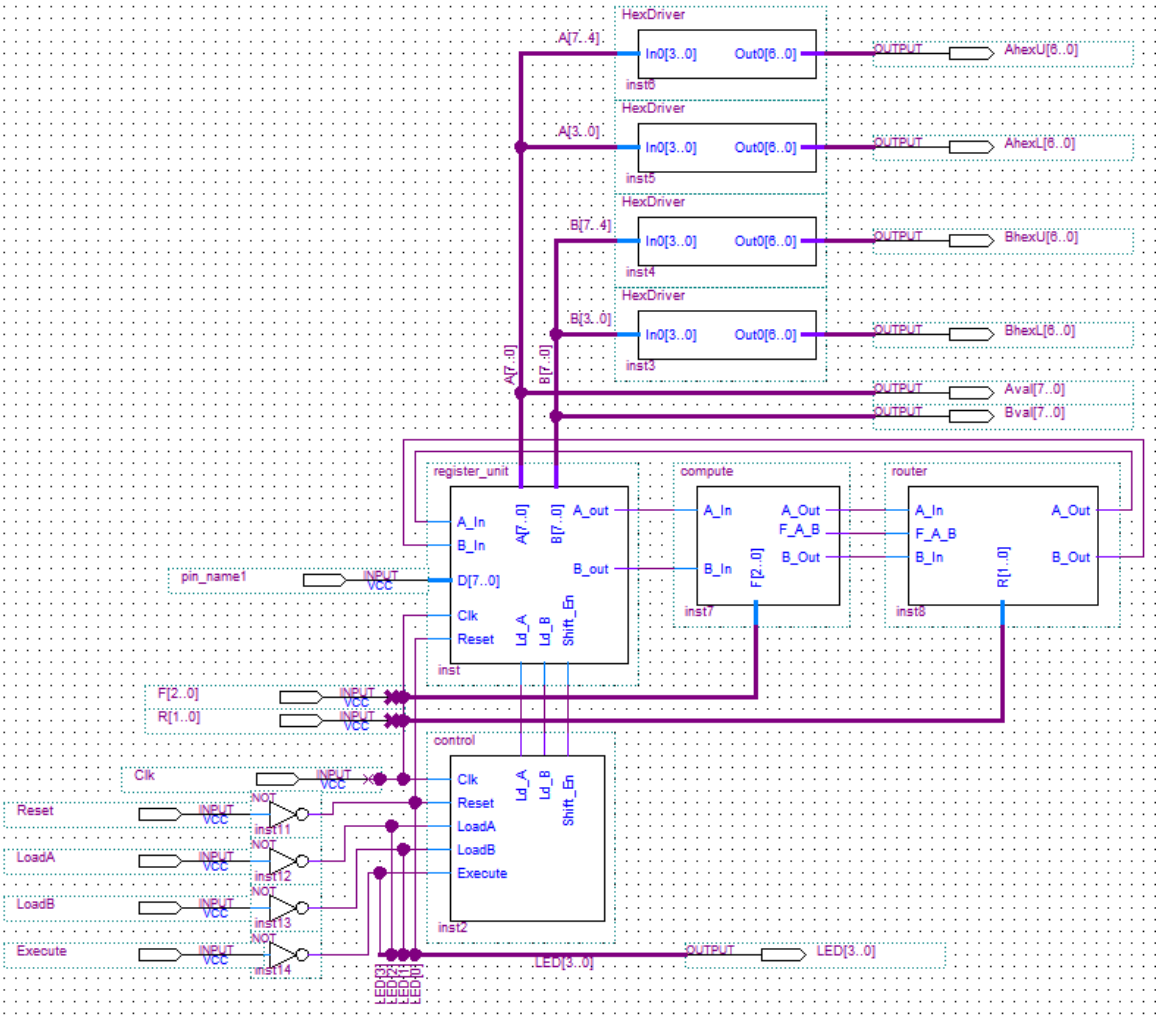


Figure 6

Checking Design Rules and Synthesizing:

Click the ***Start Analysis & Synthesis*** **button** (). If you've followed these instructions carefully, analysis & synthesis should complete to 100% with 0 warnings and errors. If you do get warnings or errors, view them in the appropriate tab and correct them. Quartus will build the hierarchy in the ***Project Navigator*** window.

Schematic to HDL Design File Conversion and vice versa:

Sometimes it is necessary to convert the schematic files into HDL design files for portability, since the schematic file format in Quartus is not supported in other compilers or simulators, not even for ModelSim-Altera. For other times, it is desirable to view the RTL synthesis schematics of the HDL design files to get a sense of what the codes are synthesized into. Quartus has the capability to perform the conversion automatically. The procedures are described below:

- **Schematic to HDL**
  Select the desired schematic file, and in the ***File*** menu, select ***Create/Update -> Create HDL design for current file***. In the window that opens, make sure ***Verilog HDL*** is selected, and click ***OK***. This will convert your schematic design into a Verilog file (currently conversion to SystemVerilog is not yet supported!) for you to hand in. Compare this code to the code for the Processor unit in the lab manual. You should find that they are functionally equivalent.

- **HDL to Schematic**
  Select the desired SystemVerilog file, and in the ***Tools*** menu, select ***Netlist Viewers -> RTL Viewer***. This will display the design file in synthesized schematic view. If the conversion HDL file is a top-level entity with instantiated modules, then the modules will be shown in RTL blocks. Double-clicking on a module block will bring you to the lower-level schematic of the module. You will be able repeat this until you have reached the lowest gate-level schematics. See [1] for detailed explanation.

The remaining steps are the same whether you've used these instructions or entered the port map directly. You should pick up from "Analyzing, Synthesizing, and Implementing the Design" in the Instructions for Performing Experiments Involving SystemVerilog in the lab manual, page IQT.12.

Final Notes:

If you've done your design well, it's probably high-enough quality to hand in as your block diagram in your lab report. Make sure any signals that need labels are labeled (by giving them names), and print it out. This not only saves you the trouble of entering the port map in text, but you don't have to draw a block diagram by hand either. You can add a title block if you like by using the symbol tool and looking in ***Libraries*** -> ***Primitives*** -> ***Other*** -> ***title.*** One point to note is that after you add pin assignments, these will appear in small tables next to each pin; you should move these tables so that they don't obscure any part of your circuit. Also note that block diagrams may be needed for subcomponents of your circuit. (For example, you might find it easier to group several block diagrams in a larger upper-level block diagram for readability and tidiness. In this case you will have to also provide the sub-block diagram to show a complete view of your circuit.) When you hand in your code, be sure to include the Verilog file you generated from your schematic.

References:

[1] "Analyzing Designs with quartus II Netlist viewers" in Quartus II handbook v13.0.0, Altera, 2012. Available at: http://www.altera.com/literature/hb/qts/qts_qii51013.pdf

INTRODUCTION TO
NIOS II and PLATFORM DESIGNER
(formerly QSYS)

ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to NIOS II and Platform Designer (formerly Qsys)

Abstract and Goals

The goal of this lab is creating a NIOS II based system on the Altera Cyclone IV device. The NIOS II is an IP based 32-bit CPU which can programmed using a high-level language (in this class, we'll be using C). A typical use case scenario is to have the NIOS II be the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic (designed using SystemVerilog) handles the high-performance operations.

The following will give you a walkthrough of the Platform Designer tool which is used to instantiate IP blocks (including the NIOS II). We will set up a minimal NIOS II device with an SDRAM (Synchronous Dynamic RAM) controller and a PIO (Parallel I/O) block to blink some LEDs using a C program running on the NIOS II to confirm it is working. You will then be asked to write a program which reads 8-bit numbers from the switches on the DE2 board and sums into an accumulator, displaying the output using the green LEDs via the NIOS II. This will involve instantiating another PIO block to read data from the switches and modifying the C program to input data, add, and display the data.

Goals (to demonstrate to your TA):
The TA will test the following functionality:
The green LEDs should always display the value of the accumulator in binary and the accumulator should be 0 on startup (all LEDs off). The accumulator should overflow at 255+1 to 0. (e.g. 255 + 1 → 0, 255 + 2 → 1, etc.)
Pressing the second to the left pushbutton at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off)
Pressing the left most pushbutton loads the number represented by the switches into the CPU, adding it to the accumulator. The 8 right-most switches are read as an 8-bit, unsigned, binary number with up being 1, down being 0.
Push buttons should only react once to a single actuation.

Be prepared to give answers to any of the italicized questions in this document from your TA when demoing. This is to ensure that you try to research what the settings do instead of simply trying to "make the picture look like your screen".

Hints:
Unit test the input and output. The output should already work, but make sure you can turn on and off every segment. If you have problems, check the schematic for the DE2, and make sure you are toggling the correct pins.
For this, and the rest of the class, you may use the C standard libraries (stdlib.h) or the C++ equivalents, this can save you a lot of work when coding in C.

Set up the System Combining the FPGA with the Nios II Processor:

Create a New Project:

- Start Quartus Prime.
- From the ***File*** menu select ***New Project Wizard***. Click *Next* to go through the intro screen, if it appears.
- The window in Figure 1 will appear. Fill in the fields from figure 1 (make sure there are no spaces in any of your entries). The program will ask you if it should create the specified directory if it does not exist; choose *yes*.
- Select ***Next*** on page 2 without adding any files.
- On page 3, select ***Cyclone IV E*** for the device family, make sure the second option under Target device is selected, and chose ***EP4CE115F29C7*** in the available devices list. See Figure 2.
  Click ***Next*** on page 4. Select *ModelSim-Altera* as the simulation tool name, and *SystemVerilog HDL* as the simulation format. See Figure 3. Click ***Finish*** on page 5.

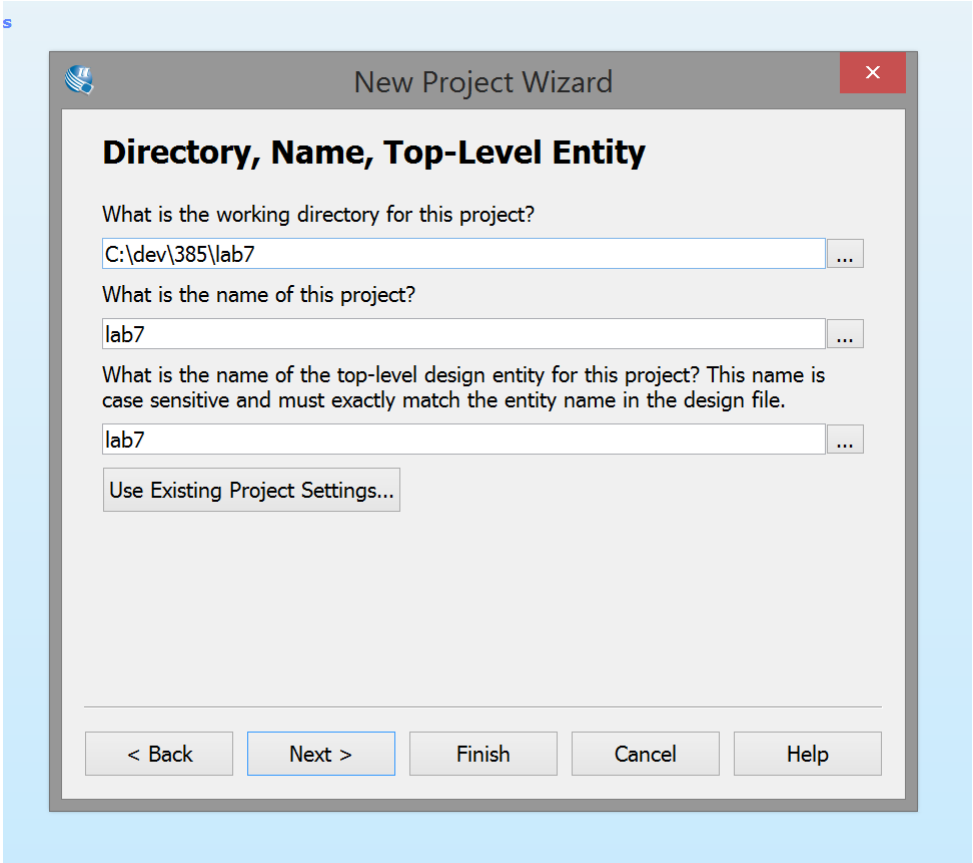


Figure 1

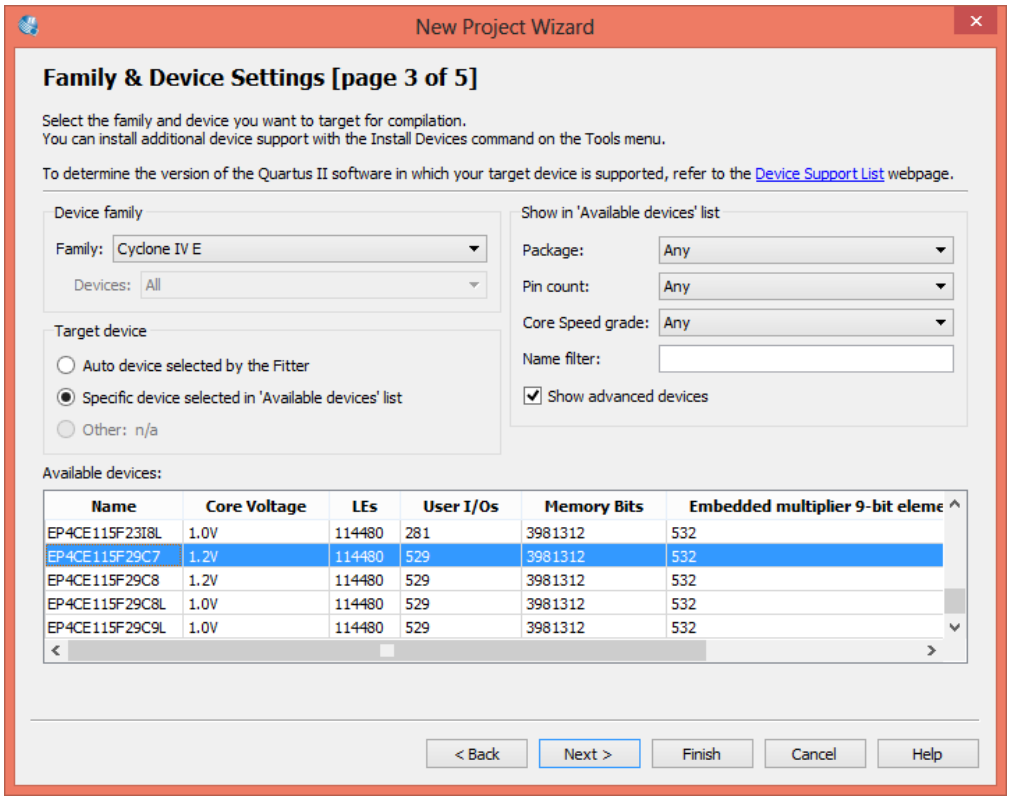


Figure 2

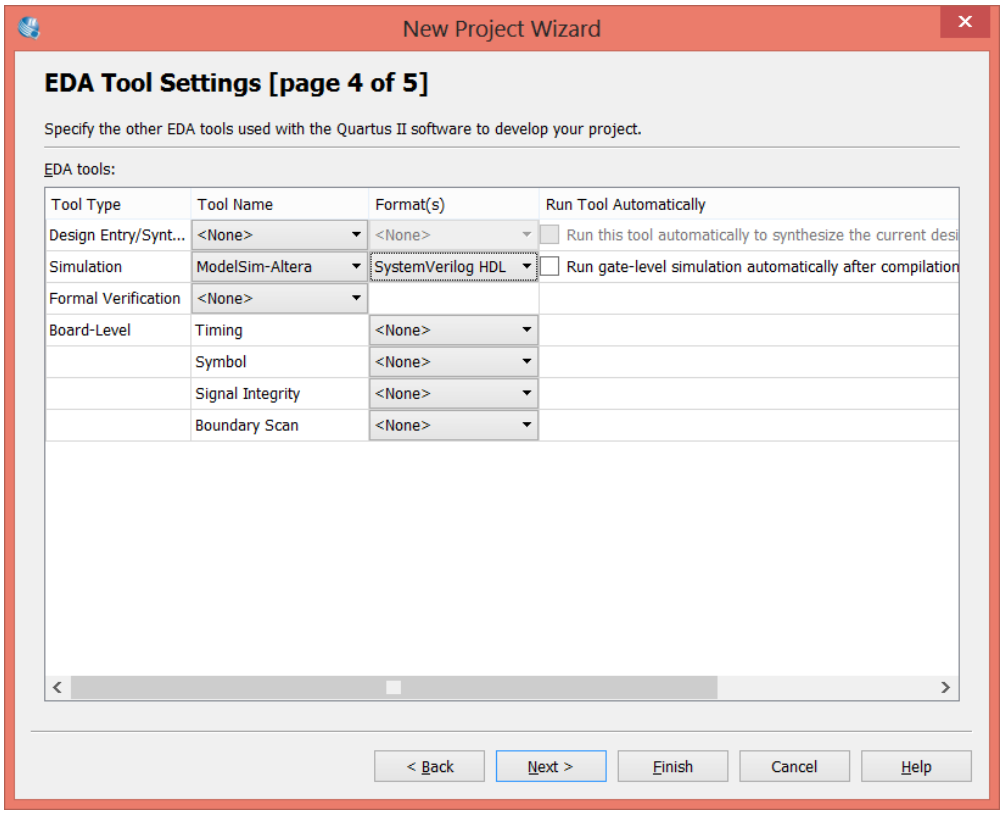


Figure 3

Create the SoC with Qsys:

Now we can set up the SoC with Platform Designer (formerly Qsys). In the ***Tools*** menu, select ***Platform Designer*** to launch Platform Designer. Immediately save the Platform Designer file as **lab7_soc.qsys**. This is the name of the hardware block which will contain the CPU and the supporting hardware (peripherals, memory, etc). A window shown in Figure 4 should pop up. Here, you can see a predefined clock signal. If needed, the clock frequency can be modified in the ***Clock Settings*** tab, as shown in Figure 5. In this lab, we will work with the default 50 MHz clock. If the *Clocks* tab is not available by default, then you can find it by selecting ***View*** > ***Clocks*** from the main menu.

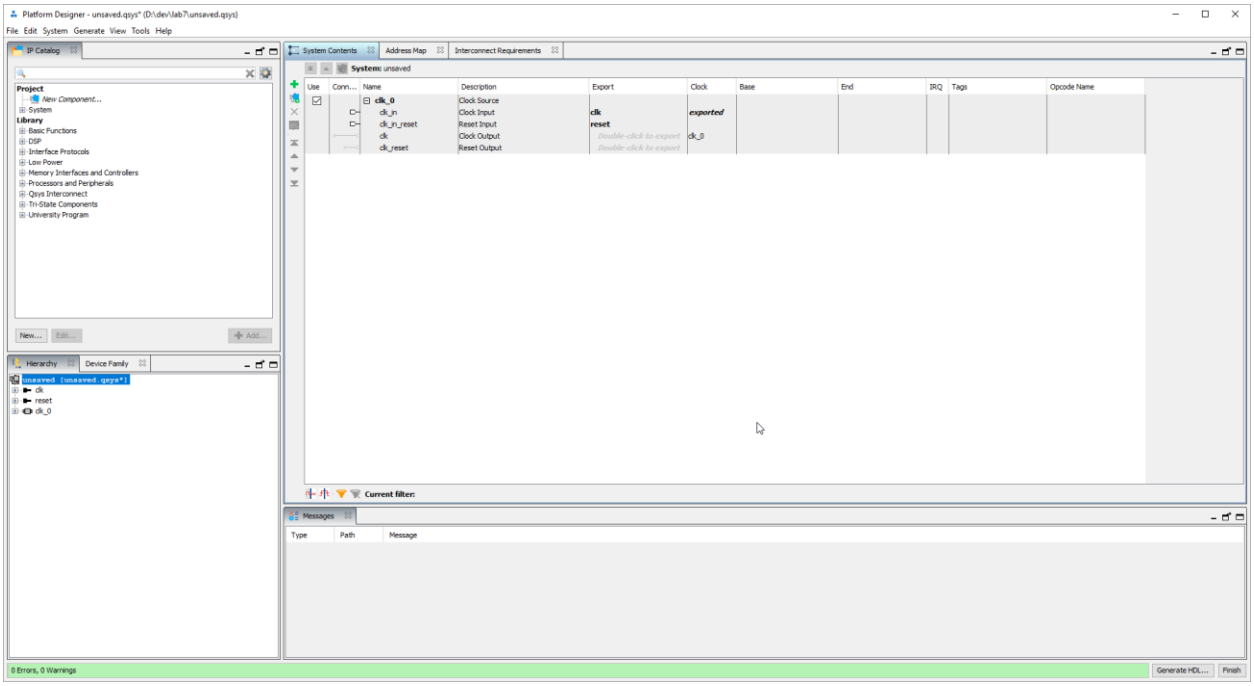


Figure 4

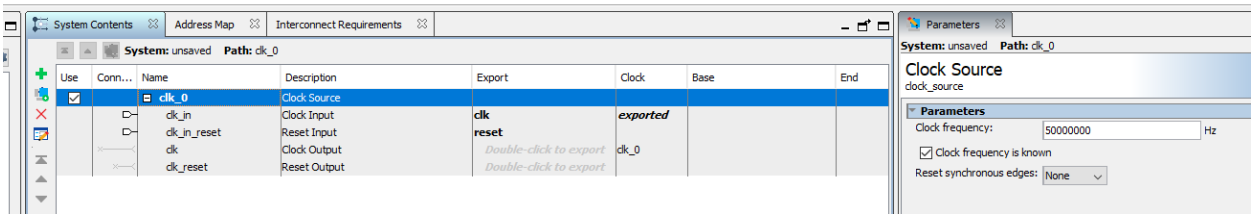


Figure 5

Next, specify the processor on the left side of the Qsys window by selecting ***Processors and Peripherals*** > ***Embedded Processors*** > ***Nios II Processor*** and clicking ***Add***. A window should pop up, as shown in Figure 6. In this window, select ***Nios II/e***, which is the economy version of the processor. Note that error messages regarding reset and exception vectors are shown on the bottom. This is because we haven't specified memory components in the system. Ignore the messages for now as we will provide the necessary information later. Click ***Finish***. Now we

have included the Nios II processor, as shown in Figure 7. *What are the differences between the Nios II/e and Nios II/f CPUs?*

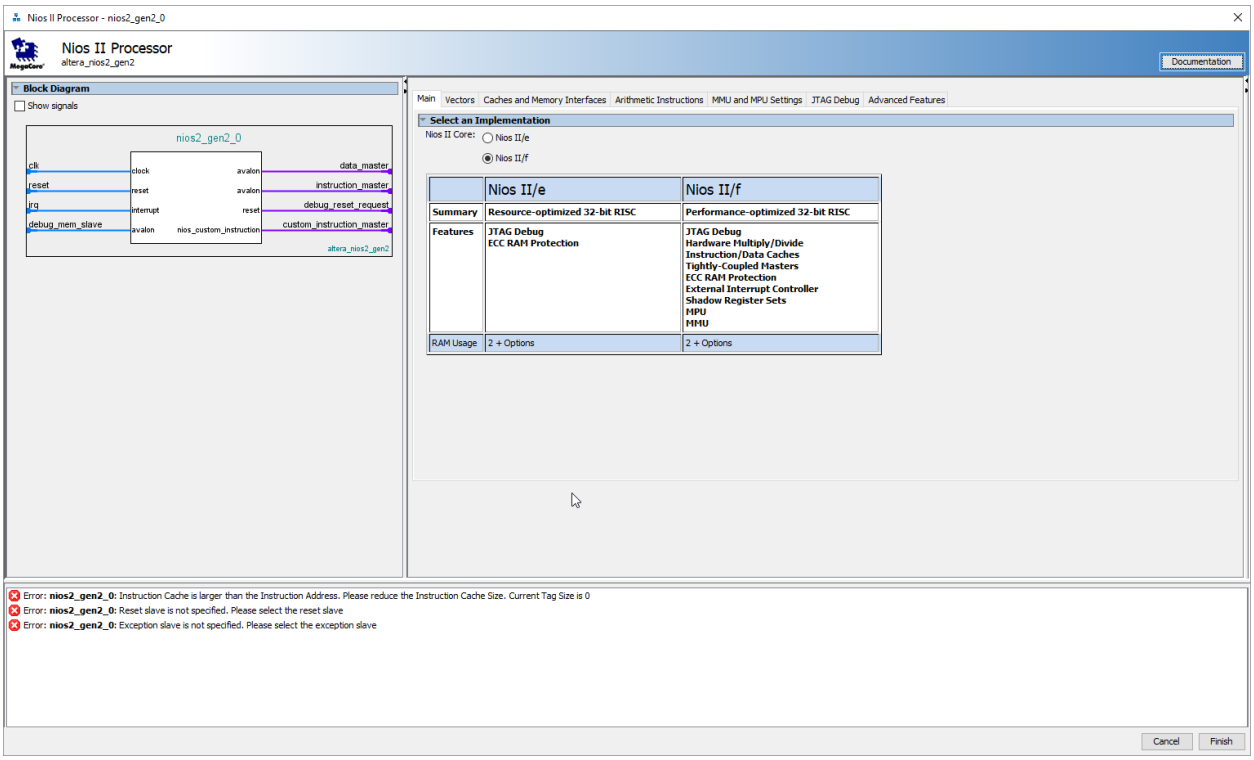


Figure 6

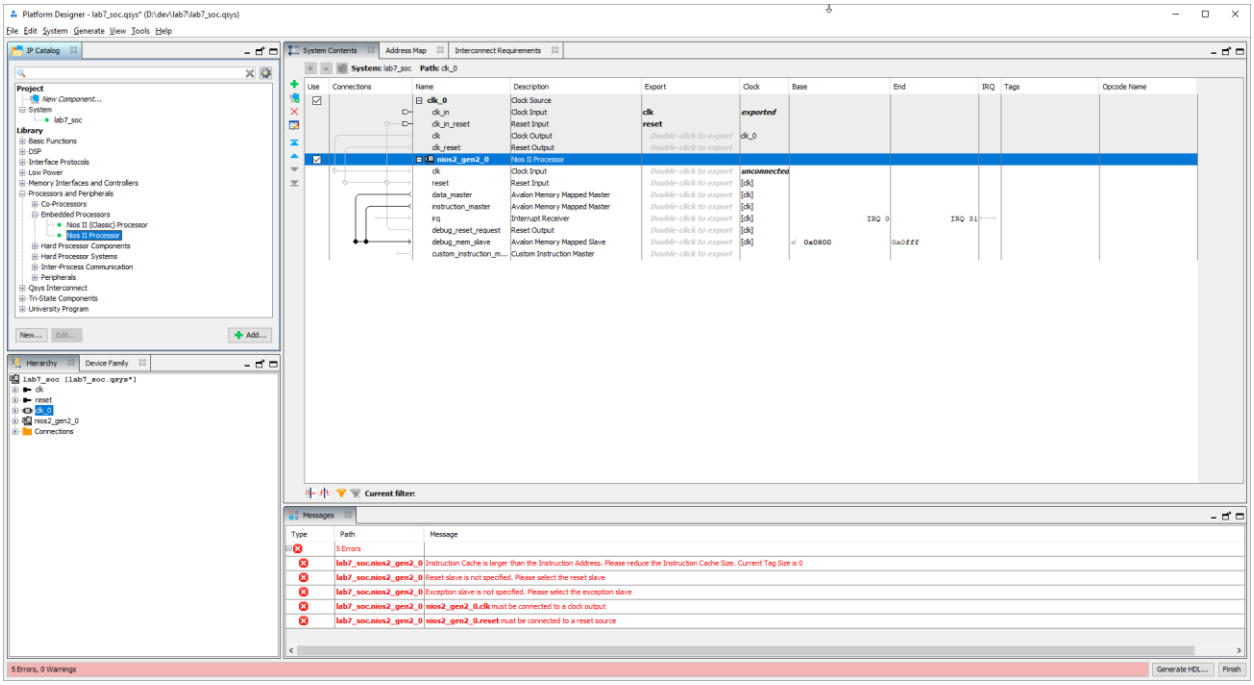


Figure 7

Next, we're going to instantiate an on-chip memory block. On the left side of the Qsys window, select ***Basic Functions > On Chip Memory > On-Chip Memory (RAM or ROM) Intel FPGA IP***, and click ***Add***. A window as shown in Figure 8 should pop up. Choose Memory Type to be ***RAM (Writable)*** and Total memory size to be 16 bytes. For most designs, we will save valuable on-chip memory and instead execute NIOS II programs from the DRAM, however we are instantiating a small on-chip RAM as a placeholder block (which you may use for your final project if you decide to use on-chip memory) *What advantage might on-chip memory have for program execution?* Note that you can get the datasheet for any IP block by clicking on documentation (top right). Click ***Finish***.

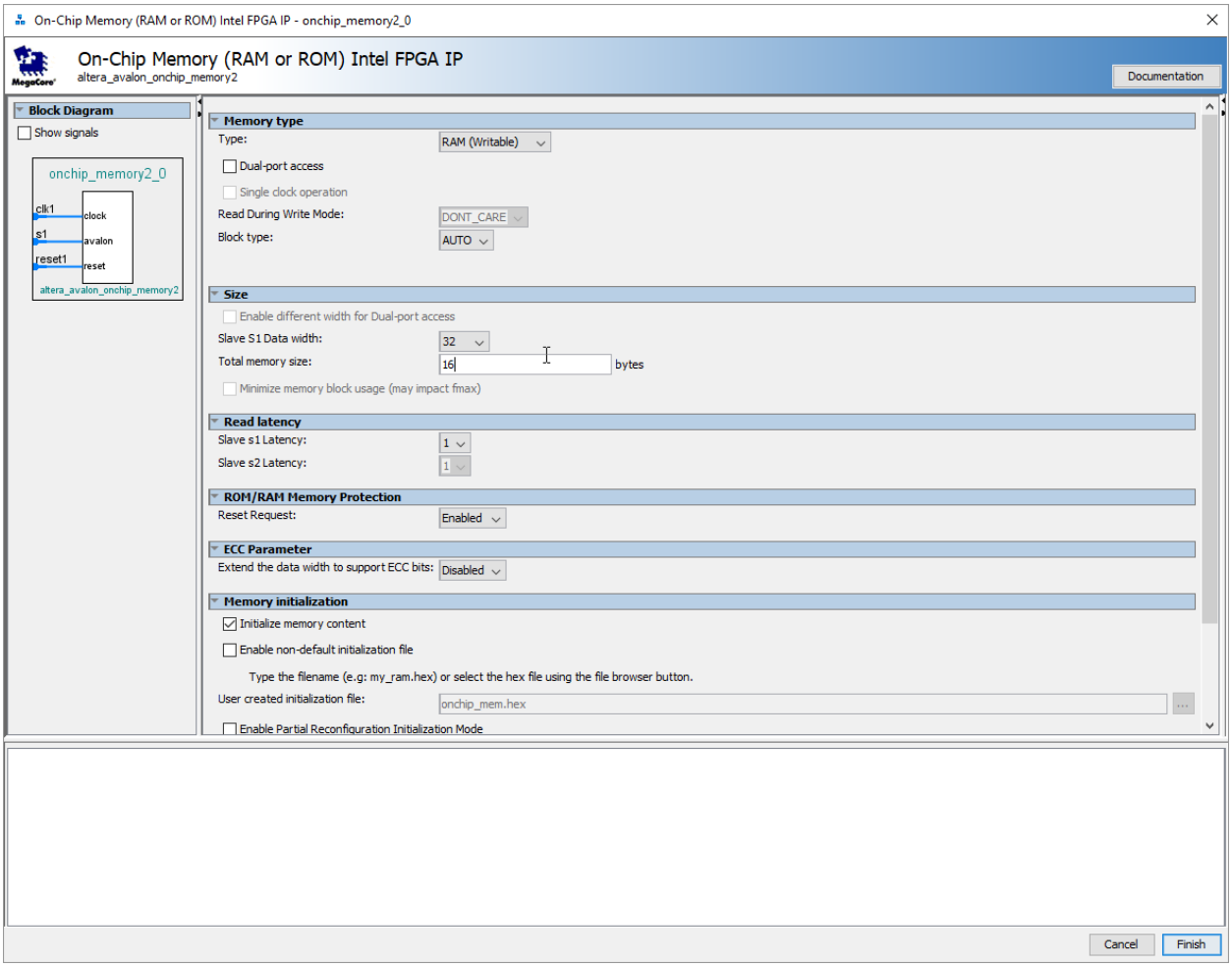


Figure 8

Next, we establish the connections between the Nios II processor and the on-chip memory. In the ***Connections*** column of the Qsys window, click on the empty circles to make a connection. Make sure you connect the NIOS II and on-chip memory clock and reset to the external clock and reset. *Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?*

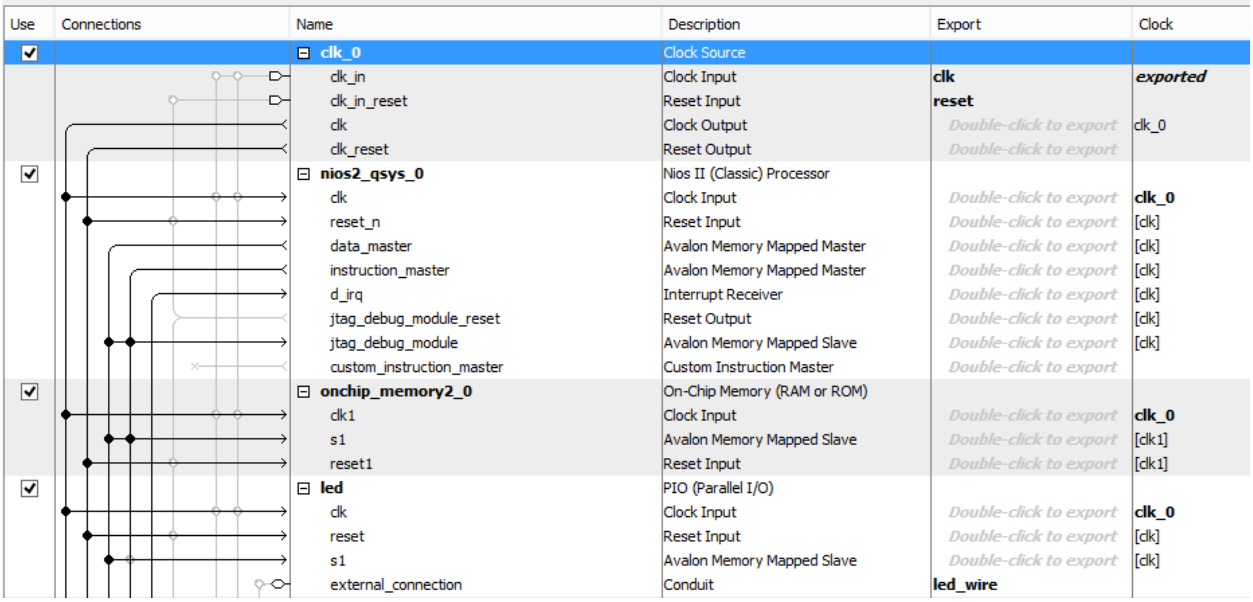


Figure 9

Next, we specify the input parallel I/O interface. This will be used to drive the LEDs and tell us that the system is working correctly. On the left side of the Qsys window, select ***Processors and Peripherals > Peripherals > PIO (Parallel I/O) Intel FPGA IP*** and click ***Add***. Specify the width of the port to be *8* bits. Choose the direction of the port to be ***Output***. Click ***Finish***.

Rename the **PIO** block **led**, so we can keep track of what it is for. Create all the bus connections as shown below between the PIO (led) peripheral and the NIOS II through the Avalon bus. Make the required connections as shown in Figure 9. Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. W*hy might this be the case?*

Since the on-chip memory has limited storage capacity, we will use the off-chip SDRAM to store the software program. SDRAM cannot be interfaced to the bus directly, as it has a complex row/column addressing scheme and requires constant refreshing to retain data. We will use a SDRAM controller IP core to interface the SDRAM to the Avalon bus. *Why does SDRAM require constant refreshing?*

You will need to determine the following parameters to instantiate the SDRAM controller. Refer to the DE2-115 schematic and the ISSIIS16320D SDRAM (note there are two on the board) datasheet (PDF). Make sure you are looking at the correct part of the datasheet (the two chips each provide 16 bits)

SDRAM parameter	Short name	Parameter value (fill in from datasheet)
Data Width	[width]	
# of Rows	[nrows]	
# of Columns	[ncols]	
# of Chip Selects	[ncs]	
# of Banks	[nbanks]	

Note that there are two 32M*16 chips, so the total amount of memory should be 1Gbit (128 Mbytes), *make sure this is consistent with your above numbers; you will need to justify how you came up with 1 Gbit to your TA.*

On the left side of the Qsys window, select ***Memory Interfaces and Controllers*** > ***SDRAM*** > ***SDRAM Controller Intel FPGA IP*** and click ***Add***. A window should pop up, as shown in Figure 14. On the ***Memory Profile*** tab, set the ***Data Width*** to be ***[width]*** bits, ***Address Width*** to be ***[nrows]*** rows and **[ncols]** columns. Make sure **[ncs]** and **[nbanks]** are correct as well. In general, we would have to also research all the SDRAM timings from the datasheet; however this has been done for you. On the ***Timing*** tab, enter the numbers according to Figure 10. *What is the maximum theoretical transfer rate to the SDRAM according to the timings given?* Click ***Finish***. Rename the component as *sdram*.

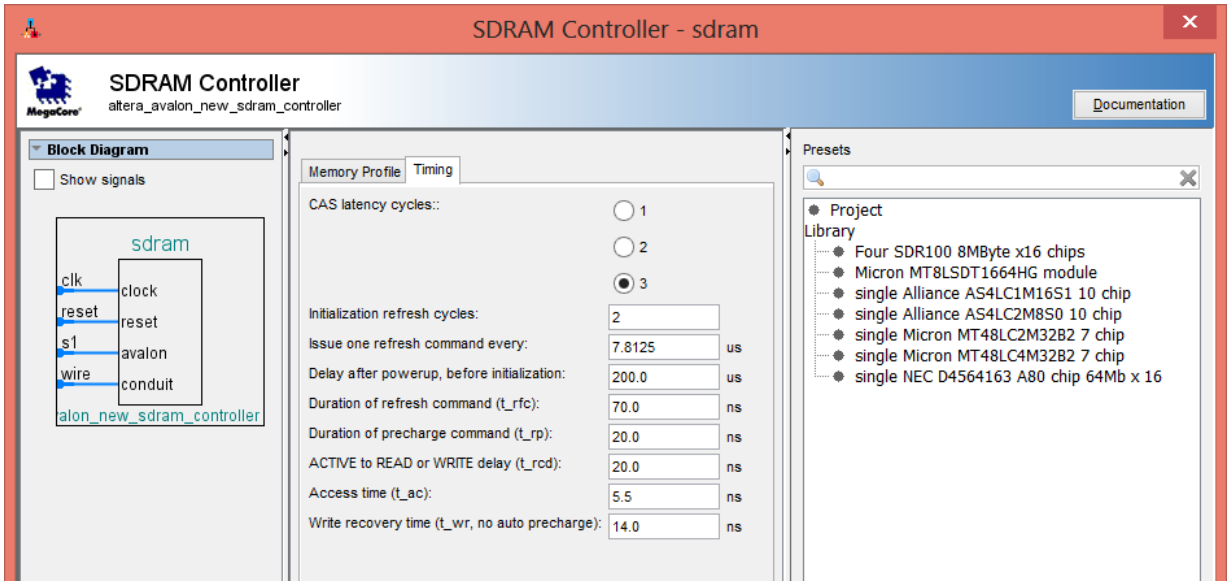


Figure 10

Next, we add a PLL component to provide the required clock signal for the SDRAM chip. This is because the SDRAM requires precise timings, and the PLL allows us to compensate for clock skew due to the board layout. The SDRAM also cannot be run too slowly (below 50 MHz). W*hy might this be the case?* A block diagram for how we will use the SDRAM controller is shown in Figure 11.

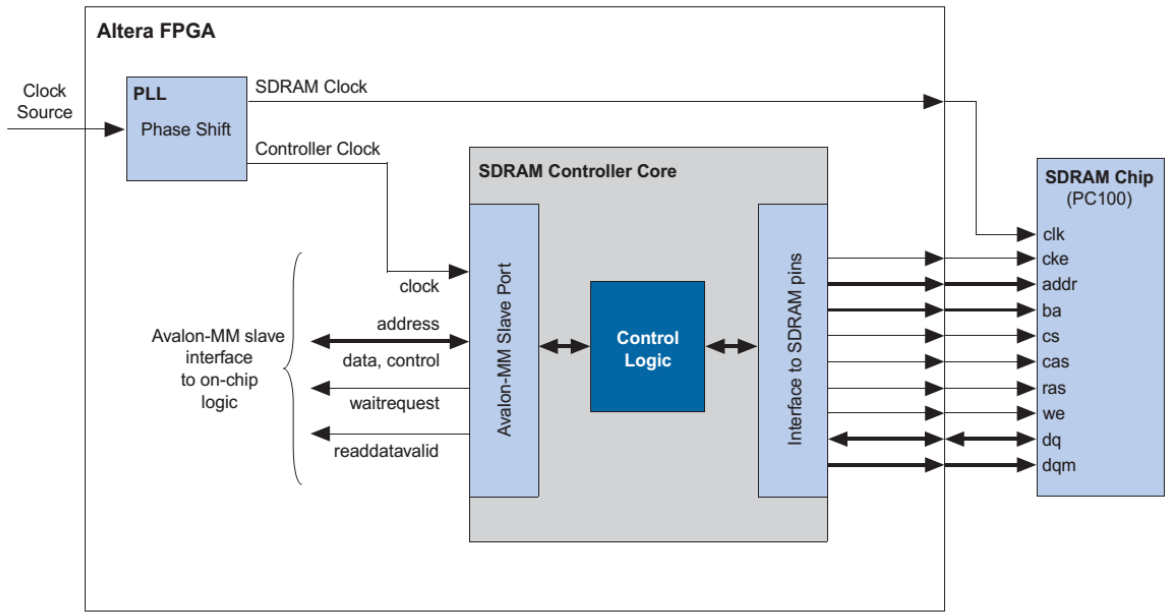


Figure 11

On the left side of the Qsys window, select ***Basic Functions > Clocks; PLLs and Resets > PLL > ALTPLL Intel FPGA IP*** and click ***Add***. A window should pop up as shown in Figure 12. Choose device speed to be ***7***, and the frequency of the ***inclk0*** input (input clock) to be ***50*** MHz. Click ***Next***.

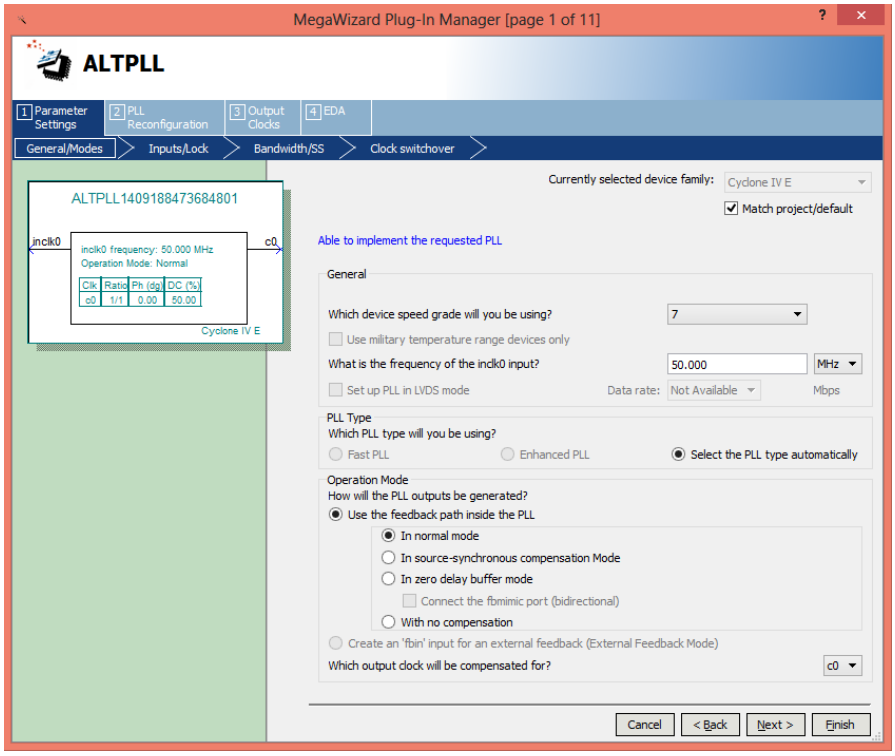


Figure 12

On page 2 (Inputs/Lock), deselect all the checked boxes because we do not need these additional ports (e.g. locked status) in this lab, as shown in Figure 17. Skip Bandwidth/SS and Clock Switchover and the entire [2] PLL Reconfiguration tab by clicking ***Next*** until you get to **[3] Output Clocks**.

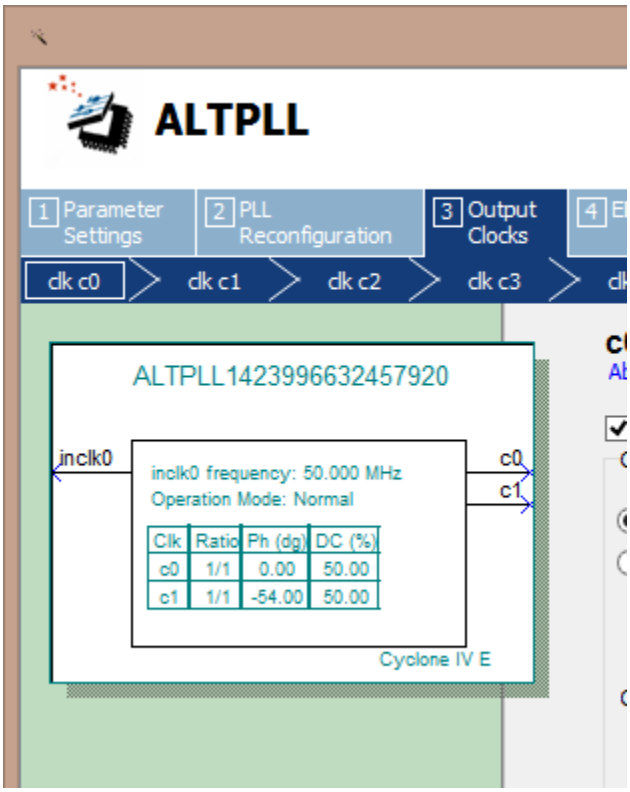


Figure 13

On page **[3] Output Clocks** for **clk c0**, make sure the actual clock frequency is 50 MHz, and set the clock phase shift to be ***0ns***. On the left, observe that the *Ph (dg)* (phase in degrees) is now *0*. This is the clock which goes to the SDRAM controller, and should be synchronous to the rest of the design, this is the *controller clock*, as shown in Figure 11.

You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking ***clk c1***, and verify it has the same settings, except that the phase shift should be **-3ns**. This puts the clock going out to the SDRAM chip (***clk c1***) 3ns behind of the controller clock (***clk c0***). *Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.* The configuration for ***clk c1*** should look like Figure 14. Now complete the rest of the wizard by clicking ***Finish*** and the ***Finish*** again. Name your new module *sdram_pll*, to remind us what this is for. Verify that there are indeed two ports as shown in Figure 13, one with 0 degrees shift, going out to the controller, one with -54 degrees shift, going out to the SDRAM chip.

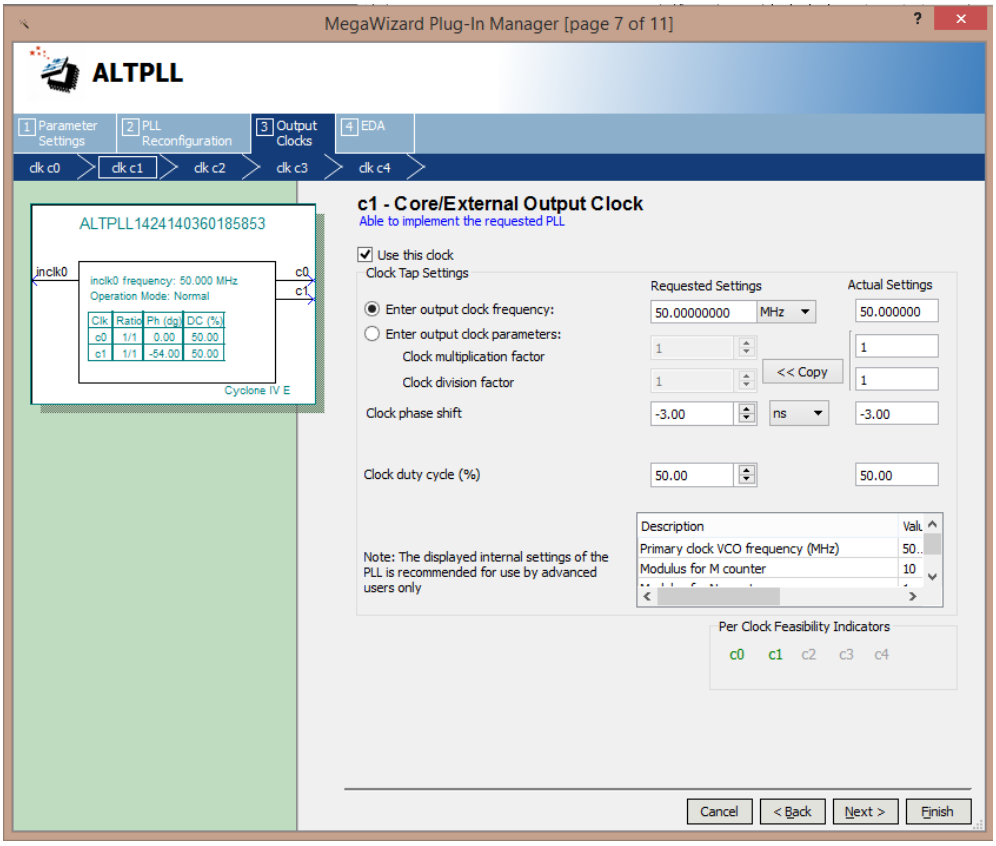


Figure 14

Next, we add a system ID checker to ensure the compatibility between hardware and software. This module gives us a serial number (we'll assign it to 0), which the software loader checks against when we start the software. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration (or an FPGA without a NIOS II at all). For example, if we had added a new NIOS II peripheral and forgot the regenerate/reprogram the FPGA, this block would prevent us from trying to load software from the old configuration onto our incompatible NIOS II. On the left side of the Qsys window, select ***Basic Functions > Simulation; Debug and Verification > System ID Peripheral Intel FPGA IP*** and click ***Add***. Accept the default settings and click ***Finish***, as per Figure 15.

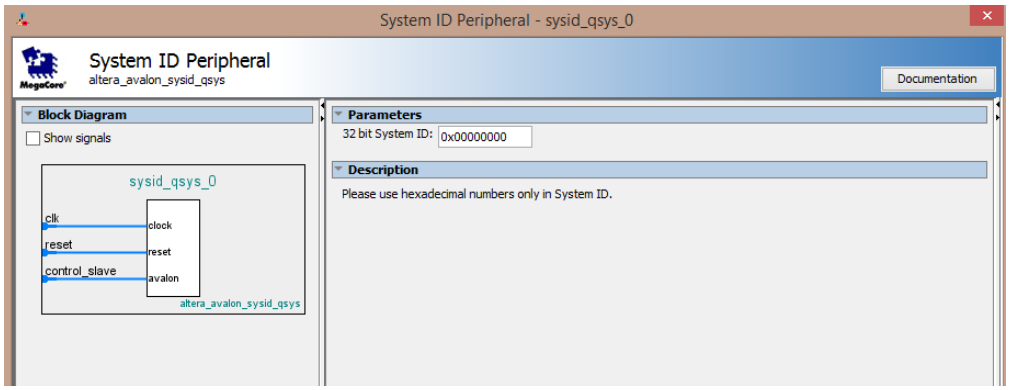


Figure 15

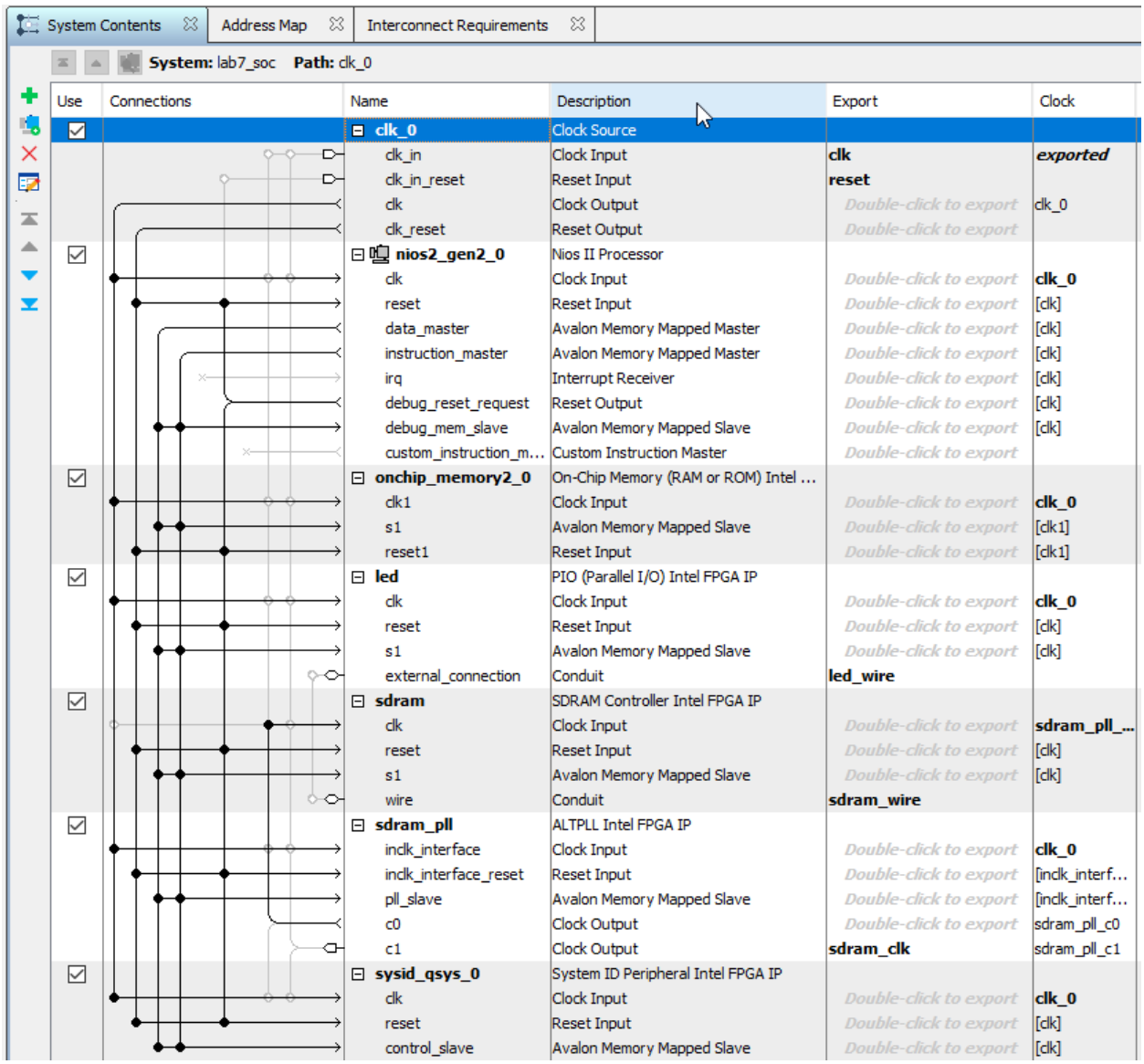


Figure 16

Next, we make the necessary connections for the newly added components, as shown in Figure 16. **Note that the *sdram* is connected to the *sdram_pll_c0* instead of the main clock**, as per Figure 11. Next, enable external access to the clock, reset, PIO, SDRAM, and the other clock out from the PLL (to the physical SDRAM chip). Do this by clicking on the ***Double-click to export*** in the ***Export*** column. Name the PIO(led) port ***led_wire***. Also, click on the ***Double-click to export*** on the right of the *sdram* wire. Type in the name ***sdram_wire***. This "breaks out" the connections to the rest of the world for the SDRAM controller (to the SDRAM chips, which are external to the FPGA itself), the PIO block, the reset, and the second port of the PLL. Export the second output of the PLL (***sdram_pll_c1)*** as ***sdram_*clk**. When you are done, make sure following connections are exported:

sdram_wire	***led_wire***	***reset***	***clk***	***sdram_clk***

These are the connections your SystemVerilog top-level needs to connect to the appropriate external hardware.

Finally, we assign memory addresses to each component. This can be done automatically in Qsys, but we want to make sure that the on-chip memory gets the base address of 0x0000_0000. Click on the base address of on-chip memory and enter ***0x0000_0000***, and then click on the lock symbol on the left to make it fixed. On the menu, click on ***System > Assign Base Addresses*** to have Qsys automatically assign memory addresses to the other peripherals. The resulting Qsys should be similar (but maybe not identical) to Figure 16. Note that these addresses will be used in the software program later in this tutorial.

After we have the memory set up, we can assign the reset and exception vectors for the Nios II processor. Right click on *nios2_qsys_0* and click ***Edit***. For both reset and exception vectors, choose *sdram.s1*, as shown in Figure 17. Do not change default settings for offsets. Click ***Finish***. *What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?*

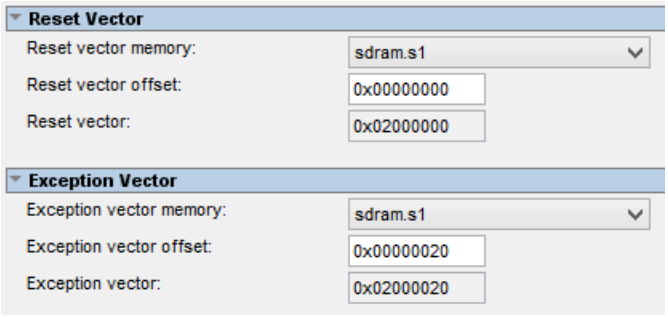


Figure 17

Now, we have specified all the necessary components. Save the system as lab7_soc. Then, go to the ***Generate > Generate HDL*** from the main menu. Specify the settings as shown in Figure 18, and then click ***Generate***.

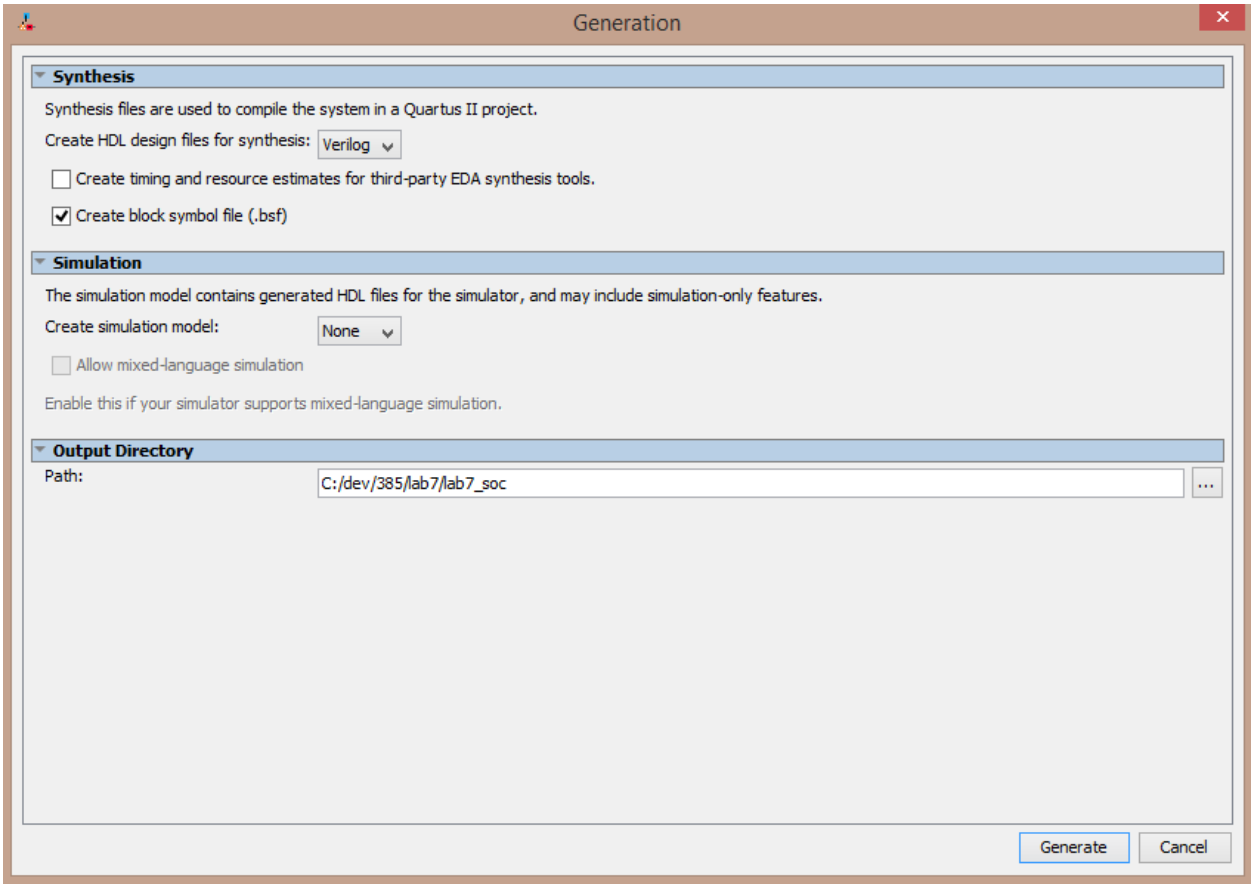


Figure 18

After generation is complete, we can find the synthesized NIOS II system-on-chip component in lab7\lab7_soc\synthesis\lab7_soc.v. It is synthesized in Verilog, but the module interface is compatible with the SystemVerilog design we are going to create in this project.

Integration of the Nios II System into a Quartus Prime Project:

In Quartus II, go to the ***Files*** tab on the left. Right click on ***Files*** and select ***Add/Remove Files in Project***. Add lab7_soc\synthesis\lab7_soc.qip into the project, which gives the required information for Quartus II to include the system we created. Then, add the lab7 top-level SystemVerilog file *lab7.sv* (given on the course website) as a new file. An instance of the ***lab7*** top level module is now created, as shown in Figure 19.

```
module lab7(      input          CLOCK_50,
                  input  [1:0]   KEY,
                  output [7:0]   LEDG,
                  output [12:0]  DRAM_ADDR,
                  output [1:0]   DRAM_BA,
                  output         DRAM_CAS_N,
                  output         DRAM_CKE,
                  output         DRAM_CS_N,
                  inout  [31:0]  DRAM_DQ,
                  output  [3:0]  DRAM_DQM,
                  output         DRAM_RAS_N,
                  output         DRAM_WE_N,
                  output         DRAM_CLK

               );
```

Figure 19

Open the generated Verilog file for the Nios II system, lab7/lab7_soc/synthesis/lab7_soc.v to take a look at the SoC module you created. This will have ports and names based on what you exported from Qsys. Make sure the names are consistent with what is expected in the top-level (lab7.sv) as in Figure 20. At this point, if your names are consistent, you should be able to synthesize your design without any errors (but not necessarily implement, since we have no pin assignments yet).

```
`timescale 1 ps / 1 ps
module lab7_soc (
      input  wire        clk_clk,           //        clk.clk
      output wire [7:0]  led_wire_export,   //   led_wire.export
      input  wire        reset_reset_n,     //      reset.reset_n
      output wire        sdram_clk_clk,     //  sdram_clk.clk
      output wire [12:0] sdram_wire_addr,   // sdram_wire.addr
      output wire [1:0]  sdram_wire_ba,     //            .ba
      output wire        sdram_wire_cas_n,  //            .cas_n
      output wire        sdram_wire_cke,    //            .cke
      output wire        sdram_wire_cs_n,   //            .cs_n
      inout  wire [31:0] sdram_wire_dq,     //            .dq
      output wire [3:0]  sdram_wire_dqm,    //            .dqm
      output wire        sdram_wire_ras_n,  //            .ras_n
      output wire        sdram_wire_we_n    //            .we_n
   );
```

Figure 20

Do the necessary pin assignments as follows. (Hint: Download DE2-115.qsf from the course website. In Quartus, go to ***Assignments*** > ***Import Assignments*** to load the default DE2 pin assignment settings. Make modifications as needed. Note that these names are from our top-level SystemVerilog file, even though for this lab, our top level doesn't do more than wire the NIOS II module (lab7_soc) to the outside world.

INQ. 16

Port Name	Location	Comments
CLOCK_50	PIN_Y2	50 MHz Clock from the on-board oscillators
KEY[3]	PIN_R24	On-board push button
KEY[2]	PIN_N21	On-board push button
KEY[0]	PIN_M23	On-board push button
LEDG[7]	PIN_G21	On-board LED
LEDG[6]	PIN_G22	On-board LED
LEDG[5]	PIN_G20	On-board LED
LEDG[4]	PIN_H21	On-board LED
LEDG[3]	PIN_E24	On-board LED
LEDG[2]	PIN_E25	On-board LED
LEDG[1]	PIN_E22	On-board LED
LEDG[0]	PIN_E21	On-board LED
DRAM_ADDR[12]	PIN_Y7	On-board SDRAM
DRAM_ADDR[11]	PIN_AA5	On-board SDRAM
DRAM_ADDR[10]	PIN_R5	On-board SDRAM
DRAM_ADDR[9]	PIN_Y6	On-board SDRAM
DRAM_ADDR[8]	PIN_Y5	On-board SDRAM
DRAM_ADDR[7]	PIN_AA7	On-board SDRAM
DRAM_ADDR[6]	PIN_W7	On-board SDRAM
DRAM_ADDR[5]	PIN_W8	On-board SDRAM
DRAM_ADDR[4]	PIN_V5	On-board SDRAM
DRAM_ADDR[3]	PIN_P1	On-board SDRAM
DRAM_ADDR[2]	PIN_U8	On-board SDRAM
DRAM_ADDR[1]	PIN_V8	On-board SDRAM
DRAM_ADDR[0]	PIN_R6	On-board SDRAM
DRAM_BA[1]	PIN_R4	On-board SDRAM
DRAM_BA[0]	PIN_U7	On-board SDRAM
DRAM_CAS_N	PIN_V7	On-board SDRAM
DRAM_CKE	PIN_AA6	On-board SDRAM
DRAM_CLK	PIN_AE5	On-board SDRAM
DRAM_CS_N	PIN_T4	On-board SDRAM
DRAM_DQ[31]	PIN_U1	On-board SDRAM
DRAM_DQ[30]	PIN_U4	On-board SDRAM
DRAM_DQ[29]	PIN_T3	On-board SDRAM
DRAM_DQ[28]	PIN_R3	On-board SDRAM
DRAM_DQ[27]	PIN_R2	On-board SDRAM
DRAM_DQ[26]	PIN_R1	On-board SDRAM
DRAM_DQ[25]	PIN_R7	On-board SDRAM
DRAM_DQ[24]	PIN_U5	On-board SDRAM
DRAM_DQ[23]	PIN_L7	On-board SDRAM
DRAM_DQ[22]	PIN_M7	On-board SDRAM
DRAM_DQ[21]	PIN_M4	On-board SDRAM
DRAM_DQ[20]	PIN_N4	On-board SDRAM
DRAM_DQ[19]	PIN_N3	On-board SDRAM
DRAM_DQ[18]	PIN_P2	On-board SDRAM
DRAM_DQ[17]	PIN_L8	On-board SDRAM
DRAM_DQ[16]	PIN_M8	On-board SDRAM
DRAM_DQ[15]	PIN_AC2	On-board SDRAM
DRAM_DQ[14]	PIN_AB3	On-board SDRAM
DRAM_DQ[13]	PIN_AC1	On-board SDRAM
DRAM_DQ[12]	PIN_AB2	On-board SDRAM
DRAM_DQ[11]	PIN_AA3	On-board SDRAM

DRAM_DQ[10]	PIN_AB1	On-board SDRAM
DRAM_DQ[9]	PIN_Y4	On-board SDRAM
DRAM_DQ[8]	PIN_Y3	On-board SDRAM
DRAM_DQ[7]	PIN_U3	On-board SDRAM
DRAM_DQ[6]	PIN_V1	On-board SDRAM
DRAM_DQ[5]	PIN_V2	On-board SDRAM
DRAM_DQ[4]	PIN_V3	On-board SDRAM
DRAM_DQ[3]	PIN_W1	On-board SDRAM
DRAM_DQ[2]	PIN_V4	On-board SDRAM
DRAM_DQ[1]	PIN_W2	On-board SDRAM
DRAM_DQ[0]	PIN_W3	On-board SDRAM
DRAM_DQM[3]	PIN_N8	On-board SDRAM
DRAM_DQM[2]	PIN_K8	On-board SDRAM
DRAM_DQM[1]	PIN_W4	On-board SDRAM
DRAM_DQM[0]	PIN_U2	On-board SDRAM
DRAM_RAS_N	PIN_U6	On-board SDRAM
DRAM_WE_N	PIN_V6	On-board SDRAM

For timing analysis purposes, add the timing constraint file lab7.sdc (given on the course website). The lines in the file describe the main clock and the SDRAM clock, and the external SDRAM clock. Also, the maximum input and output pin delays are specified and checked against. You may need to modify the file to account for your different signal names. ***If you add additional inputs/outputs (which you will need to) you must add additional constraints to this file. Any unconstrained paths in your final demo will cause you to lose points, since there is no guarantee your design will work.***

Finally, compile the project. In the ***Programmer***, find the .sof file in lab7/output_files. Program the circuit onto the FPGA.

Software Setup:

In Quartus II, go to ***Tools > Nios II Software Build Tools for Eclipse*** to launch the software development environment.

The Eclipse window will show up, as in Figure 21.

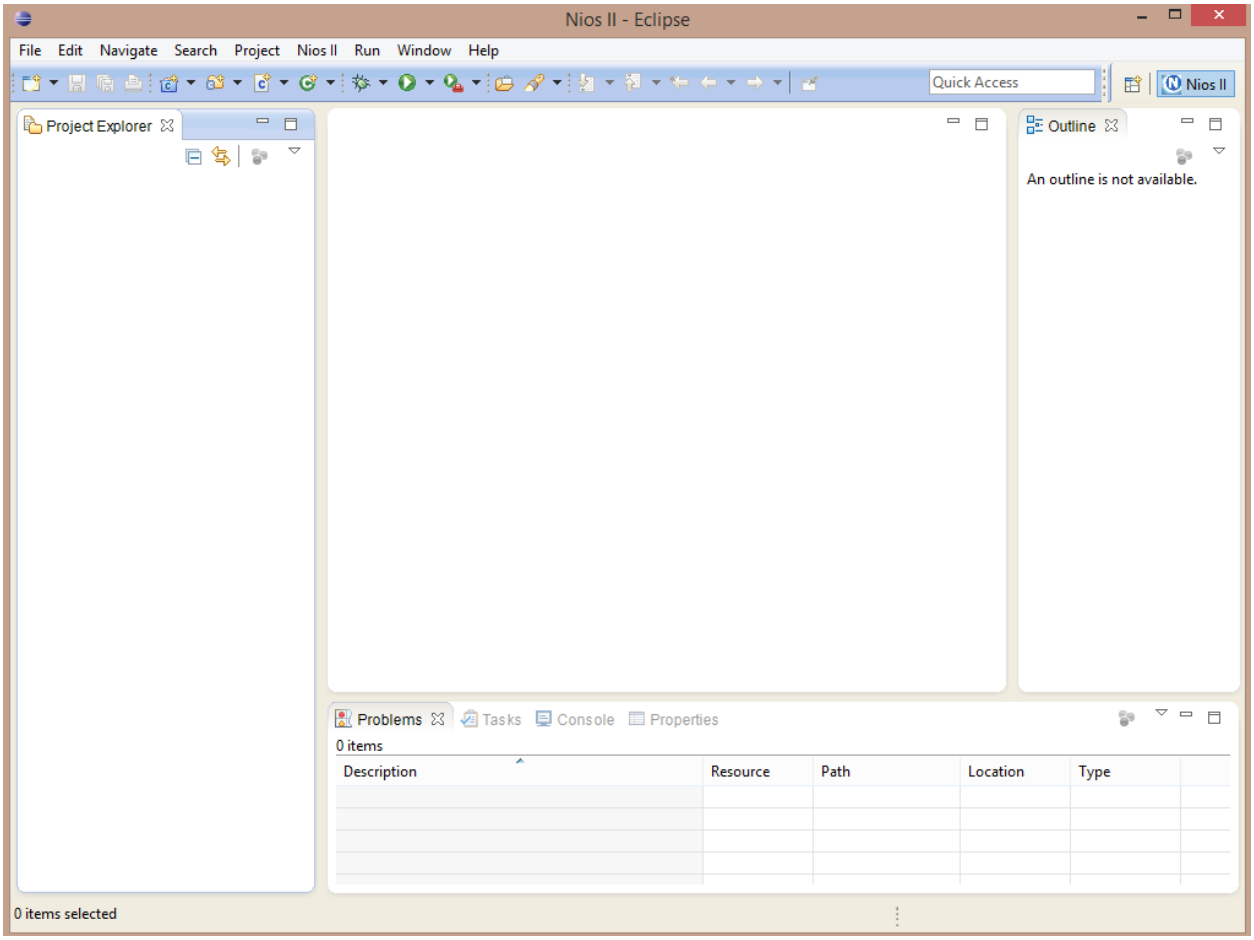


Figure 21

Eclipse works by having different "perspectives", which represent UI layouts for different tasks common to software development. If the default perspective is not Nios II, set perspective to Nios II by going to ***Menu > Window> Open Perspective > Other > Nios II***.

Create a new software program by clicking on ***File > New… > Nios II Application and BSP from Template***. A window should pop up, as shown in Figure 26. Set the SOPC Information File name to be lab7\lab7_soc.sopcinfo. The CPU name will be automatically determined. Select "Blank Project" from the available project templates on the lower left corner. Enter Project name as lab7_app. Click ***Finish***.

Two projects, *lab7_app* and *lab7_app_bsp* are generated. The bsp project contains the hardware information needed in the Makefile to compile the program. For example, the bsp contains the linker script, which tells the linker where to put various segments in memory as according to your address map. Click on the *lab7_app* project and create a new file named *main.c* and copy the contents from the *main.c* as supplied from the website, as shown in Figure 22.

main.c

```
1  // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5  int main()
6  {
7      int i = 0;
8      volatile unsigned int *LED_PIO = (unsigned int*)0x20; //make a pointer to acc
9
10     *LED_PIO = 0; //clear all LEDs
11     while ( (1+1) != 3) //infinite loop
12     {
13         for (i = 0; i < 100000; i++); //software delay
14         *LED_PIO |= 0x1; //set LSB
15         for (i = 0; i < 100000; i++); //software delay
16         *LED_PIO &= ~0x1; //clear LSB
17     }
18     return 1; //never gets here
19 }
20
```

Figure 22

At the beginning of the program, fill in the actual address assigned by Qsys to the assignment for pointer *LED_PIO. The address information can be found in the Qsys window. *You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 8), and how the set and clear functions work by working out an example on paper (lines 13 and 16).*

Right click on *lab7_app_bsp* project and select ***Nios II > Generate BSP.*** Then right click on the same project and, go to ***Nios II > BSP Editor***. A window should pop up, as shown in Figure 23. Make sure your settings are the same as shown in the figure. In particular, the *exception_stack_memory_region_name* and the *interrupt_stack_memory_region_name* should be set to *sdram*.

INQ. 20

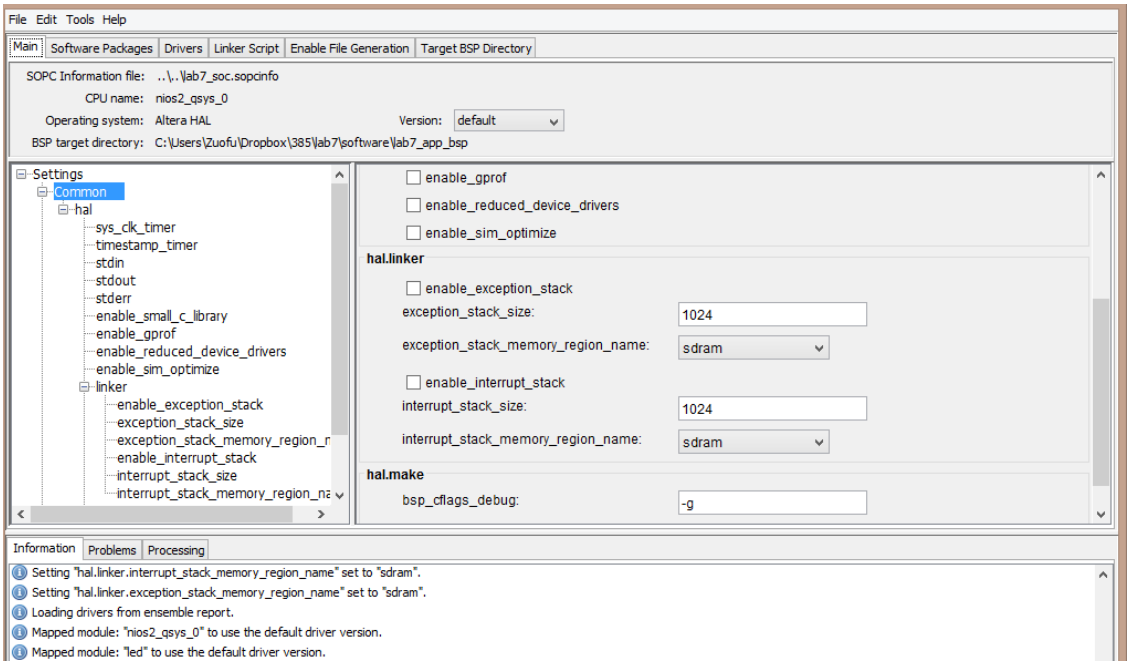


Figure 23

Finally, we have to setup the linker script. The linker script tells the linker which addresses to place the various segments of compiled code. Go to the ***Linker Script*** tab. Make sure all the linker regions and memory devices are set to *sdram*, as shown in Figure 24. Click ***Generate***.

Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code:

```
const int my_constant[4] = {1, 2, 3, 4}
```

will place 1, 2, 3, 4 into the .rodata segment.

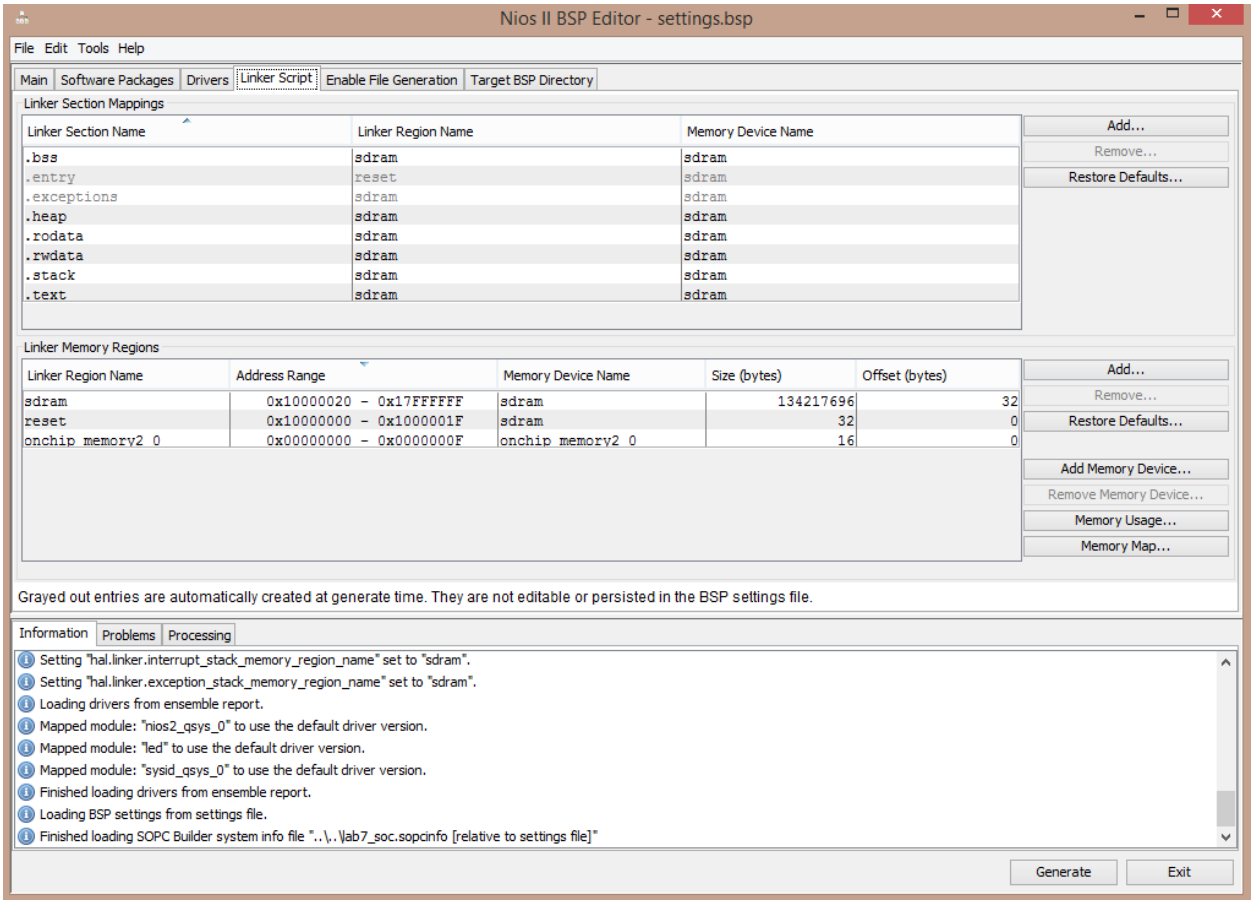


Figure 24

Now, we are ready to compile the program. Go to ***Project > Build All*** to compile the program.

IMPORTANT:
Whenever the hardware part is changed, it needs to be compiled in Quartus and programmed on the FPGA. If the programmer fails to load the .sof file on the FPGA, it's likely because the software is occupying the USB Blaster port. Simply stop the program or restart the FPGA board (and reprogram the FPGA) if this is the case.

On the software side, make sure to right click on *lab7_app_bsp* and select ***Nios II > Generate BSP*** so the latest hardware information is included in the Makefile. Then, compile the program again (***Build All***). The System ID peripheral should report an errorif you try to load software onto an incompatible hardware platform, but in either case compatibility errors occur if you fail to maintain software-hardware consistency!

Running the program Nios II:

In Eclipse, click on ***Run*** > ***Run Configurations***… to open up a dialog window, as shown in Figure 25.

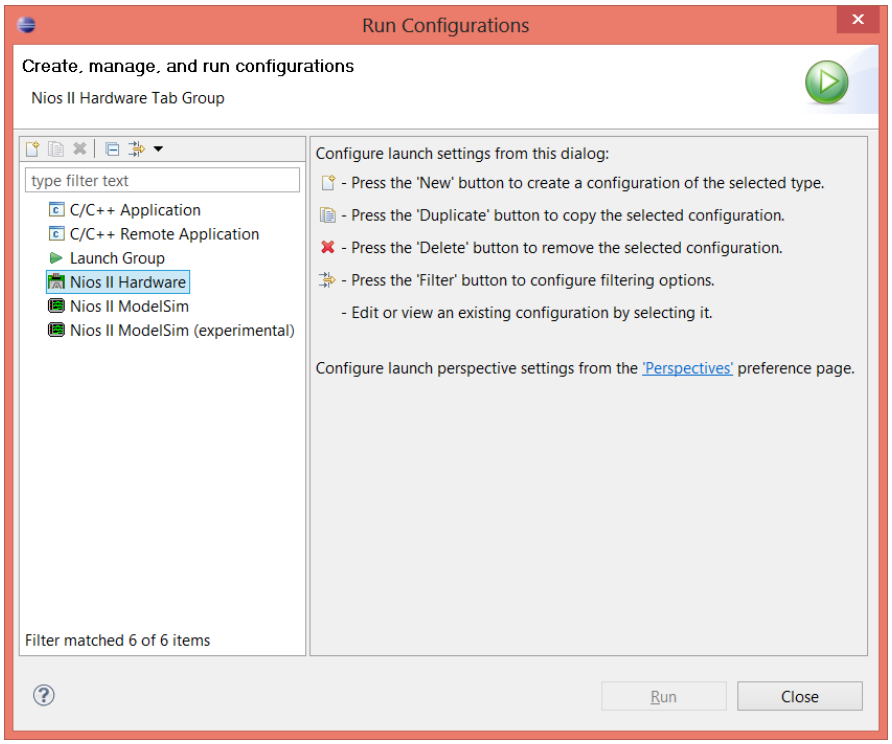


Figure 25

Right click on ***Nios II Hardware*** and select ***New***. Type in the name of the configuration as *DE2*. In the ***Project*** tab, select Project name to be *lab7_app*. The corresponding ELF file should automatically be set up. See Figure 26. The ELF file is the compiled software file that is downloaded into the system memory and run on Nios II, equivalent to an ".exe" in Windows.

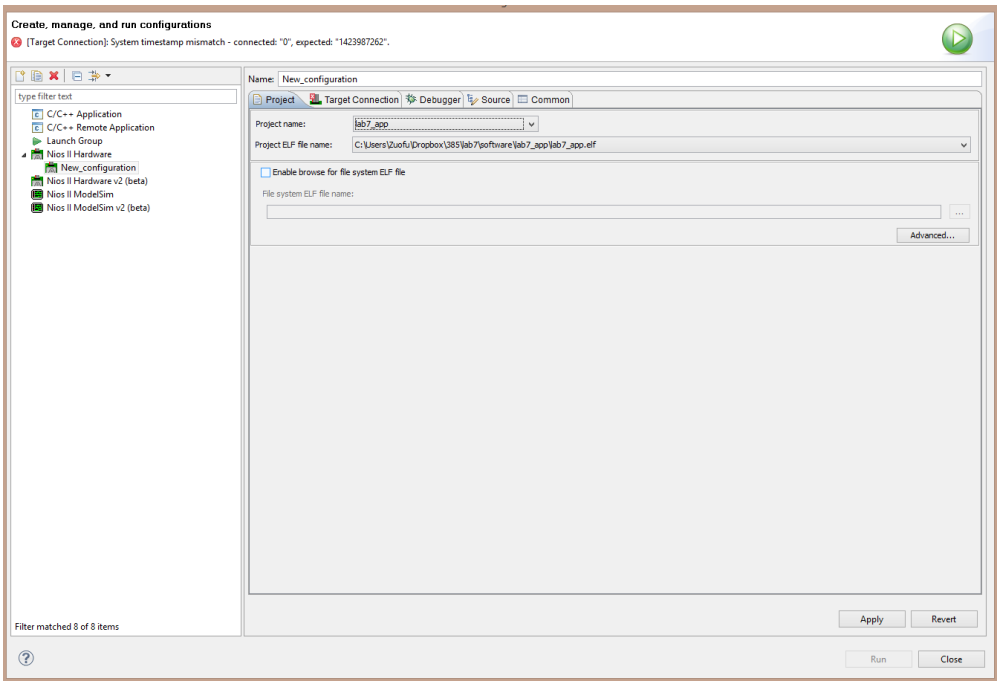


Figure 26

Go to the ***Target Connection*** tab. Click on Refresh Connections on the right. Make sure the *Processors* is listed as *USB-Blaster on localhost (in Figure 27)*. If an error message "*Connected system ID hash not found on target at expected base address*" appears, something is wrong with your NIOS II system, your computer cannot connect to the "sysid" block that you created in Qsys. Check to make the FPGA is programmed and that all the pin assignments are correct and click System ID Properties to refresh.

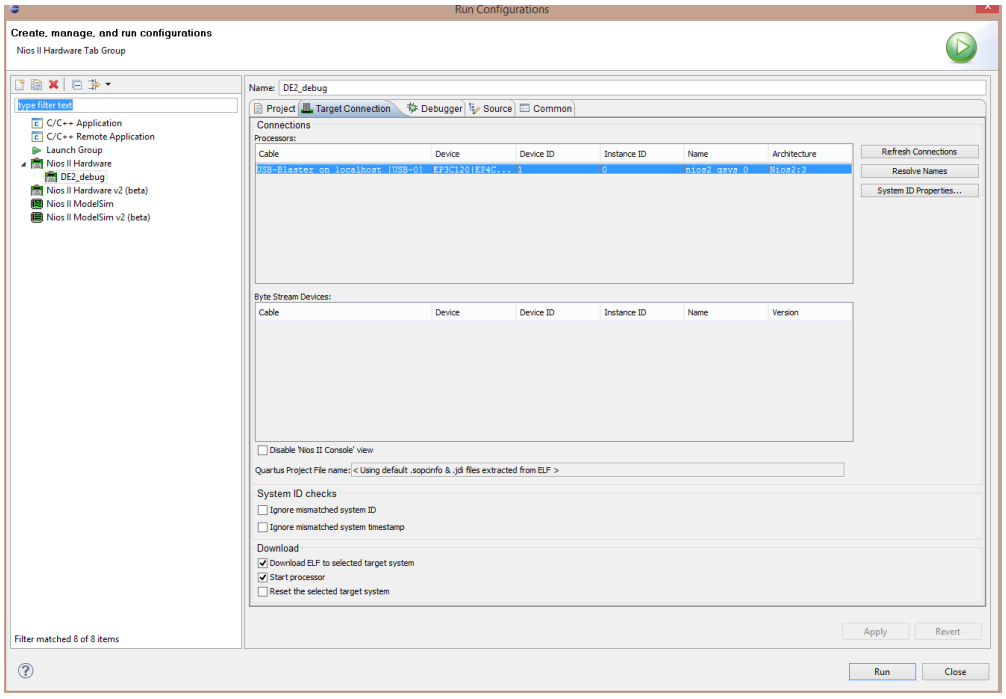


Figure 27

Click ***Run*** to run the program. Your LEDG0 should start blinking on your board as soon as the binary has been transmitted to the NIOS II CPU. Note that to do the actual lab assignment; you must add another PIO block as input corresponding to the switches. Do not forget you must export the PIO module from your SoC, modify the top-level SystemVerilog, re-assign memory addresses, assign the correct pins, and set the correct timing constraints. The following file will also be useful: Altera Embedded Peripheral IP datasheet, specifically the chapter on PIO.

INTRODUCTION TO
USB and EZ-OTG on NIOS II

ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to USB and EZ-OTG on Nios II

Embedded System with Nios II

In Lab 7, we introduced the Nios II embedded processor. We said in class that the Nios II is ideal for low speed tasks which would require a huge number of states/logic to do in hardware. USB enumeration for a HID device is one of these tasks, since the speed (for a human-interface device such as a keyboard) is very low, but there are a prohibitive number of states/cases to efficiently handle in hardware.

In Lab 8, the USB protocol is handled in the software on the Nios II, and the extracted keycode from the USB keyboard is then sent to the hardware for further use. In this tutorial, you will build a Platform Designer interface to handle to I/O interface to communicate with the USB-OTG chip. Then you will learn how to import existing NIOS II software and make changes to it to get the keyboard to work correctly.

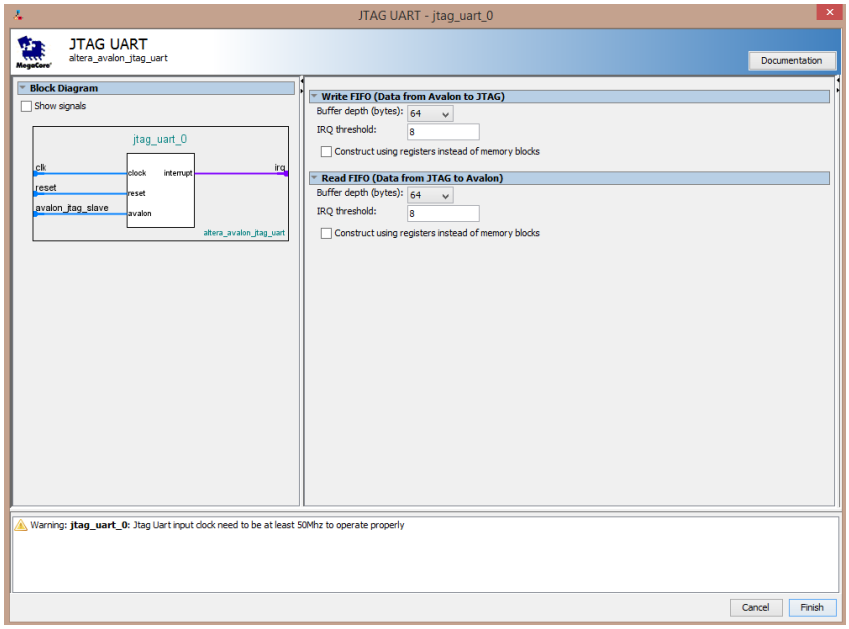
You should start with a working lab 7 Platform Designer setup. If you don't have one, you should follow the tutorial for lab 7 to complete the Platform Designer setup. Always keep a backup copy of old files that you are reusing especially if you know if they were working with the older version.

Starting from lab 7, remove the PIO modules in Platform Designer which correspond to the LEDs and switches you used in lab 7, as we will not be using them for lab 8.

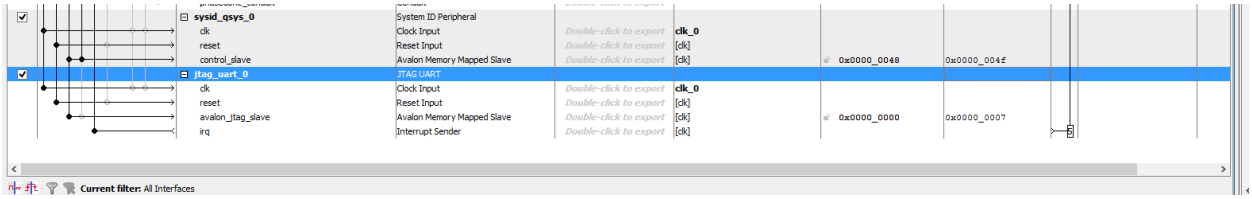
You will then need to add the JTAG UART peripheral. This is found under Interface Protocols->Serial. This is so that you can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

You can simply leave the settings here as a default. What this block does is give you the ability to use console (printf) commands from the Nios II which go through the programming cable via USB. While this is typically not a good user interface for an embedded system (as it requires the programming cable to be connected and the user to have all the Altera software installed), this is an excellent way to debug your software while in development.

IUQ. 2



Make the connections as shown below:



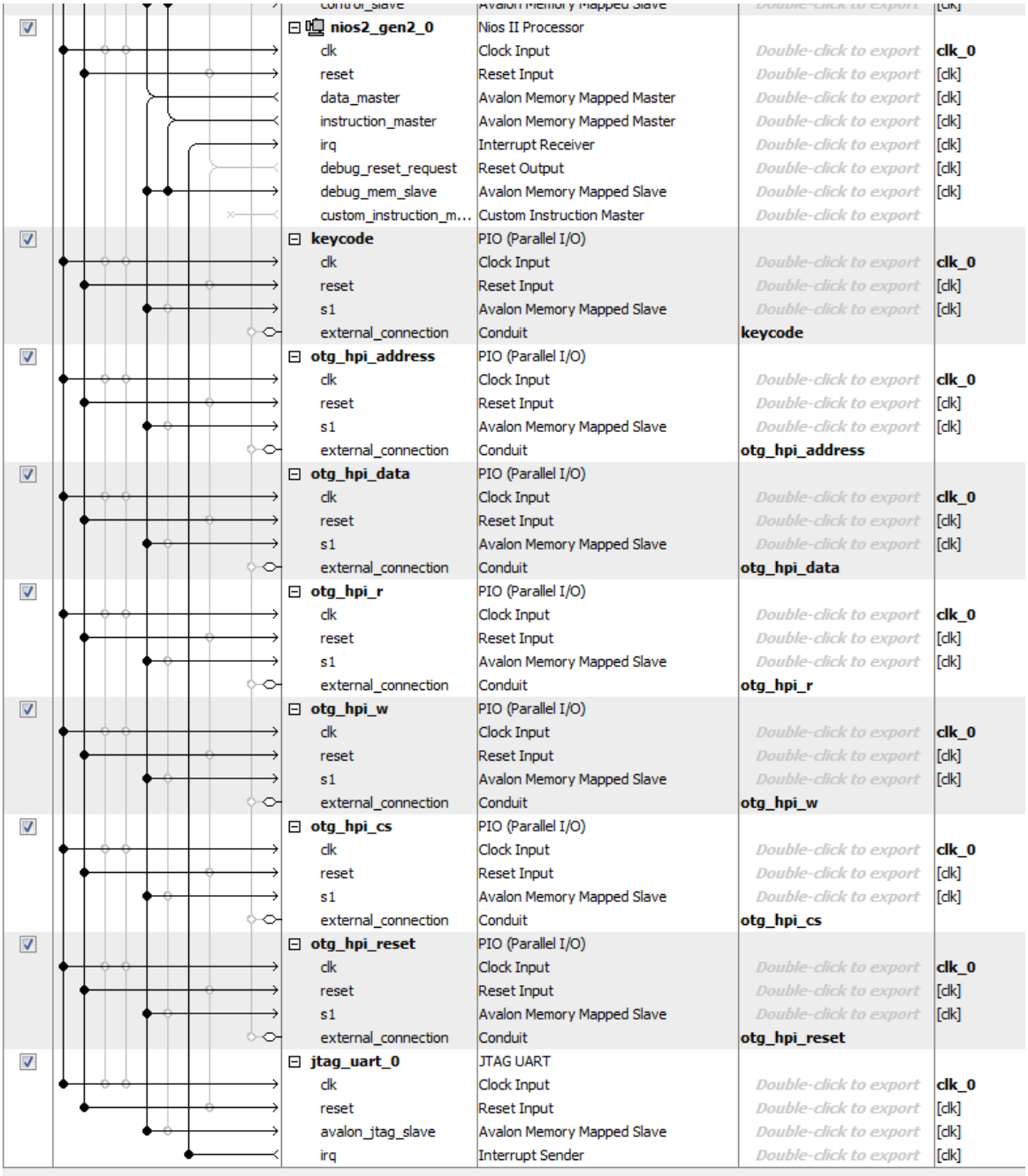
Make the standard connections for clk/reset/data bus, which is what swe've been doing for the other peripherals. One difference here is that we must assign an **interrupt** for this, which we will assign IRQ (interrupt request) 5. Connect the interrupt controller to IRQ and give it the number 5 (this is on the far right of the row. The reason for using interrupts here is that transmitting or receiving text over the console is in general very slow, and we don't want this procedure to block on the CPU. Therefore, the typical way in which printf is implemented is that control is returned program as soon as printf is called, but an interrupt is set up at the end of transmission for each buffer. This way, the CPU does not have to block while waiting for each of the characters to be transmitted, it is only interrupted whenever the peripheral (the JTAG UART) needs more data.

You need to add multiple PIO connections to the lab 8 setup, these are listed below with the direction of the ports and size of the ports.

Name	Direction	Width
keycode	Out	8
otg_hpi_address	Out	2
otg_hpi_data	Inout	16
otg_hpi_r	Out	1
otg_hpi_w	Out	1
otg_hpi_cs	Out	1
otg_hpi_reset	Out	1

You don’t need the LED and switches PIO’s anymore for this lab, so you can uncheck it to disable it.

This is how all the PIO’s and JTAG UART should look when connected (this only include additions from lab 7).



☑	**nios2_gen2_0**	Nios II Processor		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	data_master	Avalon Memory Mapped Master	Double-click to export	[clk]
	instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]
	irq	Interrupt Receiver	Double-click to export	[clk]
	debug_reset_request	Reset Output	Double-click to export	[clk]
	debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]
	custom_instruction_m…	Custom Instruction Master	Double-click to export	
☑	**keycode**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**keycode**	
☑	**otg_hpi_address**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_address**	
☑	**otg_hpi_data**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_data**	
☑	**otg_hpi_r**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_r**	
☑	**otg_hpi_w**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_w**	
☑	**otg_hpi_cs**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_cs**	
☑	**otg_hpi_reset**	PIO (Parallel I/O)		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**otg_hpi_reset**	
☑	**jtag_uart_0**	JTAG UART		
	clk	Clock Input	Double-click to export	**clk_0**
	reset	Reset Input	Double-click to export	[clk]
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]
	irq	Interrupt Sender	Double-click to export	[clk]

Figure 0

IUQ. 4

You should save the Platform Designer setup and generate HDL and make sure you don't have any errors.

For this next part you need to have a complete hardware top level entity. You need to make sure that all the pin connections mentioned in the lab 8 experiment section exist and are connected to the correct entities. **Remember that your HDL will need to pass the PIO ports to the physical pins on the FPGA which are physically connected to the USB chip. The easiest way to do this is to map them in HDL to the names found in the standard pin assignment file.** To complete this you also need to add in the HDL version of the Platform Designer system to the project by adding the .qip file like you did in lab 7. If all the connections to the qip are made correctly, you should be able to compile the project and program it onto the board.

Go to ***Tools > Nios II Software Build Tools for Eclipse*** to launch the software development environment. Specify the workspace path to be the same as your Quartus II project, as shown in Figure 1.

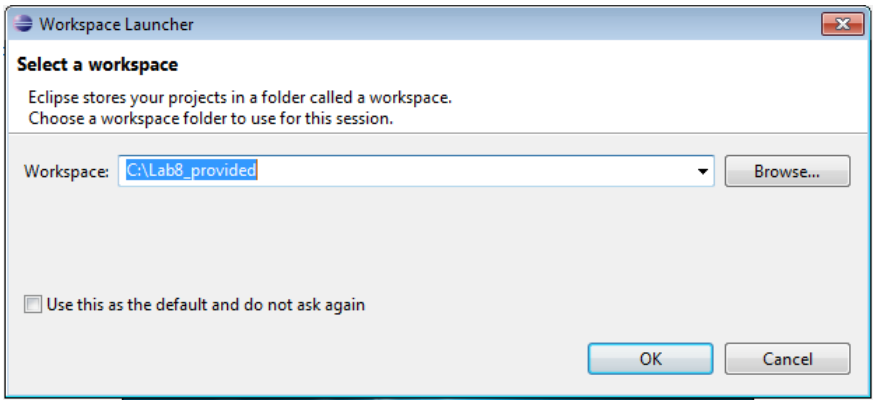


Figure 1

The Eclipse window will show up, as in Figure 2.

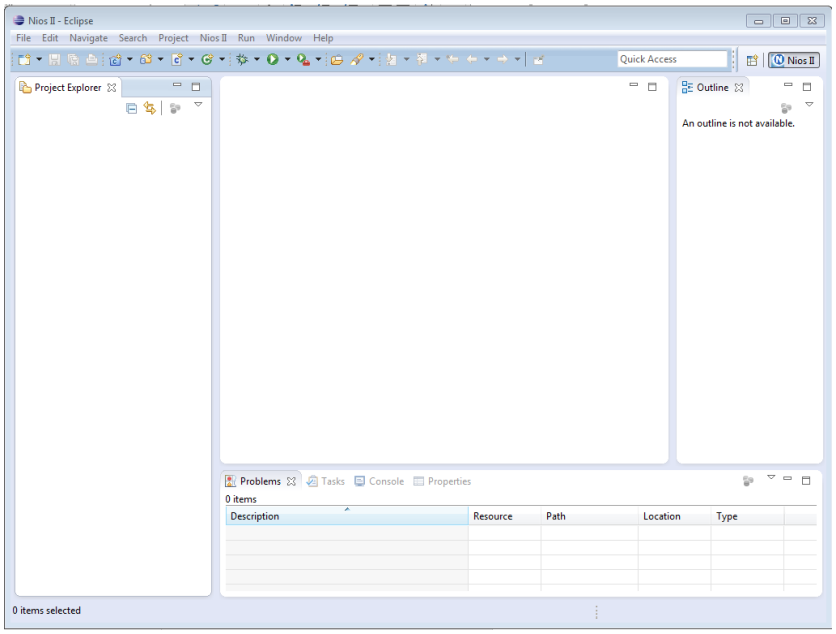


Figure 2

Set perspective to Nios II by going to ***Menu > Window> Open Perspective > Other > Nios II*** or by clicking on the ***Nios II*** icon on the upper right of the window.

Next, we will create a new Eclipse project. Go to ***Menu > New > NIOS II Application and BSP from Template***. In the pop-up window, select the .sopcinfo file generated by Platform Designer in *SOPC Information File name*, and the system should automatically detect the NIOS CPU that you use. Then, type "usb_kb" in Project name, select Blank Project in Templates, and click on **Finish**. This should create two projects *usb_kb*, and *usb_kb_bsp* in the Project Explorer.

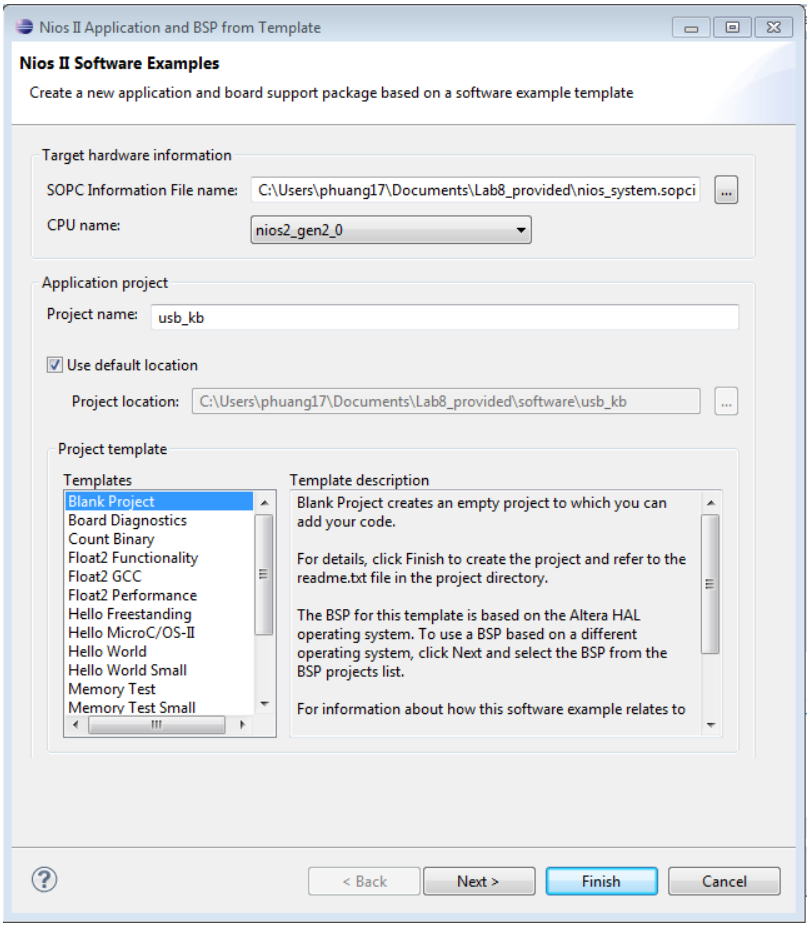


Figure 3

Copy all the files in *software_codes/* to *software/usb_kb/*. In Project Explorer, right click on usb_kb project and click on **Refresh**. Eclipse should detect all the provided .h and .c files located in *software/usb_kb/* automatically.

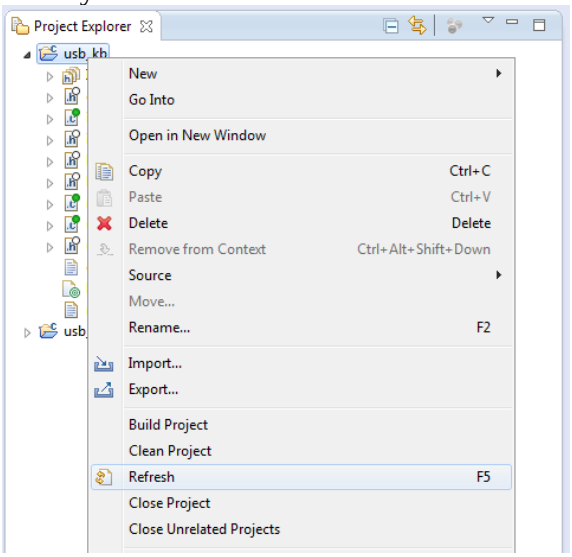


Figure 4

Now we have two projects, *usb_kb*, and *usb_kb_bsp* in the Project Explorer. Right click on *usb_kb_bsp* and select ***Nios II > Generate BSP.*** This configures the compilation environment to be compatible with the hardware design. Click on the main project, usb_kb, and then go to ***Project > Build All*** to compile the program.

IMPORTANT:
Whenever the hardware part is changed, it needs to be compiled in Quartus II and programmed on the FPGA. If the programmer fails to load the .sof file on the FPGA, it's likely because the software is occupying the USB Blaster port. Simply stop the program or restart the FPGA board if this is the case.

On the software side, make sure to right click on <*project_name_bsp*> and select ***Nios II > Generate BSP*** so the latest hardware information is included in the Makefile. Then clean and compile the program again (***Build All***). Compatibility errors occur if you fail to do so!

To run the program on Nios II, click on ***Run*** > ***Run Configurations***… to open up a dialog window, as shown in Figure 5.

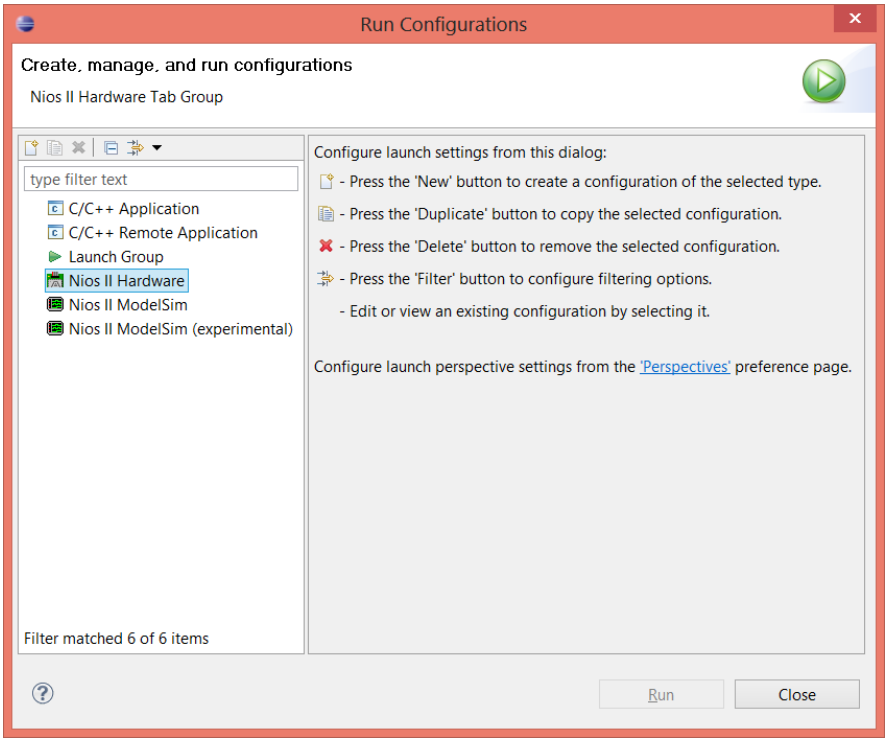


Figure 5

Right click on ***Nios II Hardware*** and select ***New***. Type in the name of the configuration as *DE2-115*. In the ***Project*** tab, select Project name to be the project you are working with. The

corresponding ELF file should automatically be set up. See Figure 6. The ELF file is the compiled software file that is downloaded into the system memory and run on Nios II.

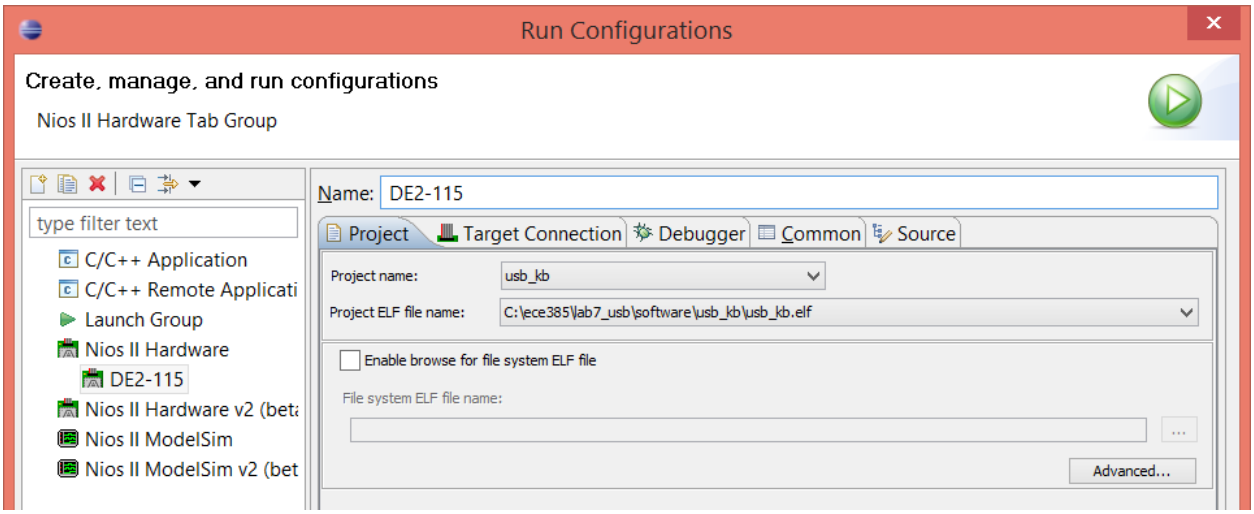


Figure 6

Go to the ***Target Connection*** tab. Click on Refresh Connections on the right. Make sure the *Processors* and the *Byte Stream Devices* are both *USB-Blaster on localhost*. If an error message "*Connected system ID hash not found on target at expected base address*" appears, this means that there is a mismatch between the hardware (FPGA configuration) and the software (BSP). Most likely, this means you did not regenerate the BSP. Ensure that Download ELF and Start processor are both checked.

Figure 7

Go to the ***Common*** tab. Check both ***Allocate console*** and ***File*** under *Standard Input and Output*. Type in a path for your log file, such as *C:\ece385\lab8_usb\mylog.txt*, as shown in Figure 8. This will log some of the error/warning messages in case you encounter them, which helps a lot in troubleshooting.

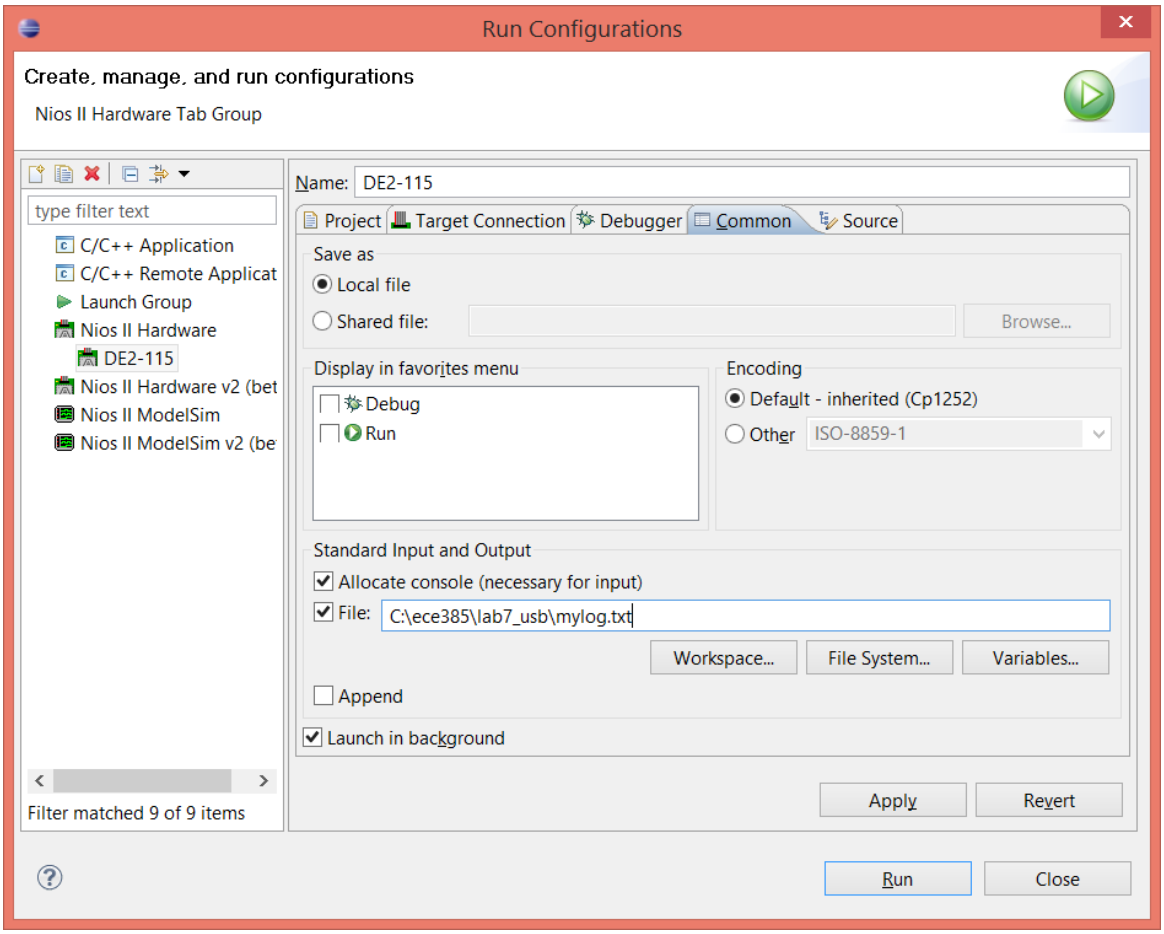


Figure 8

Click ***Run*** to run the program. A Nios II console should appear on the bottom of the Eclipse environment. Standard I/O is made on the Nios II console on your PC, transmitted via the USB Blaster cable to the DE2-115 board and to the program running on the Nios II processor.

USB Host Programming with the EZ-OTG (CY7C67200) Chip

The Cypress EZ-OTG (CY7C67200) chip handles the USB protocol. OTG stands for On-The-Go, which is an add-on specification on top of the standard USB 2.0 standard that allows not only PCs but also other mobile devices to act as a USB Host. The EZ-OTG chip itself contains a RISC microprocessor (CY16), RAM, and ROM (BIOS), as shown on the left in Figure 9. The RAM can be accessed through direct memory access (DMA) at addresses 0x0000 – 0x3FFF. The Serial Interface Engines (SIEs) are the front end of the USB controller and are where the USB ports are attached. The connections between EZ-OTG and the DE2-115 FPGA board are made through the Host Port Interface (HPI), which are shown as the connections in the center of Figure 9. The noteworthy pins are HPI_D[15:0], which is the 16-bit parallel data bus; HPI_INT, which indicates the event of interrupt; HPI_A[1:0], which indicates the addresses to four port registers

on the chip. Each of the port registers control some important functionalities on EZ-OTG, some of which will be used later. The port registers and the addresses are listed in Table 1.

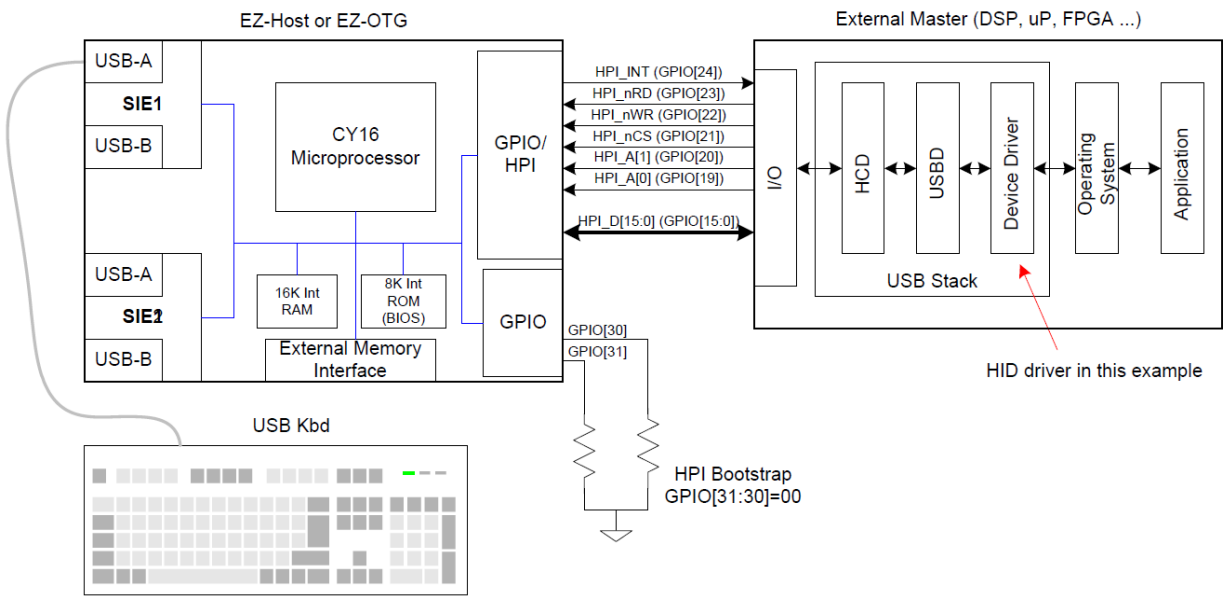


Figure 9

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Table 1

We need to program and configure EZ-OTG in order to make it act as a Host Controller and establishes a connection with the USB keyboard. In the program, we will do the following:

- Set up the EZ-OTG chip.
- Detect connection and assign an address to the connected device.
- Identify device type from descriptors.
- Set configuration.
- Poll keyboard data.

In the Nios II system, most of the peripheral IPs are memory-mapped, that is, we are able to read and write from these IPs via memory access functions. Two important functions (provided by Altera, defined in *HAL/inc/io.h*) that read and write from the peripherals are introduced here:

`IORD(base, offset):`
Read and return data from the memory location specified by (base address + offset). Offset is word-wise, that is, an offset of 1 is equivalent to a 32-bit or 4-byte offset in the address.
`IOWR(base, offset, data):`

Write data to the memory location specified by (base address + offset).

Hint: In Eclipse, right click on a function/variable name and choose Open Declaration will lead you to the function/variable. This is very helpful when you need to trace your source code in multiple files.

However in this lab we will **not** using these functions (IORD and IOWR), since the EZ-OTG's memory space is **not** memory mapped to the NIOSs II address space. Instead you will write the following helper functions which the provided C code will call to talk to the EZ-OTG.

`IO_read(address)`:
Read and return data from the memory location specified by the address. The address should be 2 bits wide as described in the CY7C67200 datasheet.

`IO_write(address, data)`:
Write data to the memory location specified by the address. The address should be 2 bits wide while the data should be 16 bits wide as described in the CY7C67200 datasheet.

By using these functions, we can communicate with the EZ-OTG. The standard procedure of reading is:

1. Write the address to access to HPI_ADDR.
2. EZ-OTG will fetch the data from the specified address (e.g. an address in the RAM) and make it ready to be transferred via HPI_DATA.
3. Read from HPI_DATA.
4. (Optional: continuous read) EZ-OTG will load the data at the next available address to HPI_DATA, that is, if more data are to be read from the next address, we can simply read again from HPI_DATA without giving the next address.

Similarly, the procedure of writing is:

1. Write the address to access to HPI_ADDR.
2. Write data to HPI_DATA in 16-bit little endian words.
3. EZ-OTG will store the data to the specified address (e.g. an address in the RAM).
4. (Optional: continuous write) If more data are to be written into the next address, we can simply write again to HPI_DATA. EZ-OTG will store the data into the next available address.

To further simplify the procedure, you will need to complete the definition of two helper functions in USB.c as follows:

```
alt_u16 UsbRead(alt_u16 Address)

void UsbWrite(alt_u16 Address, alt_u16 Data)
```

These functions perform a single read/write without utilizing the continuous read/write feature. Here, alt_u16 is Altera's built-in data type of a 16-bit unsigned integer. Therefore here the address is 16 bits wide and so is the data.

USB Keyboard Enumeration

The keyboard enumeration procedure is done in *main.c*, following the **Example Data Transfer to Enumerate a USB** section in the **Cypress Using HPI in Coprocessor Mode with OTG-Host (AN6010)** document. The steps are summarized as follows:

Step 1a: Initialize EZ-Host/EZ-OTG registers.
Step 1b: Configure SIE1 as a host.
Step 2: USB_RESET.
Step 3: Set address.
Step 4: Get device descriptor.
Step 5: Get configuration descriptor (1).
Step 6: Get configuration descriptor (2).
(Get device info.)
Step 7: Set configuration.
(Make class requests.)
Step 8: Get HID descriptor (Class 0x21).
Step 9: Get report descriptor (Class 0x22).
And finally poll keyboard data.

Each step requires a series of reads and writes on the EZ-OTG registers and RAM. In the end of Steps 1-2, COMM_EXEC_INT (0xCE01) is written to HPI_MAILBOX to issue an execution, as shown in Figure 6 of the **AN6010** document. In the rest of the steps, a list of Transaction Descriptors (TDs) is written into the RAM, and then UsbWrite(HUSB_SIE1_pCurrentTDPtr, Address) is called to specify the start address of the TDs in the RAM so EZ-OTG can process the TDs. After the TDs are read and processed, EZ-OTG will set the SIE1msg bit of the HPI_STATUS register to 1 to indicate successful execution. The bit fields in HPI_STATUS are explained in Table 4 of the **AN6010** document. If the SIE1msg bit is not set to 1, the program should make another attempt to transfer the TDs and have EZ-OTG process them again.

The Keyboard Enumeration has been completed for you in this lab and you don't need to worry about it. One thing to keep in mind is that the data bus to the EZ-OTG chip is only 16 bits wide. That means with this setup you can only get information about two keys on the keyboard simultaneously. To add the ability to get more information about more keys you need to add another UsbRead call in main.c.

Below is a table with a description of how keycode correspond to the key pressed.

Code that performs this is given as part of the C code for this lab you, this description is for your reference and to assist any debugging.

0	1	2	3	4	5	6	7	8	9	10	11
-	err	err	err	A	B	C	D	E	F	G	H
12	13	14	15	16	17	18	19	20	21	22	23
I	J	K	L	M	N	O	P	Q	R	S	T
24	25	26	27	28	29	30	31	32	33	34	35
U	V	W	X	Y	Z	1	2	3	4	5	6
36	37	38	39	40	41	42	43	44	45	46	47
7	8	9	0	Enter	Esc	BSp	Tab	Space	- / _	= / +	[/ {
48	49	50	51	52	53	54	55	56	57	58	59
] / }	\ / |	...	; / :	' / "	` / ~	, / <	. / >	/ / ?	Caps Lock	F1	F2
60	61	62	63	64	65	66	67	68	69	70	71
F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	PrtScr	Scroll Lock
72	73	74	75	76	77	78	79	80	81	82	83
Pause	Insert	Home	PgUp	Delete	End	PgDn	Right	Left	Down	Up	Num Lock
84	85	86	87	88	89	90	91	91	91	94	95
KP /	KP *	KP -	KP +	KP Enter	KP 1 / End	KP 2 / Down	KP 3 / PgDn	KP 4 / Left	KP 5	KP 6 / Right	KP 7 / Home
96	97	98	99	100	101	102	103	104	105	106	107
KP 8 / Up	KP 9 / PgUp	KP 0 / Ins	KP . / Del	...	Applic	Power	KP =	F13	F14	F15	F16
108	109	110	111	112	113	114	115	116	117	118	119
F17	F18	F19	F20	F21	F22	F23	F24	Execute	Help	Menu	Select
120	121	122	123	124	125	126	127	128	129	130	131
Stop	Again	Undo	Cut	Copy	Paste	Find	Mute	Volume Up	Volume Down	Locking Caps Lock	Locking Num Lock
132	133	134	135	136	137	138	139	140	141	142	143
Locking Scroll Lock	KP ,	KP =	Internat	Internat	Internat	Internat	Internat	Internat	Internat	Internat	Internat
144	145	146	147	148	149	150	151	152	153	154	155
LANG	LANG	LANG	LANG	LANG	LANG	LANG	LANG	LANG	Alt Erase	SysRq	Cancel
156	157	158	159	160	161	162	163	164	165	166	167
Clear	Prior	Return	Separ	Out	Oper	Clear / Again	CrSel / Props	ExSel			
224	225	226	227	228	229	230	231				
LCtrl	LShift	LAlt	LGUI	RCtrl	RShift	RAlt	RGUI				

Table 3

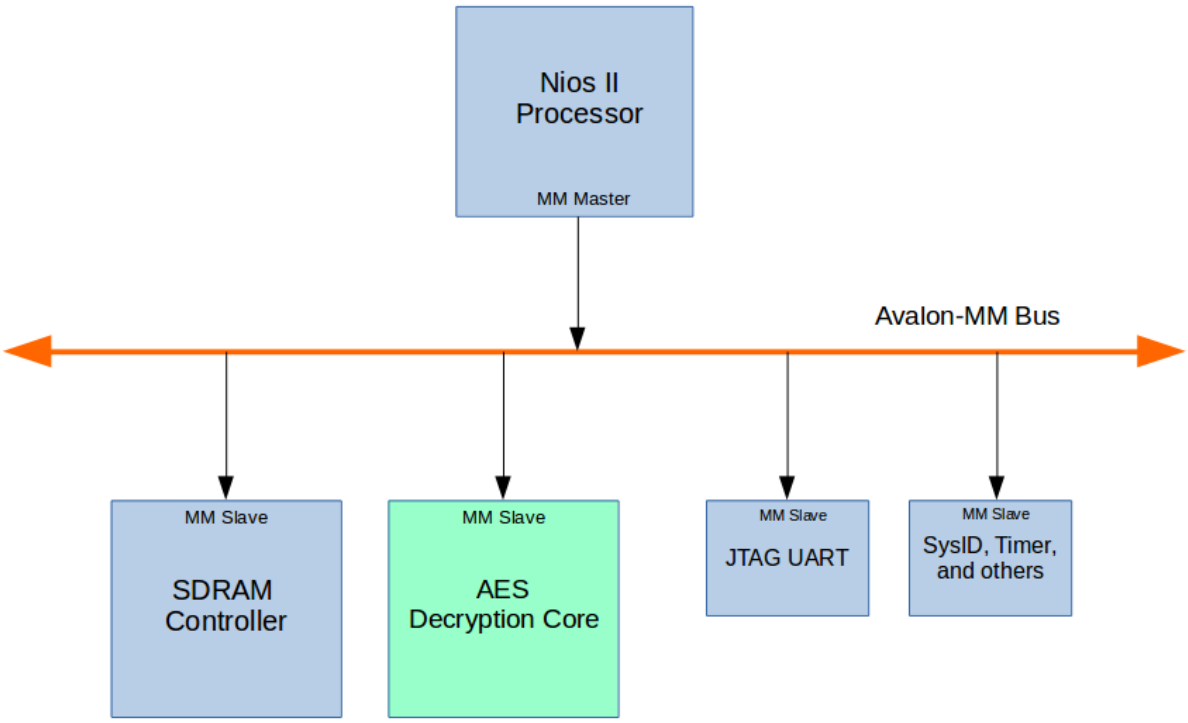
INTRODUCTION TO
THE AVALON-MM INTERFACE AND THE
ADVANCED ENCRYPTION STANDARD

ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to Avalon-MM Interface

Please read this guide carefully and thoroughly as minor mistakes can impact the functionality of your entire project.

System Overview

In this lab, we want to create and add an AES decryption IP core on the Avalon-MM (Memory Mapped) interconnect. Nios II will perform encryption in software and communicate with that IP to perform decryption in hardware.



Your custom IP core will have an Avalon-MM Slave port that allows Avalon-MM Masters like Nios II to directly access with read/write operations. To perform decryption, Nios II will write the 128-bit Encrypted Message and Key to the AES Decryption Core, then write a start signal to one of its registers, and wait until decryption is complete in hardware to read back the 128-bit Decrypted Message.

To begin, start with your Platform Designer setup from Lab 8 with the usual components like Nios II, SDRAM, and JTAG UART, and remove the unnecessary USB components. In the next section, you will design your own AES Decryption Core component and add it to Platform Designer.

The AES Decryption Core Interface

The interface module *avalon_aes_interface.sv* will be the top-level file for the AES Decryption Core component on Platform Designer (note that *lab9_top.sv* is still the top-level file for the entire project). We have provided the input/output signals declaration for you. There is a clock input (CLK), an active-high reset input (RESET), an exported conduit signal which is just a 32-bit output (EXPORT_DATA), and finally an Avalon-MM Slave port which contains a variety of signals whose specifications will be described below.

The Avalon-MM Slave port will complete read and write operations requested by its Master, the Nios II processor (read/write operations correspond to Load/Store instructions in Nios). While the Avalon specifications provide many signals to use for its interface, we only need to use 7 signals to implement the Slave port for this lab.

Avalon-MM Slave Port Interface Signals:

Name	Direction	Width	Description
read	Input	1	High when a read operation is to be performed
write	Input	1	High when a write operation is to be performed
readdata	Output	32	32-bit data to be read
writedata	Input	32	32-bit data to be written
address	Input	4	Address of the read or write operation
byteenable	Input	4	4-bit active high signal to identify which byte(s) are being written
chipselect	Input	1	High during a read or write operation

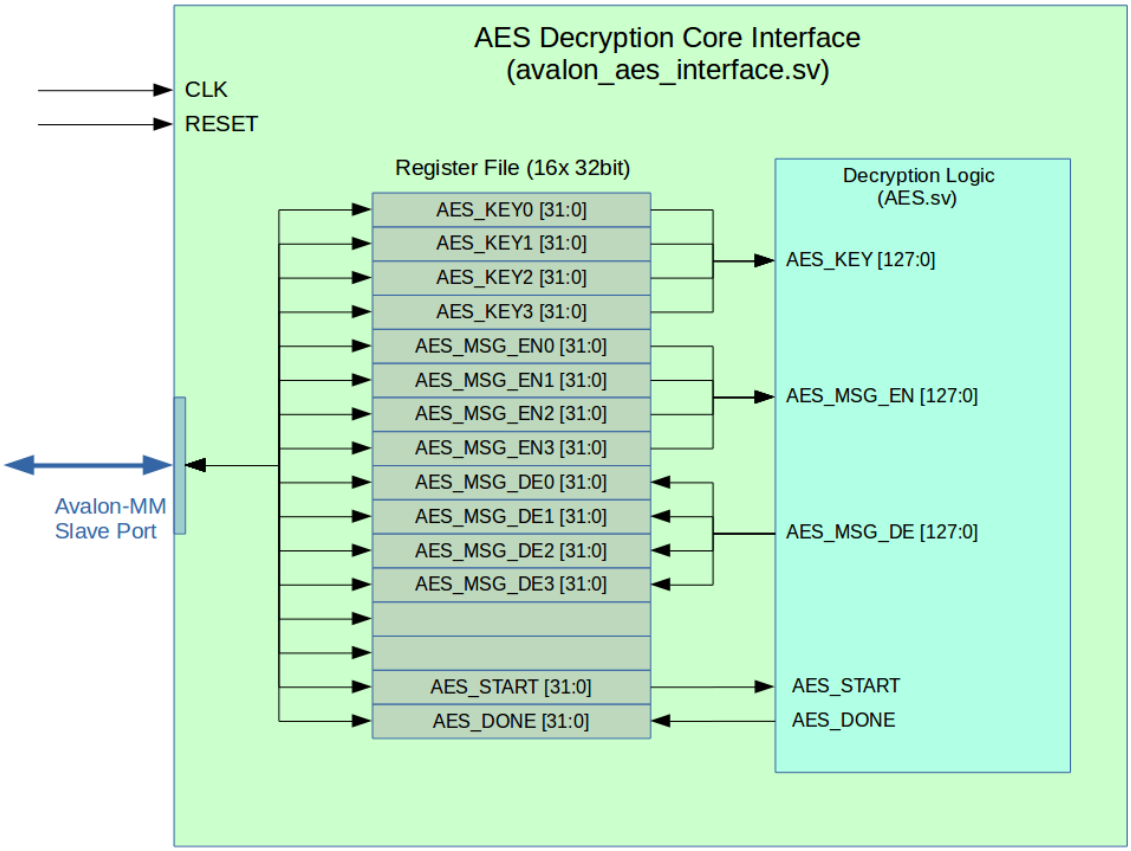
Note that the data width of 32-bit and address width of 4-bit are chosen for this lab, they may be up to 1024-bit and 64-bit respectively. We are using 32-bit readdata/writedata signals because they match the data width of Nios II, which is a 32-bit processor. As for the 4-bit address, which gives $2^4 = 16$ locations, we'll see why that is enough for our purposes.

Now it is up to you to implement the body of this module that completes the incoming read and write requests. Internally, you should create 16 registers, each 32-bit, that hold the values being read and written. There are some requirements that your design must satisfy:

- Read has a 0 cycle wait latency. In other words, when read is high, readdata should have the value of the addressed register on the same cycle.
- Write has a 0 cycle wait latency. In other words, when write is high, the addressed register should be updated with writedata on the next cycle.
- Byte enable determines the bytes being written according to the table below

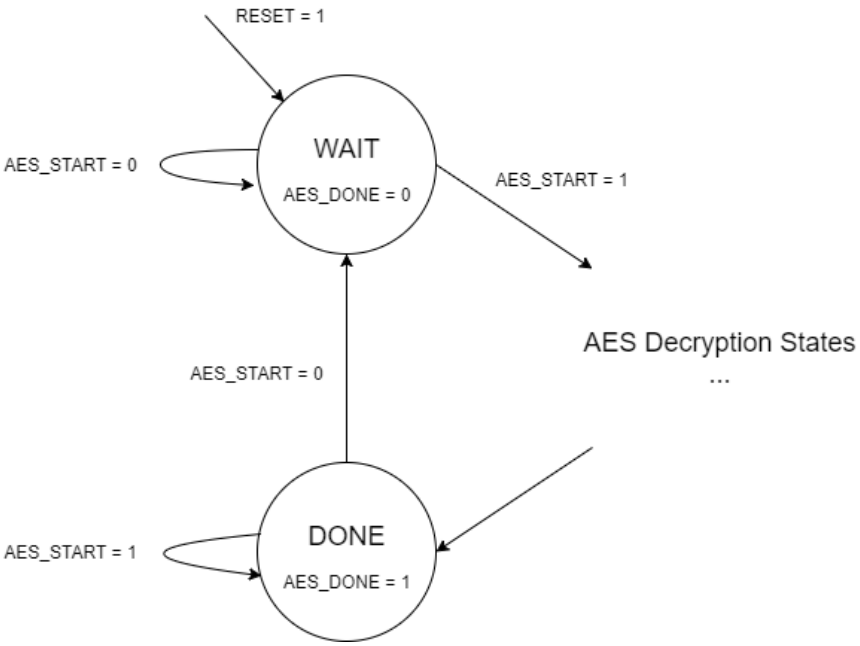
byteenable[3:0]	**Write Action**
1111	Write full 32-bits
1100	Write the two upper bytes
0011	Write the two lower bytes
1000	Write byte 3 only
0100	Write byte 2 only
0010	Write byte 1 only
0001	Write byte 0 only

NIOS II will send the 128-bit AES Key and Encrypted Message as 4 writes ($4 \times 32\ bit = 128\ bit$) each, thus we need 4 registers to hold each of the AES Key, Encrypted Message, and Decrypted Message, yielding a total of 12 registers. We need 2 more registers to hold the START and DONE signals that NIOS II will use to control the hardware state, this brings the total to 14. Since address ranges are in power of 2, we will use 14 of the 16 addressable registers, the remaining 2 simply won't be used. The recommended implementation of *avalon_aes_interface.sv* is shown below.



For week 1, implementing the register array and making sure that you can write your AES Key and Encrypted Message to them will suffice. Keep in mind that EXPORT_DATA should be assigned to the first 2 and last 2 bytes of the Encrypted Message, exported from the Platform Designer design (covered in next section), and displayed on the LEDs with hexdrivers on the lab 9 top level.

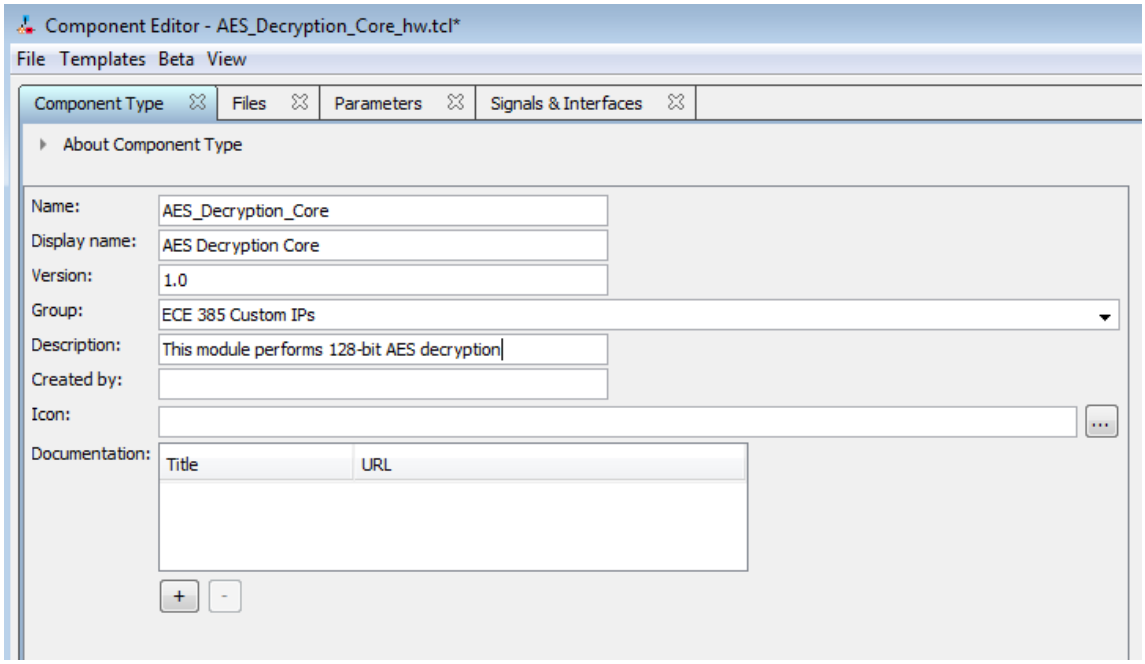
For week 2, instantiate *AES.sv*, the decryption logic with your state machine, and make the appropriate connections by packing or unpacking the 4x 32-bit registers to 128-bit input and output ports for AES Key, Encrypted Message, and Decrypted Message. The START and DONE signals are 1 bit so you can simply use the last bit of Registers 15 and 16 as shown in the figure. Your state machine in *AES.sv* should be able to perform decryption continuously. Create two control states: WAIT and DONE to handle the START input and DONE output signals as described below.



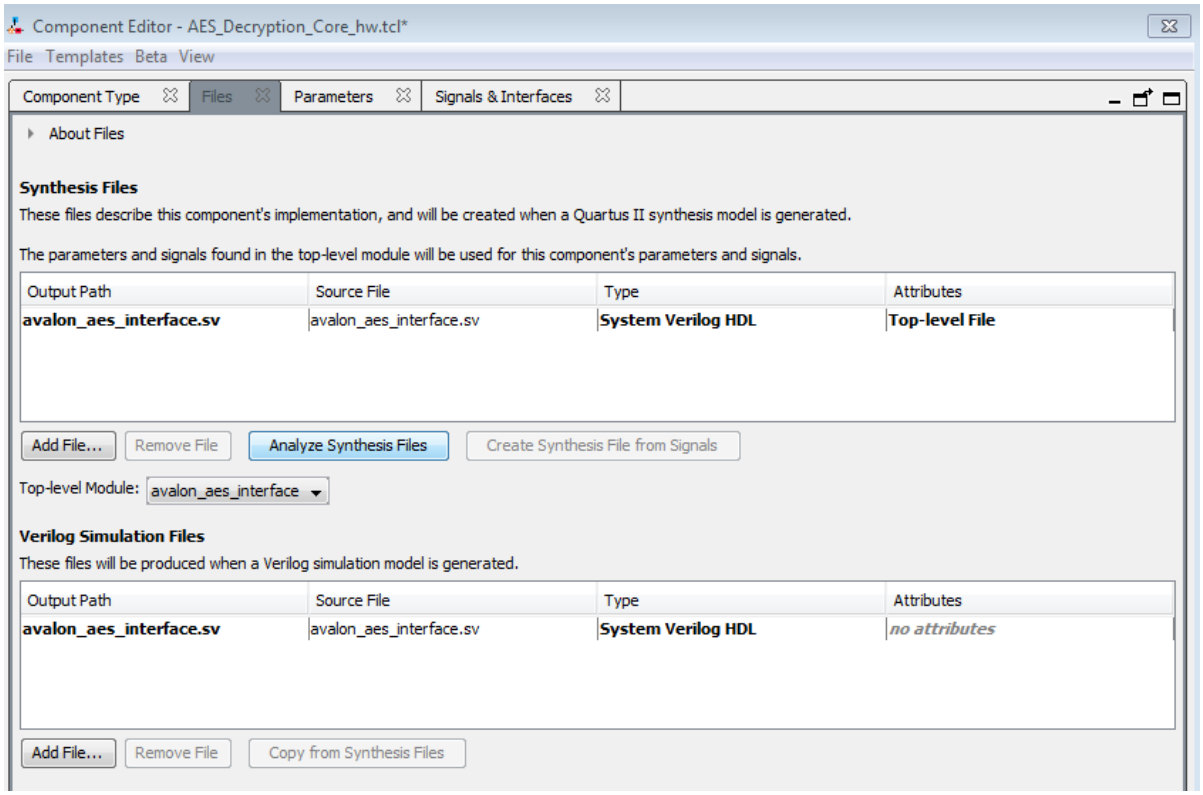
Creating a Platform Designer Component

When you are done implementing the partial or complete interface module *avalon_aes_interface.sv*, add it to the Platform Designer IP catalog with component editor.

1. Launch Platform Designer editor and load your design that already has Nios II, SDRAM, UART, etc.
2. On the upper left **IP Catalog** panel, double click ***New Component…***

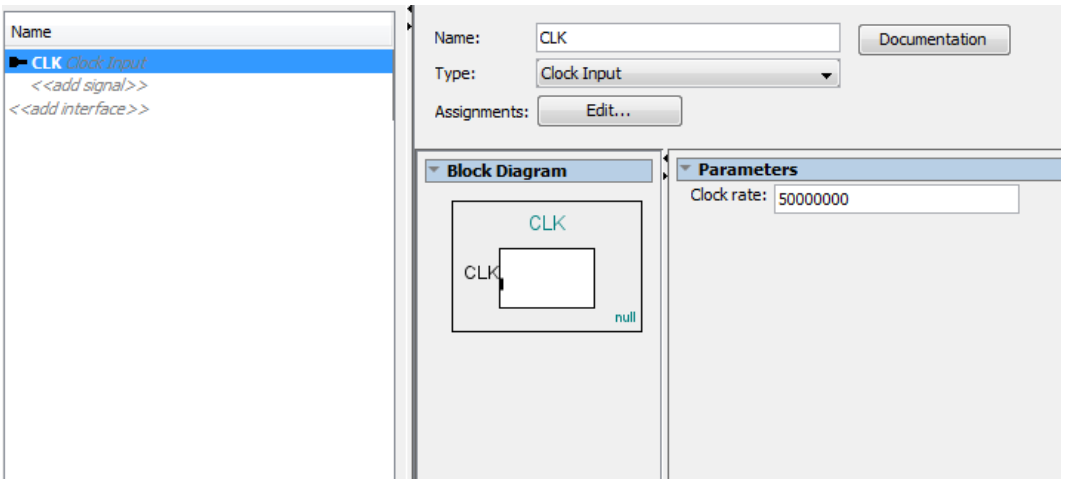


3. Enter the Name and Display name for your component. Display name is the one shown in Platform Designer IP Catalog. It is good practice to also keep track of the version of your IP, increment it when you make changes or fix issues. You can also categorize it to an existing or new group, here we make a group called “ECE 385 Custom IPs”. Finally, give it a brief description.
4. Click on the Files tab.



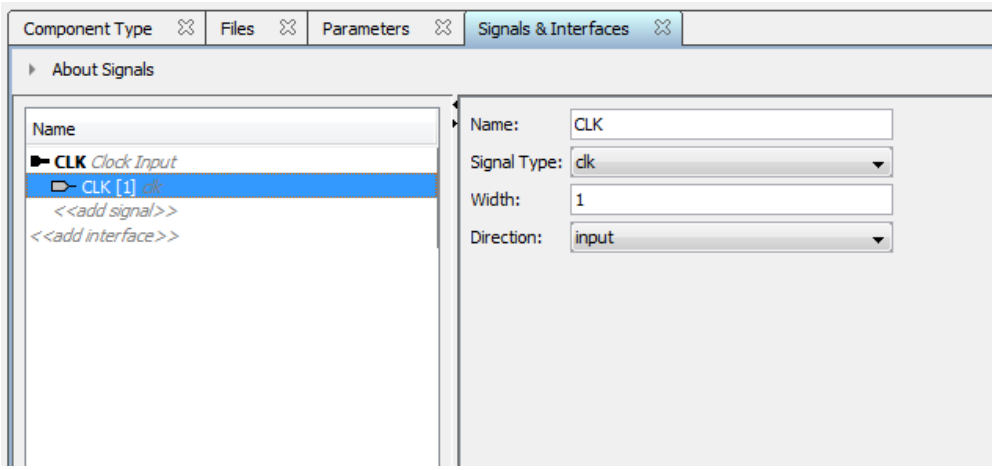
5. Under **Synthesis Files**, click on Add File… and choose *avalon_aes_interface.sv* which is the top-level module for this component. Under **Verilog Simulation Files**, click on **Copy from Synthesis Files**, this is useful if you want to simulate your system in ModelSim.
6. Click on the Signals & Interfaces tab. Now we want to create the Avalon ports and match the Avalon defined interface signals with our input/output declarations in *avalon_aes_interface.sv*.
7. Let’s add the clock input first. Click on *<<add interface>>*.
   - In Quartus 15.0, a default name of “avalon_slave” will appear, replace that with “CLK” and press Enter. The default port type is Avalon Memory Mapped Slave, but we want a clock input, click on the drop-down list for **Type**, and choose **Clock Input**.
   - In Quartus 16.0+, you are asked to choose from a list of port types, click on Clock Input, then rename it to “CLK” for **Name**.

   Finally, enter the **Clock rate** of 50MHz, your screen should look like this in both versions. If you added other interfaces by mistake, you can always right click those interfaces on the left tab and choose remove.



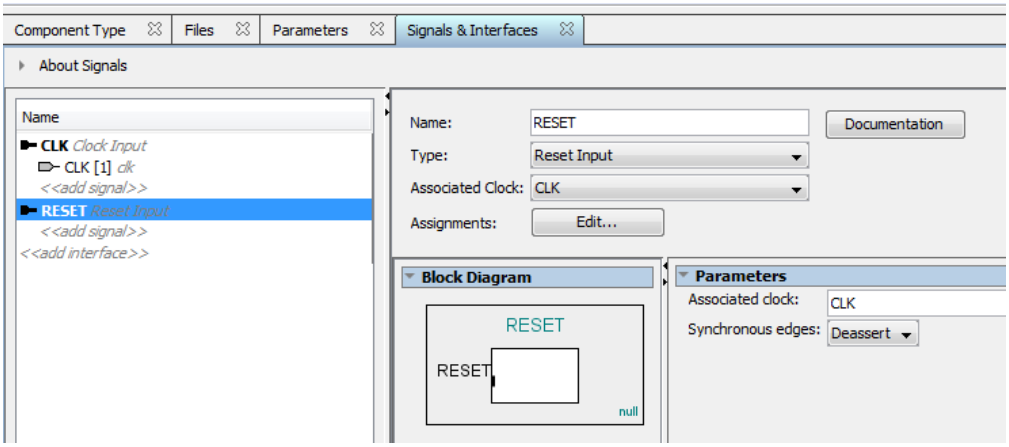
8. We now have a Clock Input interface for our component called CLK, but it's not associated with any signals defined in our top level module yet. Obviously, the clock input defined in *avalon_aes_interface.sv* is "input logic CLK". To make that association, click on *<<add signal>>*.
   - In Quartus 15.0, a default name of "new_signal" will appear, replace that with "CLK" to match the input name declared in *avalon_aes_interface.sv*.
   - In Quartus 16.0+, choose the only option of "clk", then change the Name on the right tab to "CLK" to match the input name declared in *avalon_aes_interface.sv*.

   The remaining default values should be correct, if not, change them to match the picture below.

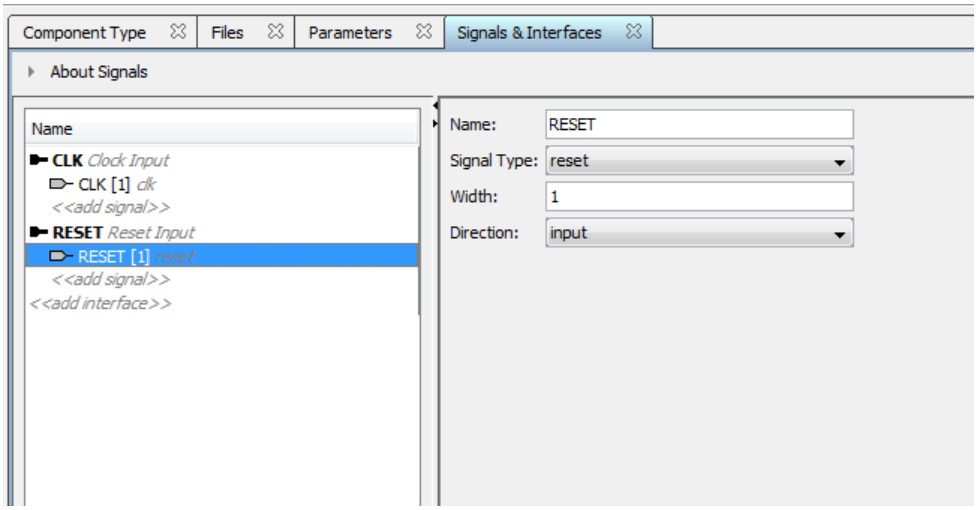


9. The Clock Input interface only needs one signal, so we are done with it. Now let's add the Reset Input interface. Click on *<<add interface>>*.
   - In Quartus 15.0, like the steps above, replace the default name with "RESET", then choose **Reset Input** for **Type**.
   - In Quartus 16.0+, likewise, choose **Reset Input** from the list of port types, then change the name to "RESET".

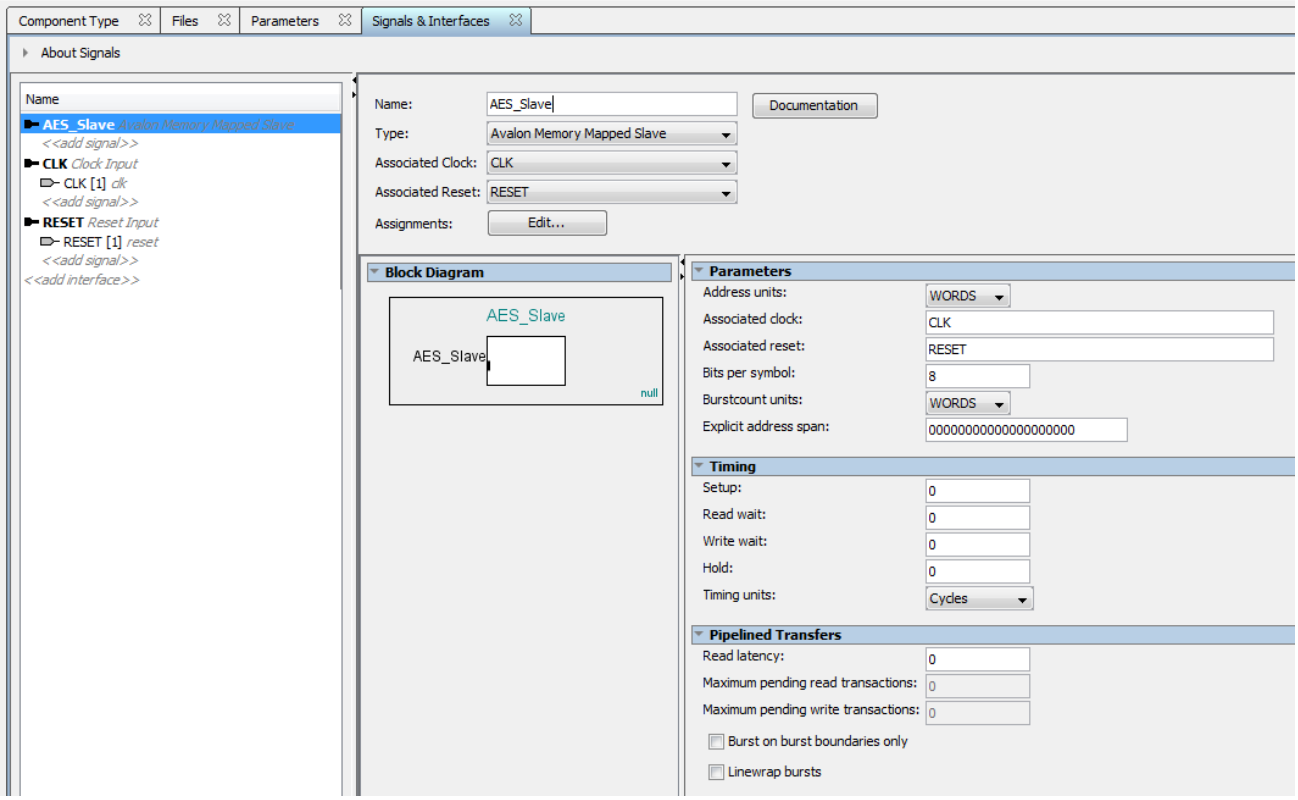
The associated clock should be set to the “CLK” interface we just defined by default, if not, set it.



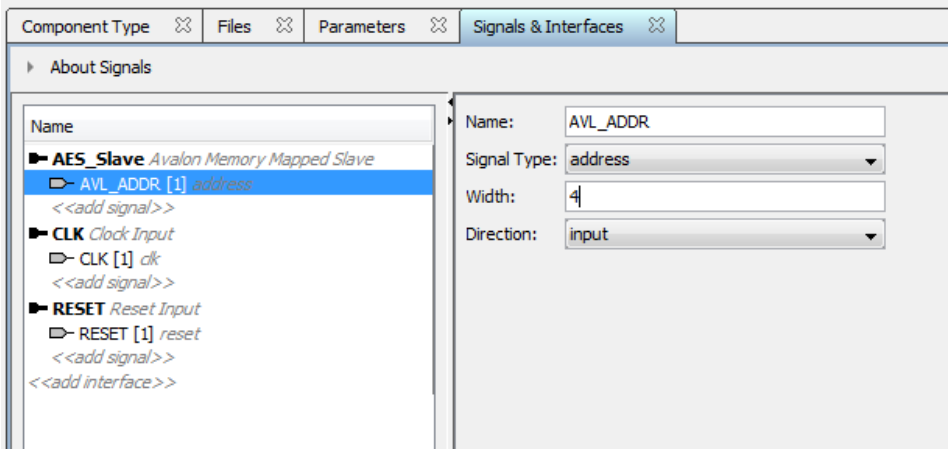
10. Now, add the associated RESET input to this interface. Click on <<*add signal*>>. Follow a similar procedure to Step 7, except use “RESET” for **Name**, and **reset** for **Type**.



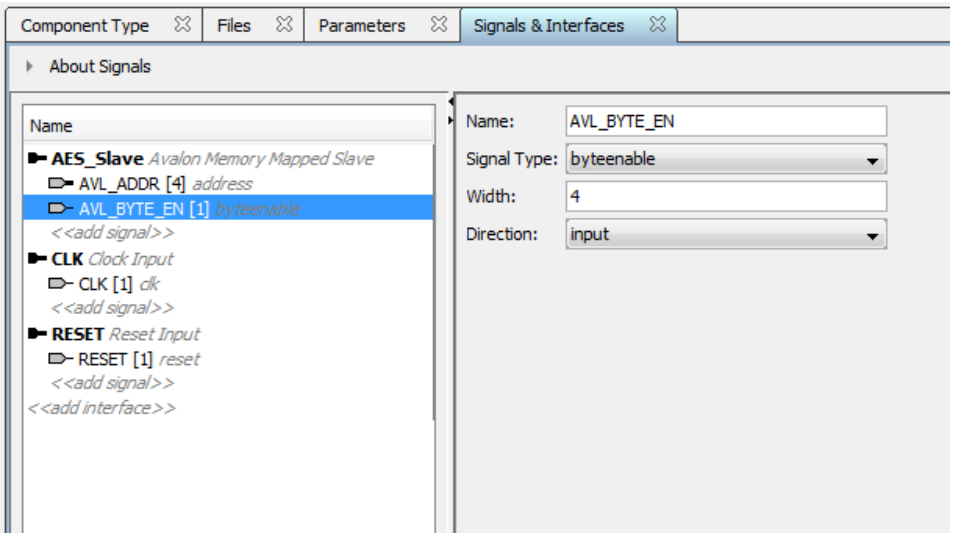
11. Next, add the main Avalon-MM Slave port. Click on <<*add interface*>>. You should be familiar with how to do this now, set the **Name** to “AES_Slave”, **Type** to **Avalon Memory Mapped Slave**. Also set the **Associated Clock** and **Reset** to the CLK and RESET ports we defined earlier.



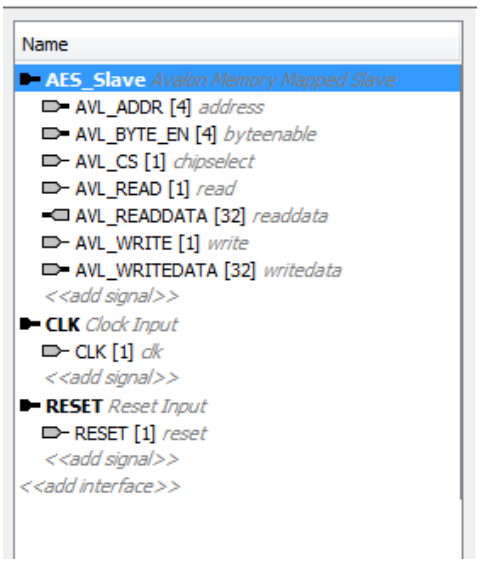
12. Under the **Parameters** section, check that you have identical settings as the picture above. Note that we make this Memory Mapped Slave byte addressable (8-bit per symbol) and the address units are in words (32-bit per word), so we expect the address to span a range of $4 \times 2^4 = 64$ (0x00 to 0x3F), making it no different than accessing no regular memory. In C, we can access this range directly as an array of 16 elements.
13. Under the **Timing** section, set both Read wait and Write wait to 0, as previously described, we want reads and writes to complete in the same cycle.
14. Now, we begin to add the relevant signals for this Memory Mapped slave port. Unlike previous ports, the MM Slave has multiple signals (read, write, chipselect, etc.) Let's start with the **address** signal, as before, click on <<*add signal*>> and set the **Name** to "AVL_ADDR", the **Signal Type** to **address**, **Width** to 4, and **Direction** to **input** in order to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_ADDR").



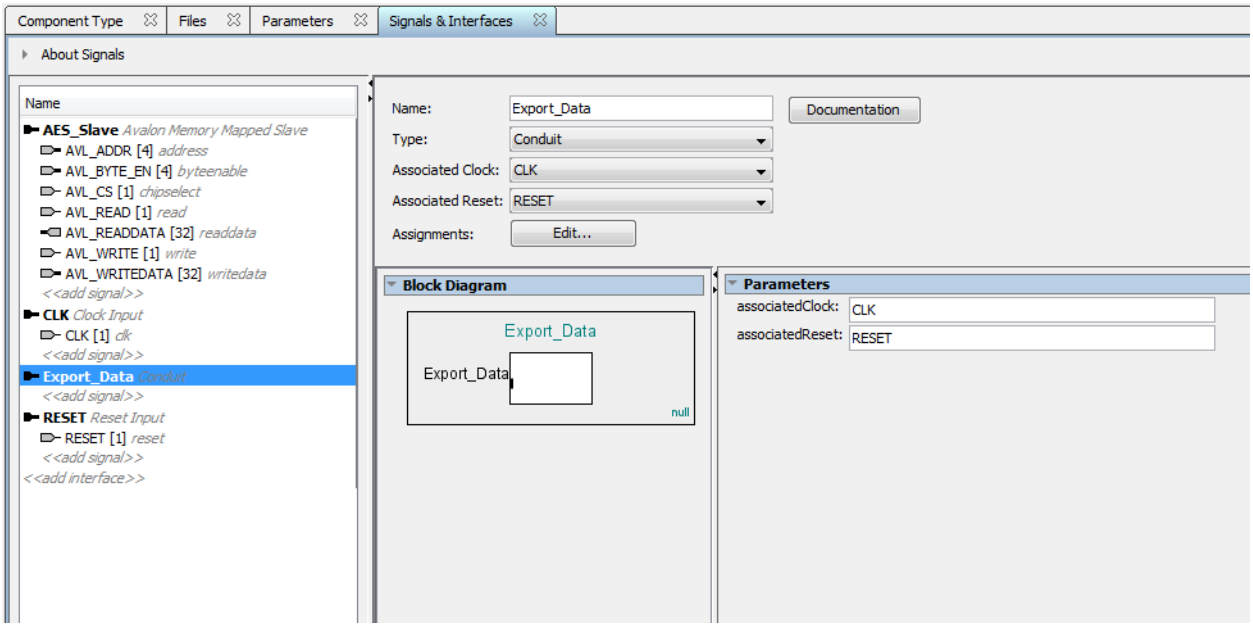
15. Add the **byteenable** signals next, click on <<*add signal*>> and set the **Name** to "AVL_BYTE_EN", the **Signal Type** to **address**, **Width** to 4, and **Direction** to **input** once again to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_BYTE_EN").



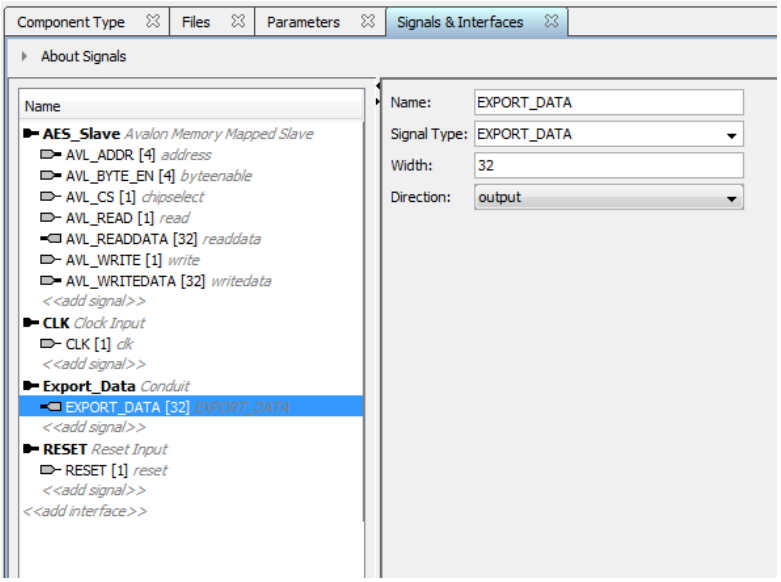
16. Complete the previous step the remaining signals for AES_Slave. Match the names, signal types, width, and direction for each signal. Be careful that **readdata** (AVL_READDATA) is an output unlike the others. When done, it should look like this:



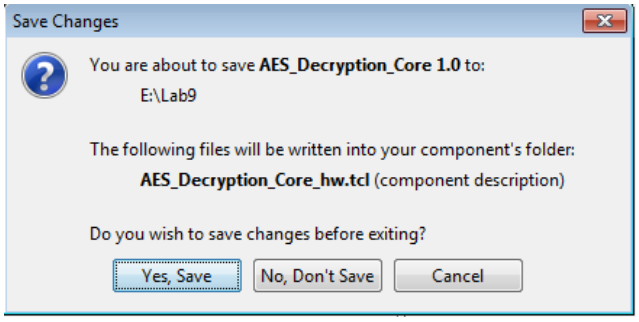
17. We are done with the AES_Slave. The last port to add is the exported conduit (EXPORT_DATA) to output part of the registers to the LEDs on the top level. Click on <<*add interface*>> and choose the settings as follows:



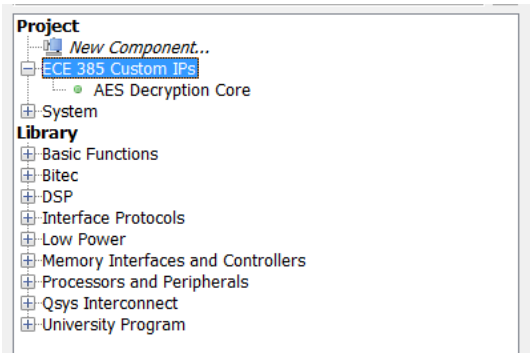
18. Add the only signal for this conduit, namely, the 32-bit EXPORT_DATA output signal. Set **Name** to EXPORT_DATA, **Signal Type** to * (it will be changed automatically or may appear as “new_signal” depending on your Quartus version), **Width** to 32, and **Direction** to output as shown below.



19. Check that there are no errors in the Message tab at the bottom. If there are errors like "[*port name*] Interface must have an associate clock/reset", click on that port on the left tab, and set the Associated Clock or Reset to CLK and RESET respectively if they have been reset to none (this is a minor bug that happens in some versions of Quartus). If there are no errors, click on *Finish…* at the bottom right and click "Yes, Save" to save the generated TCL script.

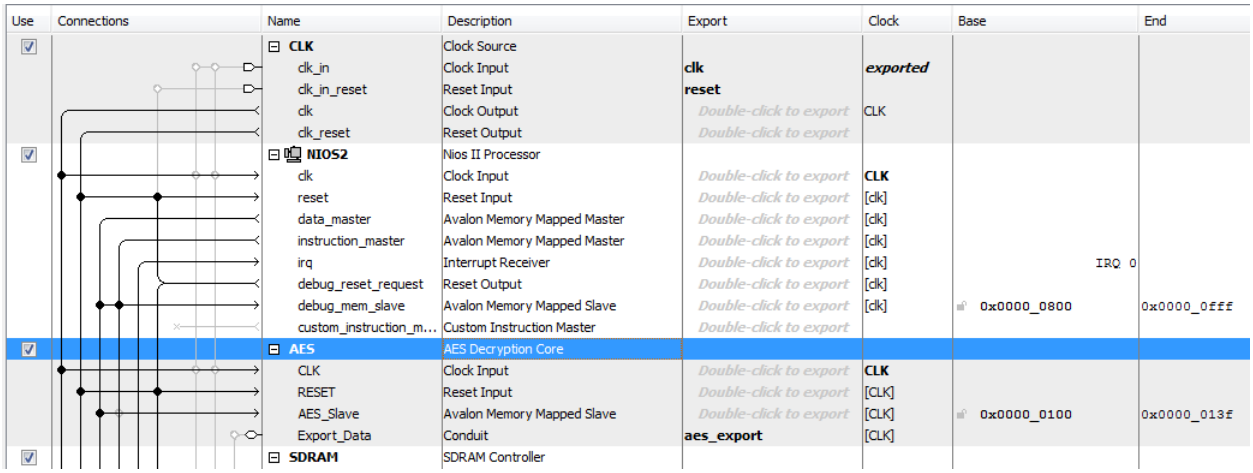


20. Now, in Platform Designer, your newly created AES Decryption Core component should appear in IP Catalog, grouped under "ECE 385 Custom IPs". It can now be added to Platform Designer like other Altera provided IPs. If you made a mistake, you can right-click it and click Edit to make any changes needed such as increasing the version number.



Finalizing your Platform Designer Design

Add your newly created component AES Decryption Core to Platform Designer by double clicking it. We did not define any parameters, so just click Finish to add it to Platform Designer, rename it if you want to. Make the appropriate **CLK** and **RESET** connections, and most importantly, connect the **AES_Slave** to Nios II's **data_master** to allow Nios to access its registers through reads and writes. We set its base address to 0x100 by default, you can choose a different address if it conflicts with your other components, and be sure to change it in software as well. Export the Export_Data conduit as "aes_export" and assign that signal to your HexDrivers on your top level file for this lab (*lab9_top.sv*).



Finally, generate the HDL for your Platform Designer design and include the QIP in your project.

IMPORTANT: Whenever you change the SystemVerilog code in *avalon_aes_interface.sv* to fix bugs or add things after this, you need to use Platform Designer to regenerate the HDL so that those changes take effect. (The actual file being compiled by Quartus is the one generated by Platform Designer located in lab9\lab9_soc\synthesis\submodules\avalon_aes_interface.sv)

IMPORTANT: There is a bug in some versions of Quartus including 18.1 that causes it to misname your new Platform Designer component. If Platform Designer' generated top level Verilog file instantiates "new_component" for your avalon_aes_interface module, close Platform Designer, go to your project folder and edit the file "AES_Decryption_Core_hw.tcl": under "# file sets" section, replace them with the lines below. Then, open Platform Designer and update the version of your component (right click it under IP Catalog > Edit... and change version from 1.0 to 1.1 or any bigger number, then click Finish... to save), save your Platform Designer system and regenerate your HDL. Alternatively, you can correct the name in the Verilog file from "new_component" to "avalon_aes_interface" each time after you generate HDL in Platform Designer.

```
#
# file sets
#
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL avalon_aes_interface
set_fileset_property QUARTUS_SYNTH ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property QUARTUS_SYNTH ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file avalon_aes_interface.sv SYSTEM_VERILOG PATH
avalon_aes_interface.sv TOP_LEVEL_FILE


add_fileset SIM_VERILOG SIM_VERILOG "" ""
set_fileset_property SIM_VERILOG TOP_LEVEL avalon_aes_interface
set_fileset_property SIM_VERILOG ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property SIM_VERILOG ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file avalon_aes_interface.sv SYSTEM_VERILOG PATH
avalon_aes_interface.sv
```

Normally, we would use [Analyze Synthesis Files] in Component Editor to automatically match Avalon signal names and generate the correct script however that tool also unreliable and sometimes outputs “no modules found when analyzing null” on valid SystemVerilog files.

Nios Software

Refer to how we defined the parameters for the Avalon-MM slave "AES_Slave" in step 12. To access its registers in a C program, we declare a pointer to the slave's base address.

```
// Pointer to base address of AES module, make sure it matches Platform
Designer

volatile unsigned int * AES_PTR = (unsigned int *) 0x00000100;
```

Since the Nios II E processor has no cache or TLB, accessing the registers is as simple as dereferencing the AES_PTR pointer, for example:

```
// Send the 128-bit Key (Split into 4x 32-bit)

AES_PTR[0] = key[0];

AES_PTR[1] = key[1];

AES_PTR[2] = key[2];

AES_PTR[3] = key[3];
```

To test that your registers are working properly for week 1, you can write a value to it and read it back and check that it matches.

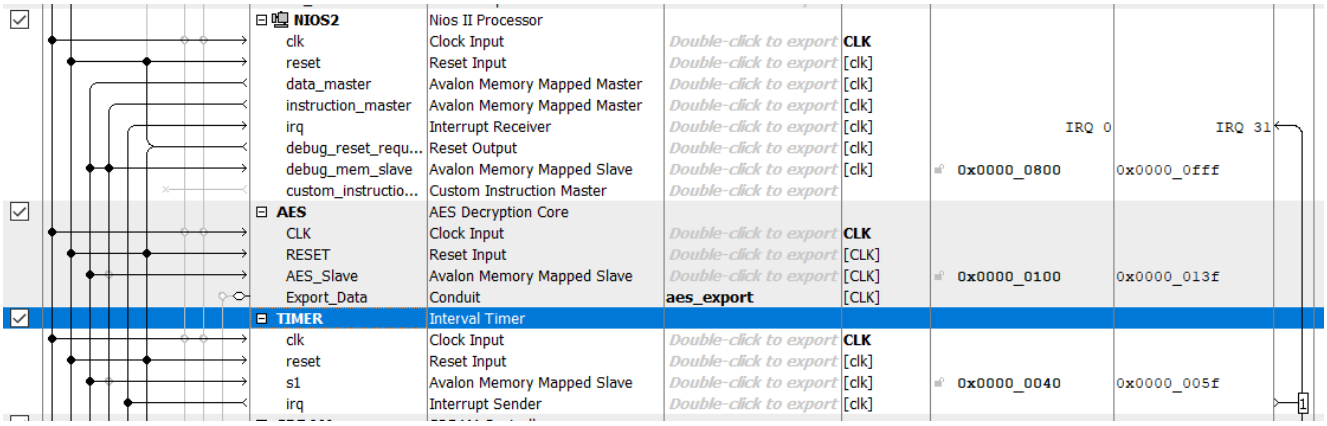
```
AES_PTR[10] = 0xDEADBEEF;

if (AES_PTR[10] != 0xDEADBEEF)

      printf("Error !");
```

For week 2, look at the state machine skeleton on page 4 to determine how you should control your hardware decryption module. Note that after sending the START signal (writing 1 to AES_PTR[14]), you should continuously read the DONE signal (AES_PTR[15]) until it becomes 1 which indicates that the decrypted message is ready, and finally, set the START signal to 0 to allow the state machine to return to WAIT state.

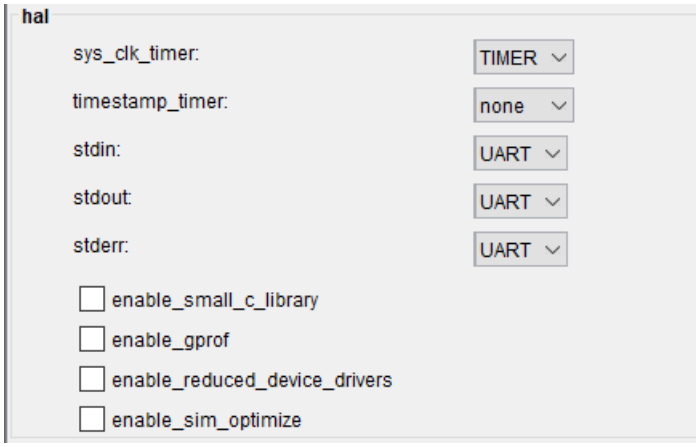
Nios II Benchmarking

When you are done testing the functionality of your code and hardware you can run your program in benchmark mode to get encryption/decryption speed measurements. However, Nios II itself does not have reliable means to measure time so we need to add an Interval Timer in Platform Designer that sends periodic interrupts.

1. Open your Platform Designer Project and add an Interval Timer. It can be found under "Processors and Peripherals" > "Peripherals" > Interval Timer. The default setting of 1ms intervals will suffice for our purposes.



2. Make the appropriate connections for each port, they should be familiar to you by now. Be sure to have the Interrupt Sender connected to Nios on the right side (in the example above, it is IRQ 1). As usual, you need to regenerate HDL and recompile your Quartus project.
3. In your Eclipse project, open BSP editor and choose your newly added interval timer for sys_clk_timer. This will make *time.h* function properly.



4. Generate BSP and recompile your Eclipse project before running.

ECE385
DIGITAL SYSTEMS LABORATORY
Advanced Encryption Standard (AES)

What is the Advanced Encryption Standard (AES):

- A cipher specification for electronic data encryption [1]
- Established by the U.S. National Institute of Standards and Technology in 2001 as Federal Information Processing Standards Publication 197 (FIPS PUB 197) to replace the older Data Encryption Standard (DES)
- A symmetric block cipher based on the Rijndael algorithm

A ***Cipher*** is an encryption process to transform a meaningful message into scrambled data for the purpose of data transfer security. A Cipher generally takes in a message, called a ***Plaintext***, and a secret key, called a ***Cipher Key*** as its inputs. Depending on the specification of the Cipher, the Plaintext and the Cipher Key will go through some kind of algorithm, and produces a scrambled data as the output, which is called a ***Ciphertext***. The ***Inverse Cipher***, the dual of the cipher, reverses the direction of the encryption process. It takes in a Ciphertext and a Cipher Key, goes through the reverse algorithm, and produces the original meaningful Plaintext message.

A block cipher is a deterministic cipher algorithm operating on a fixed number of bits, denoted a block, at a time. If the cipher uses the same set of cipher key for both the encoding and the decoding process, it is called a symmetric-key algorithm. The AES algorithm is a symmetric block cipher based on the Rijndael algorithm, which is originally designed to handle different block sizes and Cipher Key lengths. However the AES standard specifies the data block to be of 128 bits, using Cipher Keys with lengths of 128, 192, and 256 bits. In this tutorial we will introduce the 128-bit Cipher Key version of the AES algorithm in the following sections. This tutorial is adopted from FIPS PUB 197 [1], and it skips the mathematical theory behind the AES algorithm. For the detailed mathematical background, please refer to [1].

Interface and Data Structure:

- **Top-Level Inputs and Outputs**

  Inputs

  Plaintext : logic [127:0] – 128-bit data input to the Cipher or output from the Inverse Cipher. Arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix.

  Cipher Key : logic [127:0] – 128-bit secret, cryptographic key arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix.

  Outputs

  Ciphertext : logic [127:0] – 128-bit data output from the Cipher or input to the Inverse Cipher.

- **Terminalogy**

State : logic [127:0] – 128-bit intermediate results during the AES algorithm. Also is arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix, which can be denoted by four *Words*.

Word : logic [31:0] – A group of 32 bits data within a State that are processed together as a single entity. Each Word is composed of the 4 Byte data from a single State column.

Round Key : logic [127:0] – 128-bit keys derived from the Cipher Key using the Key Expansion routine. It is applied in different stages of the algorithm.

Key Schedule : logic [1407:0] – $128\times(N_r+1)$ -bit keys derived from the Cipher Key using the Key Expansion routine, where $N_r=10$ is the number of looping rounds for the 128-bit AES algorithm. Each of the individual 128-bit Keys is applied to different stages of the algorithm.

AES Encryption Algorithm:

- **AES Encryption Pseudo Code**

Figure 1 shows the pseudo code for the AES encryption algorithm, where $N_b=4$ is the number of columns in the State. Notice that there are only 9 full looping rounds for the 128-bit AES. The last round consists of the same modules as the looping rounds minus the MixColumns() module. Figure 2 shows the encryption algorithm flow.

```
AES(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
      byte state[4,Nb]
      state = in
      AddRoundKey(state, w[0, Nb-1])
      for round = 1 step 1 to Nr-1
          SubBytes(state)
          ShiftRows(state)
          MixColumns(state)
          AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      end for
      SubBytes(state)
      ShiftRows(state)
      AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
      out = state
end
```

Figure 1 Pseudo code for the AES encryption algorithm [1].

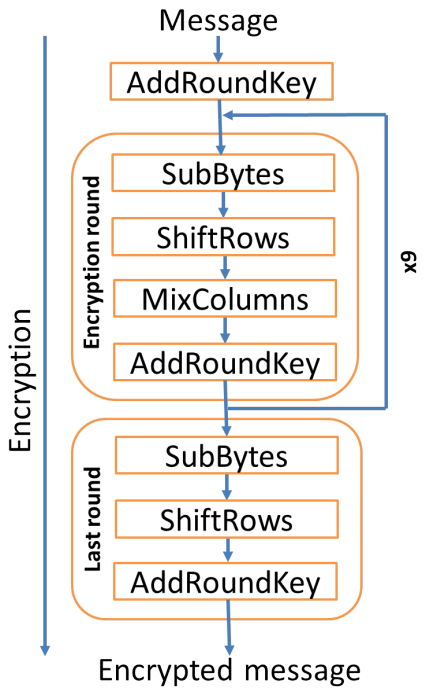


Figure 2 AES encryption algorithm flow.

- **AES modules overview**

The Pseudo code for the AES algorithm in Figure 1 takes *in* (Plaintext) and *w* (expanded Cipher Key, using a separate module **KeyExpansion()**). It then goes through several rounds of modules and computes *out* (Ciphertext). Below describes the purpose of each module.

KeyExpansion () – Takes the Cipher Key and performs a Key Expansion to generate a series of Round Keys (4-Word matrix) and store them into Key Schedule.

AddRoundKey() – A Round Key of 4-Word matrix is applied to the updating State through a simple XOR operation in every round.

SubBytes() – Each Byte of the updating State is non-linearly transformed by taking the multiplicative inverse in Rijndael's finite field (or the Galois field – $GF(2^8)$) then applying an affine transformation. The process is usually simplified into applying a lookup table called the Rijndael S-box (substitution box).

ShiftRows() – Each row in the updating State is shifted by some offsets.

MixColumns() – Each of the four Words in the updating State undergoes separate invertible linear transformations over $GF(2^8)$ such that the four Bytes of each Word are linearly combined to form a new Word.

- **KeyExpansion()**
  Key Expansion generates a Round Key (N_b Words) at a time based on the previous Round Key (use the original Cipher Key to generate the first Round Key), for a total of $N_r + 1$ Round Keys, called the Key Schedule. Each Round Key in the Key Schedule will be used in each looping rounds of the algorithm as the updated Cipher Key.

  The KeyExpansion function consists of three separate steps. Figure 3 shows the pseudo code for the KeyExpansion function. The process is as follows: the first four Words of the Key Schedule is filled with the original Cipher Key as the first Round Key. This first Round Key will be used later in the first AddRoundKey module in the AES algorithm before the 9 looping rounds.

  The next 10 Round Keys are generated based on the immediate previous Round Key. Every Word $w[i]$ is generated by XORing $w[i-1]$ and $w[i-N_k]$, where N_k is the number of 32-bit Words in the Cipher Key. In addition, if i is a multiple of N_k, i.e., for the first Word in every Round Key, the Word has to first go through a **`RotWord()`** function then a **`SubWord()`** function, and then XOR the result with the corresponding Word from the **Rcon** table (Round Constant Word array). The **`RotWord()`** function simply cyclically rotate the current Word from $[a_{0,i} \quad a_{1,i} \quad a_{2,i} \quad a_{3,i}]^T$ into $[a_{1,i} \quad a_{2,i} \quad a_{3,i} \quad a_{0,i}]^T$. The **`SubWord()`** function is identical to the **`SubBytes()`** function, where the individual Bytes are substituted using the S-box lookup table (more explanation is provided in the following `SubBytes()` module section). Table 1 shows the Rcon table, which consists of the values given by $[x^{i-1} \quad \{00\} \quad \{00\} \quad \{00\}]^T$, $i = 1 \sim 10$.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
word temp
   i = 0
   while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
   end while
   i = Nk
   while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
   end while
end
```

Figure 3 Pseudo code for the KeyExpansion module [1].

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Table 1 Rcon lookup table.

- **AddRoundKey()**
  An N_b -Word Round Key is fetched form the pre-computed Key Schedule where each Byte is bitwise XORed with the corresponding Byte from the updating State. Figure 4 shows the AddRoundKey module.

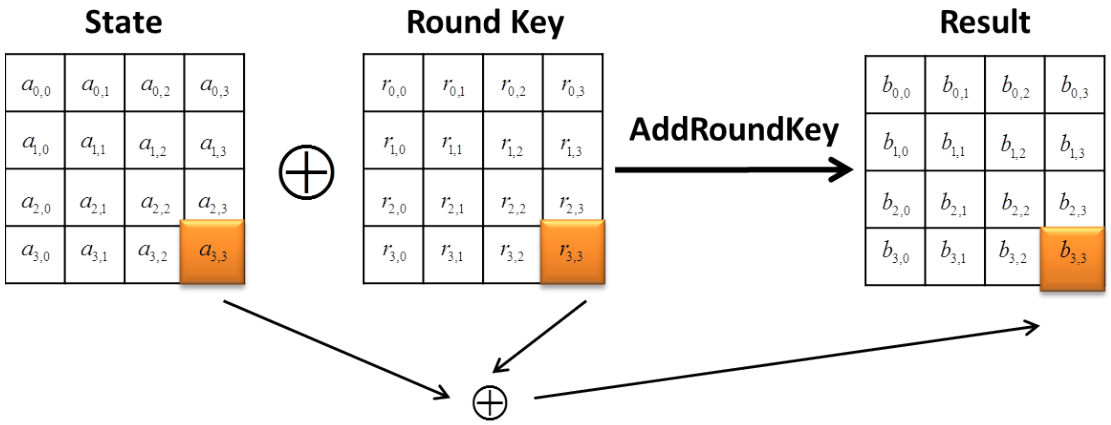


Figure 4 AddRoundKey module.

- **SubBytes()**
  The transformation process for each Byte in the State is pre-calculated and stored into S-box for the encryption process. Table 2 shows the lookup table for the S-box. To apply the transformation, simply loop through each Byte in the State and look up the table using the two hex values of the Byte as the row and column indices of the table to find the transformed value, such that

$$a(i, j) = SBox[a(i, j)],$$

  where $a(i, j)$ is the Byte under transformation, and $SBox$ is stored as a 256 Byte consecutive array instead of a 16×16 matrix. Figure 5 shows the SubBytes module. Table 2 shows the lookup table for S-box.

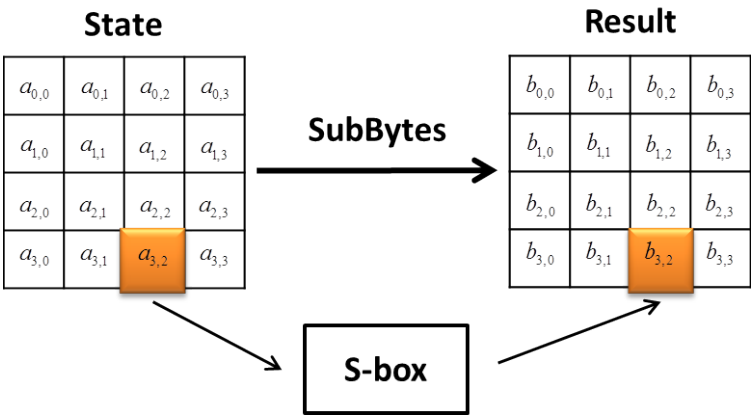


Figure 5 AddRoundKey module.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1x	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2x	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3x	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4x	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5x	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6x	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7x	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8x	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9x	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
ax	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
bx	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
cx	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
dx	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
ex	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
fx	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2 Rijndael S-box.

- **ShiftRows()**
  The rows in the updating State is cyclically shifted by a certain offset. Specifically, row n is left-circularly shifted by $n-1$ Bytes. That is, the first row remains unchanged; the second row is left-circularly by 1 Byte; the third row is left-circularly by 2 Bytes; and the fourth row is left-circularly by 3 Bytes. Figure 6 shows the ShiftRows module.

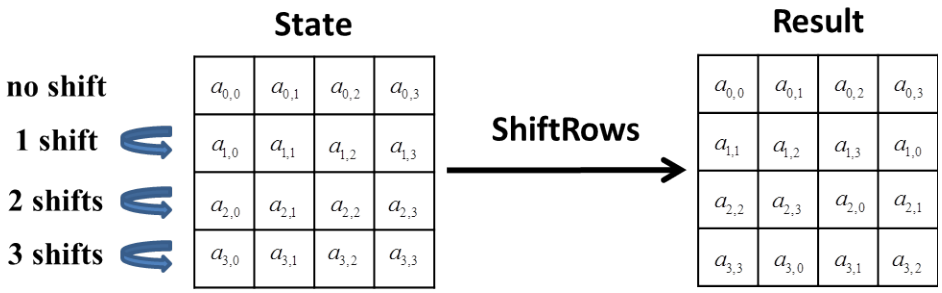


Figure 6 ShiftRows module.

- **MixColumns()**

  Each Word of the updating State is considered as a polynomial over $GF(2^8)$, which is multiplied (denoted by $\bullet$) by a fixed polynomial matrix $c(x)$:

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \bullet \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix},$$

  where $[a_{0,i} \quad a_{1,i} \quad a_{2,i} \quad a_{3,i}]^T$ is the ith Word in the updating State, such that the resulting column $[b_{0,i} \quad b_{1,i} \quad b_{2,i} \quad b_{3,i}]^T$ is calculated by:

$$\begin{aligned} b_{0,i} &= (\{02\} \bullet a_{0,i}) \oplus (\{03\} \bullet a_{1,i}) \oplus a_{2,i} \oplus a_{3,i} \\ b_{1,i} &= a_{0,i} \oplus (\{02\} \bullet a_{1,i}) \oplus (\{03\} \bullet a_{2,i}) \oplus a_{3,i} \\ b_{2,i} &= a_{0,i} \oplus a_{1,i} \oplus (\{02\} \bullet a_{2,i}) \oplus (\{03\} \bullet a_{3,i}) \\ b_{3,i} &= (\{03\} \bullet a_{0,i}) \oplus a_{1,i} \oplus a_{2,i} \oplus (\{02\} \bullet a_{3,i}). \end{aligned}$$

  It is important to keep in mind that the multiplication over $GF(2^8)$ is different from our ordinary multiplication. It involves multiplication of polynomials then modulo an irreducible polynomial of degree 8. Please refer to Chapter 4 of [1] for the mathematical background and the precise calculation steps.

  Fortunately, there are shortcuts for performing the multiplication. Multiplication by $\{02\}$ can be implemented at the Byte level as "a left shift, followed by a conditional bitwise XOR with $\{1b\}$ if the 8th bit before the shift is 1." This module is denoted as `xtime()`. All other multiplier values can be recursively deduced from a combination of $(\{02\} \bullet a)$ and a (which is equivalent to $(\{01\} \bullet a)$), by treating the multiplication symbol ' $\bullet$ ' as multiplication, and XOR operator ' $\oplus$ ' as addition. Therefore, $(\{03\} \bullet a)$ is equivalent to XORing $(\{02\} \bullet a)$ with the multiplicand a itself, such that

$$\begin{aligned} \{02\} \bullet a &= \text{xtime}() \\ \{03\} \bullet a &= (\{02\} \bullet a) \oplus a = \text{xtime}() \oplus a. \end{aligned}$$

  Figure 7 shows the MixColumns module.

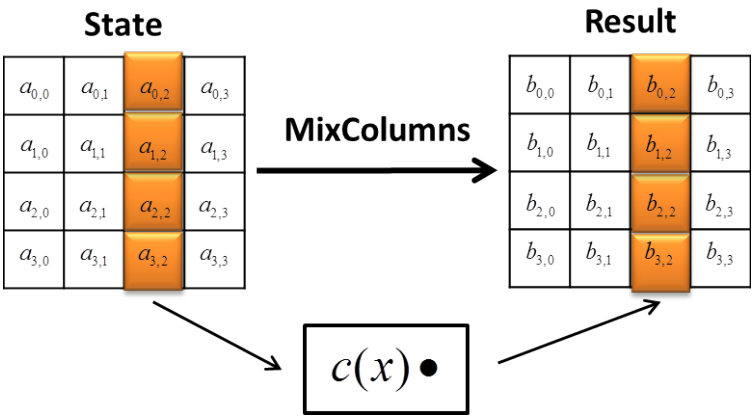


Figure 7 MixColumns module.

AES Decryption Algorithm:

- **AES Decryption Pseudo Code**

  Figure 8 shows the pseudo code for the AES decryption algorithm. The general algorithm flow pretty much remains the same, except for the reversed looping rounds and the inverse of the original modules. Figure 9 shows the decryption algorithm flow.

```
InvAES(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
      byte state[4,Nb]
      state = in
      AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
      for round = Nr-1 step -1 downto 1
          InvShiftRows(state)
          InvSubBytes(state)
          AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
          InvMixColumns(state)
      end for
      InvShiftRows(state)
      InvSubBytes(state)
      AddRoundKey(state, w[0, Nb-1])
      out = state
end
```

Figure 8 Pseudo code for the AES decryption algorithm [1].

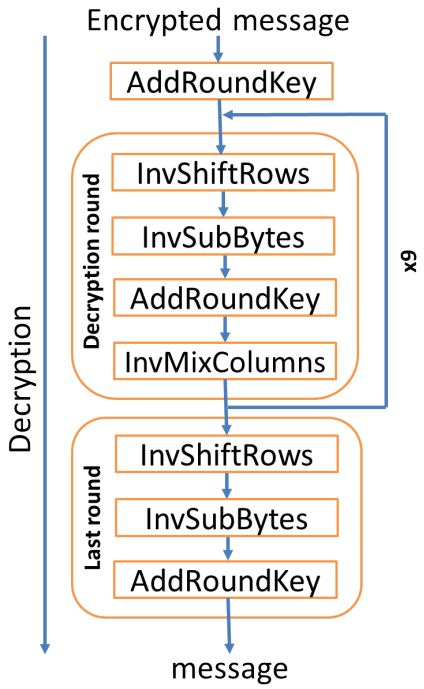


Figure 9 AES decryption algorithm flow.

- **InvSubBytes()**
  The Inverse SubBytes module is identical to the SubBytes module, except that it uses the Inverse S-box for the decryption process. Table 3 shows the lookup table for the Inverse S-box.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1x	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2x	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3x	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4x	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5x	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6x	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7x	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8x	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9x	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
ax	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
bx	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
cx	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
dx	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
ex	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
fx	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 3 Rijndael Inverse S-box.

- **InvShiftRows()**
  The Inverse ShiftRows module is identical to the ShiftRows module, except that it now shifts rightwards instead. Specifically, row n is right-circularly shifted by $n-1$ Bytes.

- **InvMixColumns()**
  Each Word of the updating State is considered as a polynomial over $GF(2^8)$, which is multiplied (denoted by $\bullet$) by a another fixed polynomial matrix $c(x)$:

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \bullet \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix},$$

  where the resulting column is calculated by:

$$\begin{aligned} b_{0,i} &= (\{0e\} \bullet a_{0,i}) \oplus (\{0b\} \bullet a_{1,i}) \oplus (\{0d\} \bullet a_{2,i}) \oplus (\{09\} \bullet a_{3,i}) \\ b_{1,i} &= (\{09\} \bullet a_{0,i}) \oplus (\{0e\} \bullet a_{1,i}) \oplus (\{0b\} \bullet a_{2,i}) \oplus (\{0d\} \bullet a_{3,i}) \\ b_{2,i} &= (\{0d\} \bullet a_{0,i}) \oplus (\{09\} \bullet a_{1,i}) \oplus (\{0e\} \bullet a_{2,i}) \oplus (\{0b\} \bullet a_{3,i}) \\ b_{3,i} &= (\{0b\} \bullet a_{0,i}) \oplus (\{0d\} \bullet a_{1,i}) \oplus (\{09\} \bullet a_{2,i}) \oplus (\{0e\} \bullet a_{3,i}) \end{aligned}.$$

As described previously, multiplications with $\{09\}$, $\{0b\}$, $\{0d\}$ and $\{0e\}$ can be recursively deduced from a combination of $(\{02\} \bullet a)$ and a, where

$$(\{04\} \bullet a_{0,i}) = (\{02\} \bullet (\{02\} \bullet a_{0,i}))$$
$$(\{08\} \bullet a_{0,i}) = (\{02\} \bullet (\{04\} \bullet a_{0,i}))$$
$$(\{09\} \bullet a_{0,i}) = (\{08\} \bullet a_{0,i}) \oplus a_{0,i}$$
$$(\{0b\} \bullet a_{0,i}) = (\{08\} \bullet a_{0,i}) \oplus (\{02\} \bullet a_{0,i}) \oplus a_{0,i}$$
$$(\{0d\} \bullet a_{0,i}) = (\{08\} \bullet a_{0,i}) \oplus (\{04\} \bullet a_{0,i}) \oplus a_{0,i}$$
$$(\{0e\} \bullet a_{0,i}) = (\{08\} \bullet a_{0,i}) \oplus (\{04\} \bullet a_{0,i}) \oplus (\{02\} \bullet a_{0,i}).$$

Oh the other hand, all possible multiplication values can be pre-calculated and placed into lookup tables, just like that of the S-box. Table 4 shows the lookup tables for the multiplication with $\{09\}$, $\{0b\}$, $\{0d\}$, and $\{0e\}$, respectively.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1x	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2x	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3x	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4x	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5x	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6x	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7x	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8x	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9x	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
ax	d7	de	c5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
bx	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
cx	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
dx	0a	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
ex	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
fx	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

(a)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
	00	0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1x	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2x	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3x	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4x	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5x	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6x	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7x	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8x	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9x	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
ax	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
bx	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
cx	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
dx	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
ex	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
fx	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

(b)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1x	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2x	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3x	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4x	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5x	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6x	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7x	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8x	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9x	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
ax	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
bx	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
cx	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
dx	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
ex	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
fx	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

(c)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1x	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2x	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3x	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4x	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5x	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6x	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7x	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8x	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9x	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
ax	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
bx	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
cx	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
dx	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
ex	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
fx	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

(d)

Table 4 Lookup tables for the multiplication with (a) $\{09\}$, (b) $\{0b\}$, (c) $\{0d\}$, and (d) $\{0e\}$.

- **InvAddRoundKey ()**
  Since the original AddRoundKey module only applies an XOR operation, which is its own inverse, the Inverse AddRoundKey module is completely identical to the original AddRoundKey module.

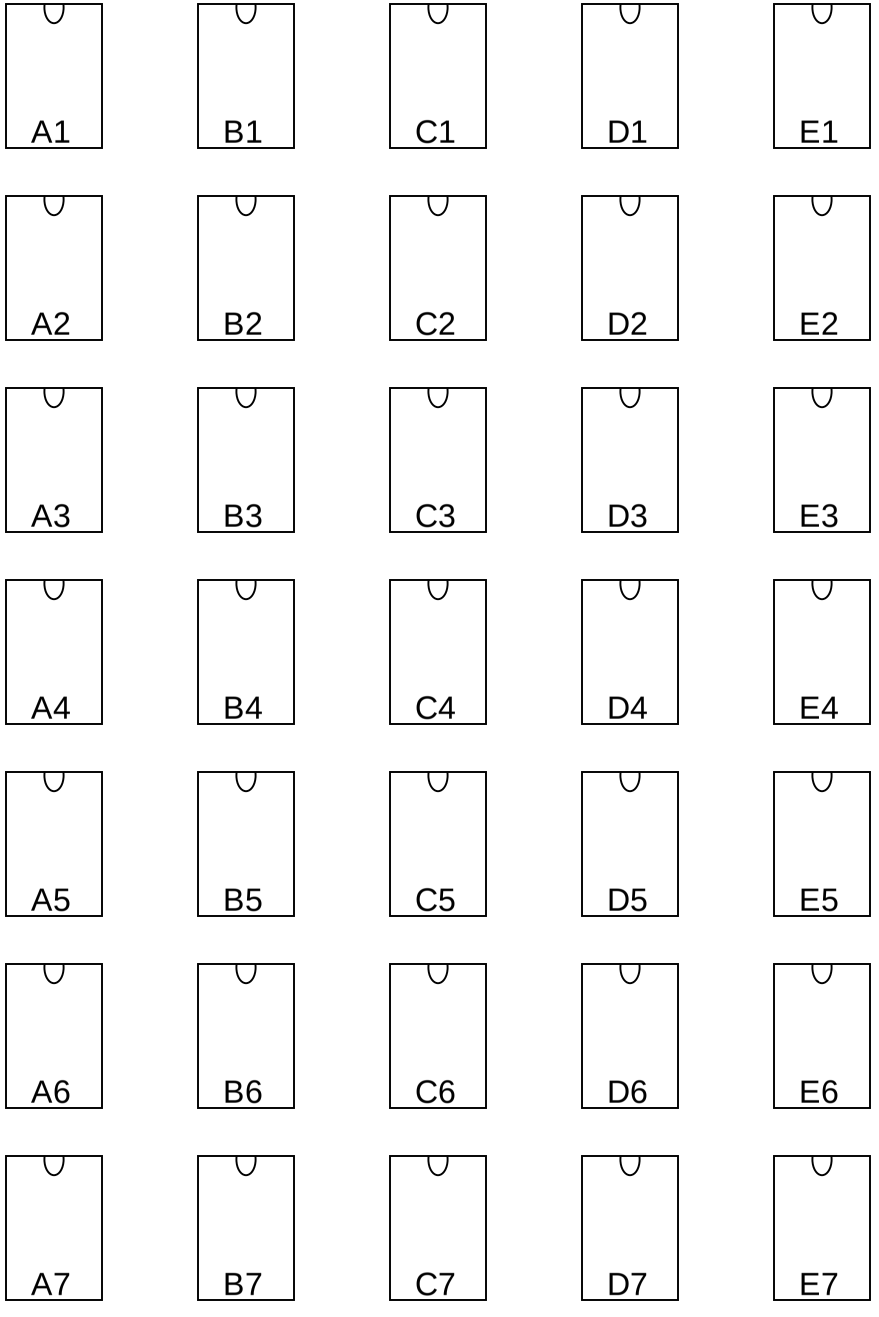
References:

[1] "Announcing the Advanced Encryption Standard (AES)", National Institute of Standards and Technology, 2001. Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

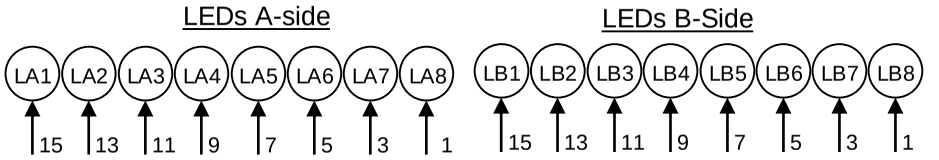
LAYOUT SHEETS

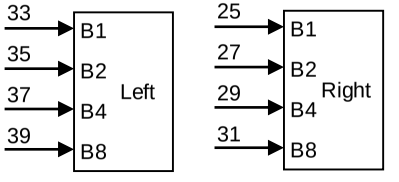
COMPONENT LAYOUT AND I/O ASSIGNMENT

PROTOBOARD

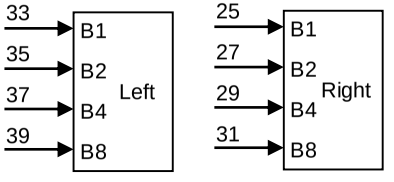


16-bit I/O BOARD

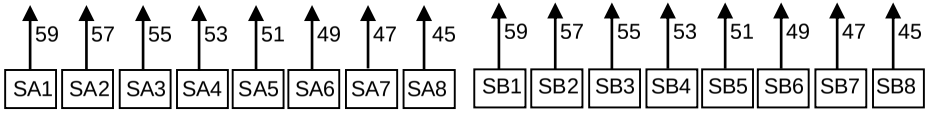




HEX A-side



HEX B-side

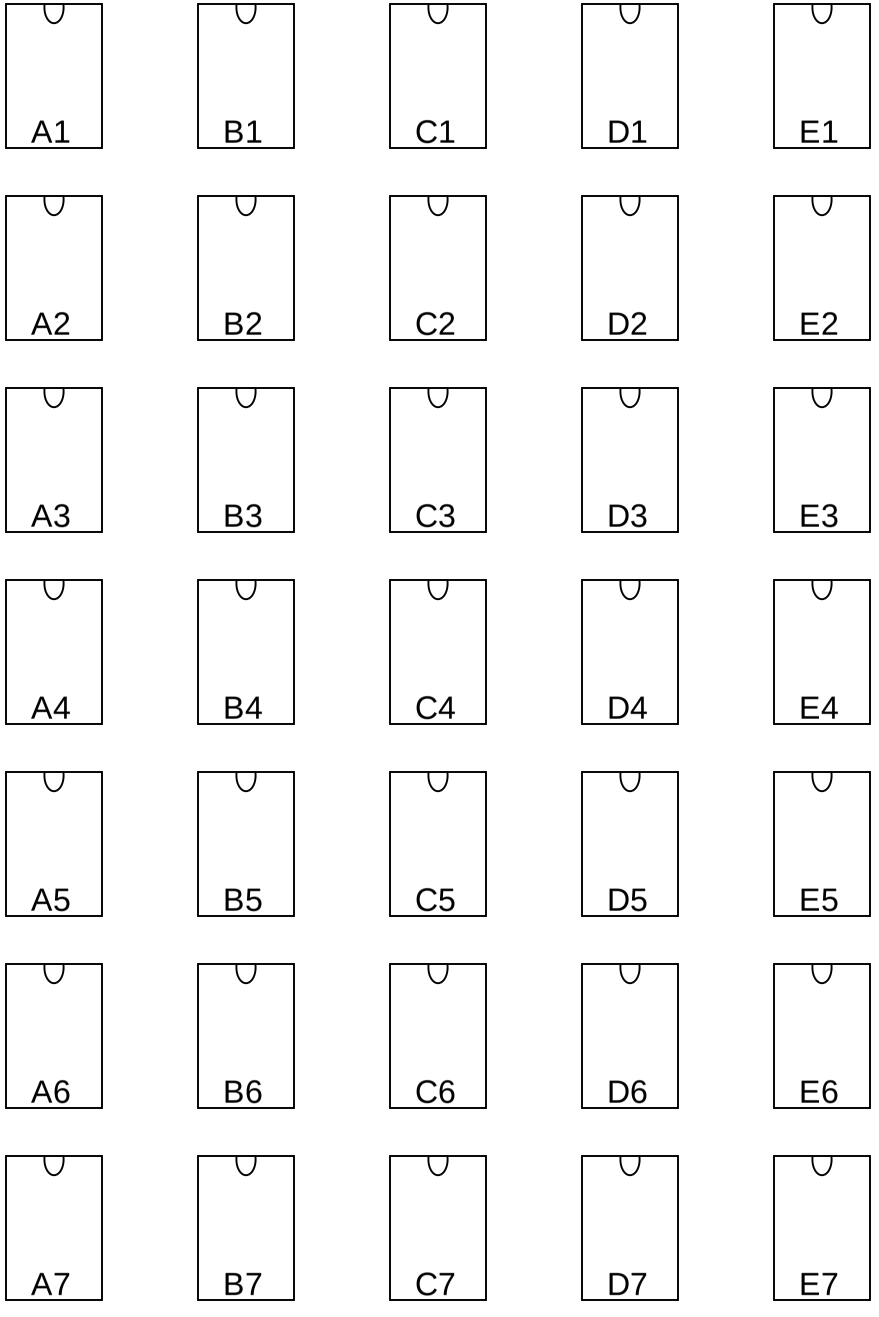


SWITCHES A-side

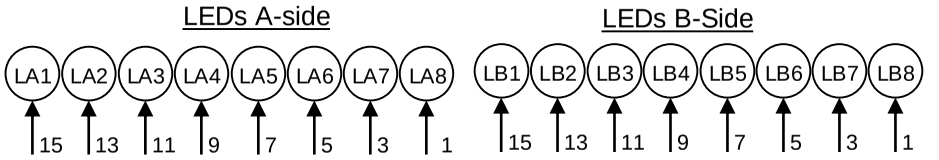
SWITCHES B-side

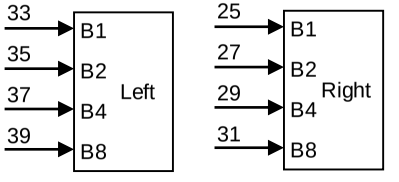
COMPONENT LAYOUT AND I/O ASSIGNMENT

PROTOBOARD

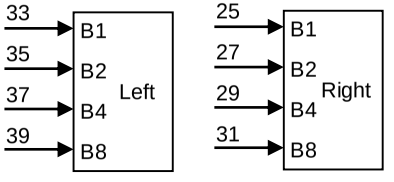


16-bit I/O BOARD

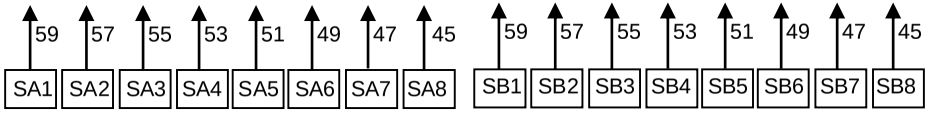




HEX A-side



HEX B-side

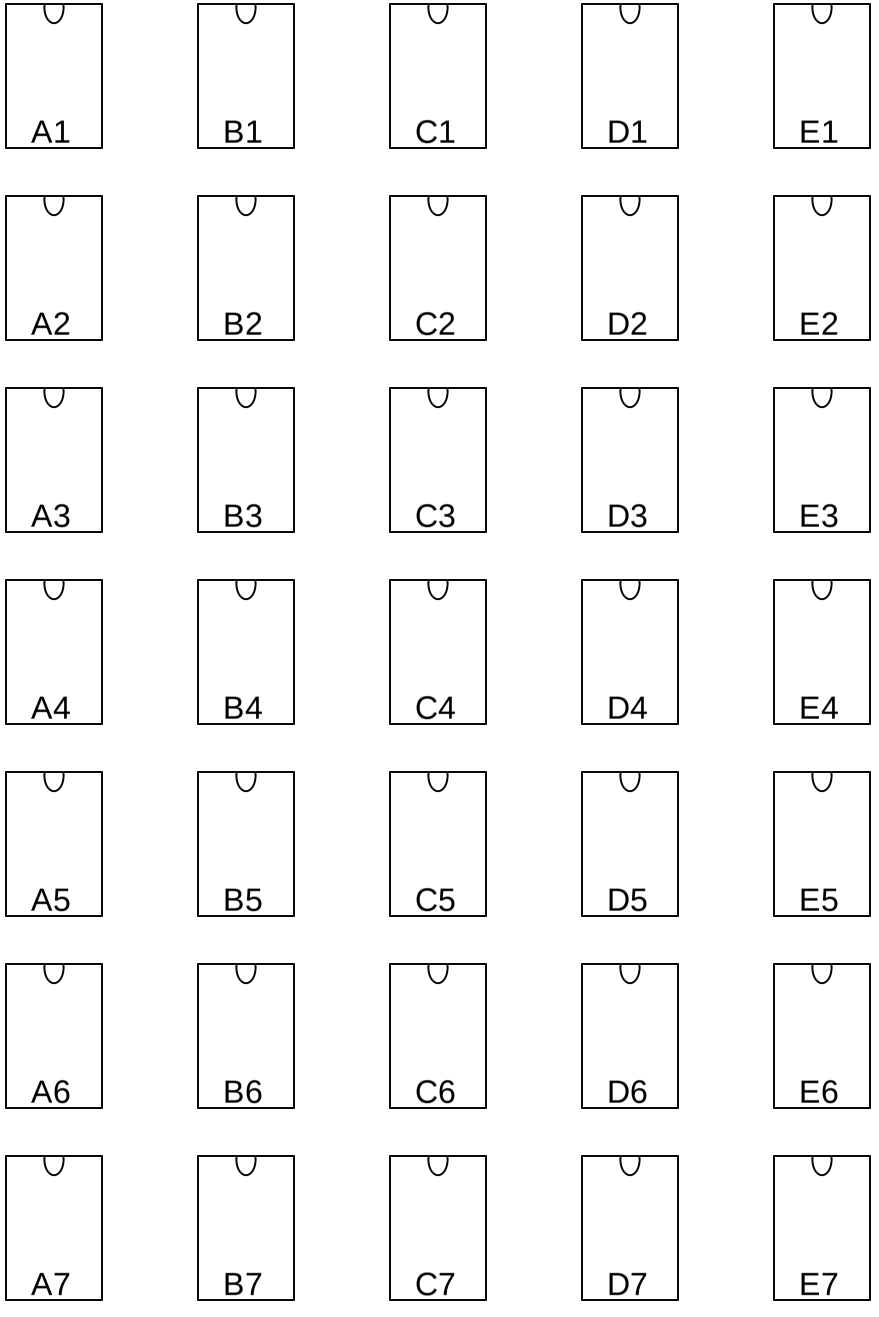


SWITCHES A-side

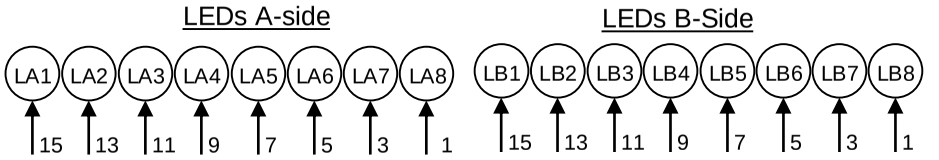
SWITCHES B-side

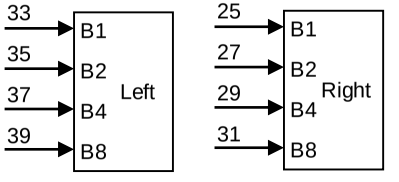
COMPONENT LAYOUT AND I/O ASSIGNMENT

PROTOBOARD

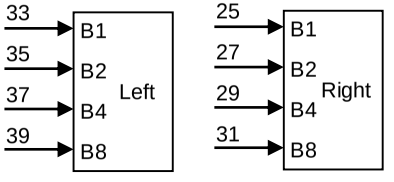


16-bit I/O BOARD

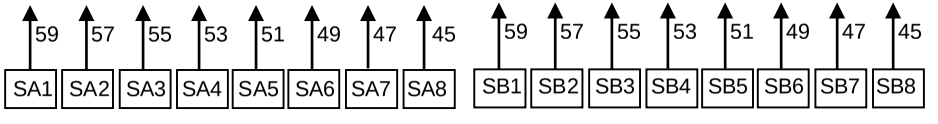




HEX A-side



HEX B-side



SWITCHES A-side

SWITCHES B-side